DevOps Culture and Practice with OpenShift

Deliver continuous business value through people, processes, and technology

Tim Beattie | Mike Hepburn | Noel O'Connor | Donal Spring

Illustrations by Ilaria Doria

DevOps Culture and Practice with OpenShift

Deliver continuous business value through people, processes, and technology

Tim Beattie

Mike Hepburn

Noel O'Connor

Donal Spring

BIRMINGHAM—MUMBAI

DevOps Culture and Practice with OpenShift

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author(s), nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Authors: Tim Beattie, Mike Hepburn, Noel O'Connor, and Donal Spring

Illustrator: Ilaria Doria

Technical Reviewer: Ben Silverman

Managing Editors: Aditya Datar and Siddhant Jain

Acquisitions Editor: Ben Renow-Clarke

Production Editor: Deepak Chavan

Editorial Board: Vishal Bodwani, Ben Renow-Clarke, Edward Doxey, Alex Patterson, Arijit Sarkar, Jake Smith, and Lucy Wan

First Published: August 2021

Production Reference: 1100821

ISBN: 978-1-80020-236-8

Published by Packt Publishing Ltd.
Livery Place, 35 Livery Street,
Birmingham, B3 2PB, UK.

www.packt.com

Praise for DevOps Culture and Practice with OpenShift

"Creating successful, high-performing teams is no easy feat. DevOps Culture and Practice with OpenShift provides a step-by-step, practical guide to unleash the power of open processes and technology working together."

–Jim Whitehurst, President, IBM

"This book is packed with wisdom from Tim, Mike, Noel, and Donal and lovingly illustrated by Ilaria. Every principle and practice in this book is backed by wonderful stories of the people who were part of their learning journey. The authors are passionate about visualizing everything and every chapter is filled with powerful visual examples. There is something for every reader and you will find yourself coming back to the examples time and again."

–Jeremy Brown, Chief Technology Officer/Chief Product Officer at Traveldoo, an Expedia Company

"This book describes well what it means to work with Red Hat Open Innovation Labs, implementing industrial DevOps and achieving business agility by listening to the team. I have experienced this first hand. Using the approach explained in this book, we have achieved a level of collaboration and engagement in the team we had not experienced before, the results didn't take long and success is inevitable. What I have seen to be the main success factor is the change in mindset among team members and in management, which this approach helped us drive."

–Michael Denecke, Head of Test Technology at Volkswagen AG

"This book is crammed full to the brim with experience, fun, passion, and great practice. It contains all the ingredients needed to create a high performance DevOps culture...it's awesome!"

–John Faulkner-Willcocks, Head of Coaching and Delivery Culture, JUST

"DevOps has the opportunity to transform the way software teams work and the products they deliver. In order to deliver on this promise, your DevOps program must be rooted in people. This book helps you explore the mindsets, principles, and practices that will drive real outcomes."

–Douglas Ferguson, Voltage Control Founder, Author of Magical Meetings and Beyond the Prototype

"Fun and intense to read! Somehow, the authors have encapsulated the Red Hat culture and expression in this book."

–Jonas Frydal, Director at Volvo Cars

"This book is really valuable for me. I was able to map every paragraph I read to the journey we took during the residency with Red Hat Open Innovation Labs. It was such an intense but also rewarding time, learning so much about culture, openness, agile and how their combination can make it possible to deliver crucial business value in a short amount of time.

Speaking from my personal experience, we enabled each other, my team bringing the deep knowledge in the industry and Red Hat's team bringing good practices for cloud-native architectures. This made it possible to reinvent how vehicle electronics technology is tested while pushing Red Hat's OpenShift in an industrial DevOps direction.

I am looking forward to keeping a hard copy of the book at my desk for easy review."

—Marcus Greul, Program Manager at CARIAD, a Volkswagen Group company

"Innovation requires more than ideas and technology. It needs people being well led and the 'Open Leadership' concepts and instructions in DevOps Practice and Culture with OpenShift should be required reading for anyone trying to innovate, in any environment, with any team."

—Patrick Heffernan, Practice Manager and Principal Analyst,
Technology Business Research Inc.

"Whoa! This has to be the best non-fiction DevOps book I've ever read. I cannot believe how well the team has captured the essence of what the Open Innovation Labs residency is all about. After reading, you will have a solid toolbox of different principles and concrete practices for building the DevOps culture, team, and people-first processes to transform how you use technology to act as a force multiplier inside your organization."

—Antti Jaakkonen, Lean Agile Coach, DNA Plc

"Fascinating! This book is a must-read for all tech entrepreneurs who want to build scalable and sustainable companies. Success is now handed to you."

—Jeep Kline, Venture Capitalist, Entrepreneur

"In a digital-first economy where technology is embedded in every business, innovation culture and DevOps are part and parcel of creating new organizational values and competitive advantages. A practical and easy to understand guide for both technology practitioners and business leaders is useful as companies accelerate their ***Digital Transformation*** *(**DX**) strategies to thrive in a changed world."*

—Sandra Ng, Group Vice President, ICT Practice

"DevOps Culture and Practice with OpenShift is a distillation of years of experience into a wonderful resource that can be used as a recipe book for teams as they form and develop, or as a reference guide for mature teams as they continue to evolve."

—David Worthington, Agile Transformation Coach, DBS Bank, Singapore

Table of Contents

Foreword i

Preface iii

Acknowledgements ix

Section 1: Practices Make Perfect 1

Chapter 1: Introduction — Start with Why 3

Why — For What Reason or Purpose? 4
Why Should I Listen to These Folks? 5
Where Did This Book Come From? 6
Who Exactly Is This Book For? 8
From I to T to M 10
Conclusion 11

Chapter 2: Introducing DevOps and Some Tools 13

The Value Chain 14
The Gaps 16
The Big List of Things to Do 16
Demonstrating Value and Building the Right Thing 17
How Do We Do the Things on Our List? 18
Development to Operations 22
People, Process, and Technology 24
The Mobius Loop and the Open Practice Library 26
Conclusion 32

Chapter 3: The Journey Ahead 33

A Story about Telling a Practice 35
PetBattle – the Backstory 36
What about Legacy Systems? 37
Borrowing Brilliance 38
What to Expect from the Rest of This Book? 38
What about Distributed Teams? 43
Some Words about the World of 'Open' 44
Conclusion 45

Section 2: Establishing the Foundation 47

Chapter 4: Open Culture 53

Why Is It Important? 54
Information Radiators 56
Can You Make Those Red Lights Go Green, Please? 56
Culture 57
Motivation 58
PetBattle — Creating Autonomy, Mastery, and Purpose 60
Social Contracts 61
Do I Need One? If So, How Do I Build One? 63
It's OK to Be Wrong 67
Social Contracting for Distributed People 68
Stop the World 70
The Andon Cord and Psychological Safety 71
We're Just Rebuilding the Same Experience. Stop the World! 72
Losing Track of Original Purpose 73

Real-Time Retrospective ... 75
Team Identity ... 79
Socializing ... 80
Network Mapping ... 81
Team Logo and Prime Directive ... 83
Team Name + Building a Team Logo = the Beginning of Team Identity ... 84
Creating a Team Identity with Distributed People ... 85
Radiate Everything ... 86
Radiating Everything When Distributed ... 88
Team Sentiment ... 89
Blending Team Sentiment with Other Practices ... 90
Team Sentiment Achieving a Different Purpose – Banter! ... 92
Team Sentiment with Distributed People ... 93
Radiate Failures ... 93
Radiating Failure – as Useful (If Not More) as Radiating Success ... 94
Inspect and Adapt ... 96
PetBattle — Establishing the Cultural Foundation ... 97
Conclusion ... 99
Chapter 5: Open Environment and Open Leadership 101
The Kodak Problem ... 103
Learning from History ... 105
Open Leadership ... 105
Changing an Organization ... 106
Leading Sustainable Change ... 107
Achieving Greatness ... 109
Giving Intent ... 109
Moving Decisions to Where the Information Is ... 109

Setting the Environment ... 109
How Do We (as Leaders) Convince the Doubters? ... 110
No Computers in the Company! The 1990s or the 1890s? ... 111
Priority Sliders ... 112
Running Priority Sliders with Distributed People ... 116
The Space ... 117
The Minimal Viable Space ... 120
"We See What You Want To Do and Why and We'll Help You Get There" in Just 4 Weeks ... 121
Virtual Spaces ... 123
Conclusion ... 125
Chapter 6: Open Technical Practices – Beginnings, Starting Right 127
Green from Go! ... 129
Pair Programming and Mob Programming ... 130
Mob to Learn, Pair to Build ... 131
Containers and Being Container-Native ... 133
Container History ... 133
How Containers Work ... 134
Pipelines — CI or CD or CD²? ... 137
Derek the DevOps Dinosaur ... 137
Continuous Integration ... 144
Integrate Continuously ... 145
Continuous Delivery ... 146
Building Confidence in the Quality of the Software Delivery Pipeline ... 147
Continuous Deployment (CD²) ... 149
When the Work Is Done, Ship It! ... 150

Everything-as-Code 152
Can You Build a Second One of Those for Me, Please? 154
Establishing the Technical Foundation for PetBattle 156
Jenkins – Our Best Friend! 157
Helm Overview 158
Installing Jenkins Using Helm 160
Developer Workflows 165
GitFlow 165
GitHub Flow 166
Trunk-Based Development 167
Too Many Choices — Tell Me What to Do 168
Conclusion 170
Chapter 7: Open Technical Practices — The Midpoint 171
The Big Picture 172
PetBattle – Building a Big Picture 175
GitOps 180
ArgoCD 181
If It's Not in Git, It's Not Real! 182
Implementing GitOps 184
Testing Testing Testing! 194
The Test Automation Pyramid 195
Testing in Practice 196
Testing and the Definition of Done 198
TDD or BDD or DDT 199
BDD for Our Ops Tooling Python Library 202
Product Owners Seeing Their Thoughts in Code! 204
Example Mapping 204

Example Mapping in the Field 205
Non-functional Testing 207
Performance Testing Sam's Code 208
A Few Final Thoughts on Testing 211
Emerging Architecture 211
Observations from the Field 213
Conclusion 217

Section 3: Discover It 221

Chapter 8: Discovering the Why and Who 223

The North Star 225
PetBattle as a Business 230
Our North Star at Open Innovation Labs 232
Impact Mapping 233
Start with the WHY — the Goal 235
PetBattle – the Goal 237
WHO Can Help Us Reach the Desired Effect? The Actors 239
PetBattle – the Actors 239
HOW Should Our Actors' Behaviors Change? The Impacts 241
PetBattle – the Impacts 241
WHAT Should We Build? The Deliverables 243
PetBattle – the Deliverables 244
PetBattle – Placing Bets 248
Hypothesis Examples 251
Connecting Engineers to Business Outcomes 253

Human-Centered Design ... 255
UX Design and Empathy Mapping a PetBattle User ... 259
Users Do Strange and Unexpected Things ... 261
Empathy Mapping an Organization — Dev versus Ops ... 263
Engineers Build Out Empathy Maps during User Interviews ... 265
Conclusion ... 266
Chapter 9: Discovering the How 269
Event Storming ... 270
What Is Event Storming? ... 271
The Ingredients ... 273
The Recipe ... 274
Event Storming with Doubters ... 285
PetBattle Event Storm ... 287
Final Thoughts on Event Storming ... 297
Emerging Architecture ... 297
Transitioning an Event Storm to an Emergent Architecture ... 299
The Non-Functional Map ... 304
From Non-Functional Map to Backlog ... 305
Discovering the Case for Continuous Delivery ... 308
Metrics-Based Process Map ... 309
Finding and Making Improvements ... 311
Improving through Iteration ... 312
Scoping an Entire Engagement Using MBPM ... 313
PetBattle – MBPM ... 316
Conclusion ... 319

Chapter 10: Setting Outcomes 321

What Is an Outcome? 322

Outcomes versus Outputs 323

Why Have Target Outcomes? 324

How to Capture Target Outcomes 325

Examples of Target Outcomes 327

Visualizing Target Outcomes 329

Optimizing Target Outcomes 330

Chaining Target Outcomes with Other Practices 331

PetBattle Target Outcomes 332

The Balance of Three: People/Process/Technology 335

Target Outcomes from a Telecoms Product – Stopwatch at the Ready! 337

Differentiating between Primary Outcomes and Enabling Outcomes 338

Software Delivery Metrics 340

Platform Adoption Metrics 341

Continuous Metrics Inspection 342

Creating a Discovery Map 343

Conclusion 346

Section 4: Prioritize It 349

Chapter 11: The Options Pivot 353

Value Slicing 355

The Beer and the Curry 360

One to Few to Many Slices of Value – Continuous Delivery 362

PetBattle – Slicing Value towards Continuous Delivery 366

Design of Experiments 373

Qualitative versus Quantitative Feedback 374

Impact and Effort Prioritization Matrix 377

How-Now-Wow Prioritization 379

The Design Sprint 382

Forming the Initial Product Backlog 385

PetBattle — Tracing Value through Discovery and Delivery Practices 388

Product Backlog Refinement 389

Prioritization 391

Value versus Risk 391

Cost of Delay and WSJF 392

PetBattle – Prioritizing using WSJF 394

Product Ownership 396

Experimenting with Different Product Owners 398

Patterns of Early Sprints and the Walking Skeleton 399

Advanced Deployment Considerations 400

A/B Testing 401

Blue/Green Deployments 402

Canary Releases 403

Dark Launches 404

Feature Flags 405

PetBattle – Tech Spikes, Prototypes, Experiments, and Feature Implementations 406

Reframing the Question – How Much Can I Borrow or How Much House Can I Afford? 408

Research, Experiment, Implement 409

Creating an Options Map 410

Conclusion 412

Section 5: Deliver It 415

Chapter 12: Doing Delivery 419

Waterfall 421
The Birth of Agile 424
How Does OpenShift Help? 427
Decision-Making Contexts 428
The Cynefin Framework 428
The Ferrari and the Rainforest 430
When Does a Mobius Loop Mindset Make Sense? 432
PetBattle—Complex, Complicated, or Clear? 433
The Definition of Ready 435
PetBattle – Definition of Ready 439
Scrum 440
The 3-5-3 Formation 441
The Product Owner Role 442
The ScrumMaster Role 443
The Development Team Role 444
The Product Backlog Artifact 445
The Sprint Backlog Artifact 447
The Product Increment Artifact 450
Show Me the Product! 450
The Sprint Planning Event 451
The Daily Scrum Event 453
The Sprint Review Event 455
When WOULD We Have Uncovered This In a Traditional Mode of Delivery? 457
The Sprint Retrospective Event 458

The Pub Retro! 463
A Sprint in the Life of PetBattle: Getting Ready 465
A Sprint in the Life of PetBattle: Sprint 1 Planning 467
A Sprint in the Life of PetBattle: Sprint 1 Delivery 470
A Sprint in the Life of PetBattle: Sprint 1 Review And Retrospective 471
Using Scrum with distributed people 472
When should we stop Scrumming? 474
Teams asking questions that suggest we've matured out of Scrum 474
Kanban 475
Kanban Board! 476
PetBattle – Release Early, Release Often, Release Continuously 478
The Definition of Done 478
PetBattle – Definition of Done 479
Bad Agile Smells 482
Conclusion 483
Chapter 13: Measure and Learn 485
Metrics-Driven Transformation 486
Where to Measure and Learn 487
The Showcase 488
The Retrospective 488
The Retrospective – an Engineering Perspective 489
Inspecting the Build Stats at Retrospectives 491
Experiments – the Results! 492
User Testing 493
Usability Testing 493
"We Are Not Our Users" 494
Guerrilla Testing 496

Guerrilla testing with a box of donuts in a busy Dublin bank! ... 496
PetBattle Usability Testing ... 497
What to Measure? ... 499
Measuring Service Delivery and Operational Performance (SDO) ... 499
Pelorus ... 502
Measuring Lean Metrics ... 502
Measuring SLOs, SLAs, and SLIs ... 503
PetBattle Service Levels ... 504
Measuring Security ... 505
PetBattle Security ... 506
Measuring Performance ... 507
PetBattle Performance ... 508
Measuring Deployment Pain ... 509
Measuring Culture ... 510
Measuring Application Metrics ... 511
PetBattle Application Metrics ... 511
Measuring Infrastructure Platform Costs and Utilization ... 512
Measuring Resources and Services ... 514
User Experience Analytics ... 515
PetBattle User Experience Analytics ... 516
Visualize Measurable Outcomes ... 516
Proactive Notification ... 517
Altering the Customers ... 518
Having Fun with Notifications and the Build! ... 518
Creating a Delivery Map ... 522
Conclusion ... 524

Section 6: Build It, Run It, Own It 527

Chapter 14: Build It 531

Cluster Resources 533
Existing PetBattle Architecture 533
PetBattle Components 534
Plan of Attack 537
Running PetBattle 538
Argo CD 543
Trunk-Based Development and Environments 545
The Anatomy of the App-of-Apps Pattern 546
Build It – CI/CD for PetBattle 549
The Big Picture 549
The Build 551
The Bake 552
The Deploy 553
System Test 554
Promote 555
Choose Your Own Adventure 558
Jenkins–The Frontend 559
Connect Argo CD to Git 559
Secrets in Our Pipeline 563
The Anatomy of a Jenkinsfile 572
Branching 579
Webhooks 580
Jenkins 581

Bringing It All Together 581
What's Next for Jenkinsfile 584
Tekton–The Backend 585
Tekton Basics 585
Reusable Pipelines 588
Build, Bake, Deploy with Tekton 589
Triggers and Webhooks 593
GitOps our Pipelines 595
Which One Should I Use? 596
Conclusion 598

Chapter 15: Run It 599

The Not Safe For Families (NSFF) Component 600
Why Serverless? 600
Generating or Obtaining a Pre-trained Model 601
The OpenShift Serverless Operator 603
Deploying Knative Serving Services 604
Invoking the NSFF Component 607
Let's Talk about Testing 611
Unit Testing with JUnit 612
Service and Component Testing with REST Assured and Jest 613
Service Testing with Testcontainers 616
End-to-End Testing 617
Pipelines and Quality Gates (Non-functionals) 621
SonarQube 621
Perf Testing (Non-Functional) 627
Resource Validation 633
Image Scanning 636

Linting 638
Code Coverage 639
Untested Software Watermark 642
The OWASP Zed Attack Proxy (ZAP) 643
Chaos Engineering 644
Accidental Chaos Testing 646
Advanced Deployments 648
A/B Testing 649
The Experiment 649
Matomo – Open Source Analytics 650
Deploying the A/B Test 652
Understanding the results 655
Blue/Green deployments 656
Deployment previews 658
Conclusion 660
Chapter 16: Own It 661
Observability 661
Probes 662
Domino Effect 664
Fault Tolerance 664
Logging 665
Tracing 666
Metrics 666
Configuring Prometheus To Retrieve Metrics From the Application 669
Visualizing the Metrics in OpenShift 671
Querying using Prometheus 671
Visualizing Metrics Using Grafana 672

Metadata and Traceability 673
Labels 673
Software Traceability 676
Annotations 677
Build Information 677
Alerting 678
What Is an Alert? 678
Why Alert? 678
Alert Types 679
Managing Alerts 680
User-Defined Alerts 680
OpenShift Alertmanager 684
Service Mesh 685
Why Service Mesh? 685
Aside – Sidecar Containers 686
Here Be Dragons! 687
Service Mesh Components 688
PetBattle Service Mesh Resources 689
Operators Everywhere 693
Operators Under the Hood 695
Control Loops 695
Operator Scopes 696
Operators in PetBattle 697
Service Serving Certificate Secrets 700
Conclusion 701

Section 7: Improve It, Sustain It 703

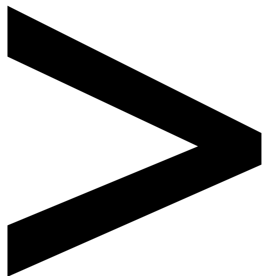
Chapter 17: Improve It 705

What Did We Learn? 707
Did We Learn Enough? 708
We Need Two Apps, Not One! 709
"Just Enough" Leads to Continuous Everything 710
Learning from Security Experts 712
Always Improve Metrics and Automation 713
Revisiting the Metrics-Based Process Map 714
My management only really understand numbers and spreadsheets 714
Improve the Technology 718
Long Live the Team 719
Visualizing the Transition from I to T to M 719
Wizards and Cowboys 721
Conclusion 722

Chapter 18: Sustain It 723

The Journey So Far 724
Infectious Enthusiasm 726
Demo Day 728
Documenting the Journey 730
Sketching the Experience 731
Walk the Walls 732
Written Showcases 733
Word of Mouth 734
Mind-Blowing Metrics That Cannot Be Ignored 734
Transitioning From One Team To Seven Teams 735

More Teams, More Application Products 738
The Power of Three Iterations in Enablement 739
The App Does Something This Week That It Didn't Do Last Week! 740
Bolster the Foundations 741
Sustaining the Technology 742
The Double Mobius Loop – Platform and Application Product 747
Connecting Many Levels of Product Teams 749
Conclusion 752
Appendix A – OpenShift Sizing Requirements for Exercises 755
How To Resize Storage in Your CRC Virtual Machine 758
Tekton Persistent Storage 758
Appendix B – Additional Learning Resources 761
Index 763



Foreword

Over the past two decades, as the popularity of Agile and Lean approaches hit the mainstream, many new frameworks have emerged, each promising to solve your problems if you just bought their secret sauce. Yet the pioneers in the early days didn't get the answers handed to them; instead, they had to figure out the recipes through trial and error. This relentless discovery and invention process led to great leaps forward; it drove the most innovative companies in the world. So the question is, why did we stop reinventing? When was it enough to follow guidelines rather than constantly evolve and grow?

A common problem for organizations is to stay competitive in a constantly evolving market. Competitors emerge fast and disrupt the playing field. To deal with this challenge, organizations hire expensive creative agencies to run ideation workshops in the hope that they can spark new ideas and future proof their organization. But it doesn't stick. Bringing in someone else's creative talent that leaves when the workshops are over doesn't breed an innovation culture.

Red Hat recognized that to help their clients innovate, a model was needed that could be fully customized and adapted it to their needs. A model that would help organizations build their own innovation culture. To help people learn to fish rather than fishing for them. By blending Mobius, an open innovation model, with Red Hat's open culture, organizations can create their own process that builds up their innovation muscle. That creates their own process, develops their own people, and applies technology in novel ways to achieve their desired outcomes faster.

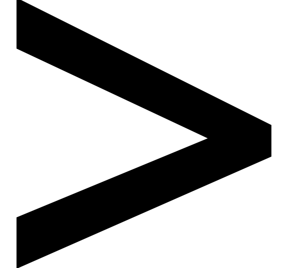
With pragmatic stories from the trenches, the team at Red Hat Open Innovation Labs has created an essential handbook. It takes you on a journey from Day One, from setting up the workspace to practical tips on getting a team to gel and collaborate on their real-world challenges. We get to see under the covers how the DevOps culture emerges through stories and photos. Rather than keeping their secret sauce recipe, Red Hat is following its own principles of being open and sharing its knowledge in a pragmatic, easy-to-follow way.

This book brings together the key ingredients: the people, processes, and technology. It's like having a great travel guide that gives you the tips you need when you need them. I also love that the authors speak with candor and share their real-world war stories, including the mistakes and pitfalls.

The last thing I will say is that the idea of fun is integral to the book, from the simple how-to guides to the engaging illustrations and photos. This book is the culmination of the learning collected along the way and I hope this book brings forth great ideas that can help shape the future and create not only awesome products, but awesome organizations.

Gabrielle Benefield

Founder, Mobius Loop



Preface

About

This section briefly introduces the authors, the coverage of this book, the skills you'll need to get started, and the hardware and software needed to complete all of the technical topics.

About DevOps Culture and Practice with OpenShift

DevOps Culture and Practice with OpenShift features many different real-world practices - some people-related, some process-related, some technology-related - to facilitate successful DevOps, and in turn OpenShift, adoption within your organization. It introduces many DevOps concepts and tools to connect culture and practice through a continuous loop of discovery, pivots, and delivery underpinned by a foundation of collaboration and software engineering.

Containers and container-centric application lifecycle management are now an industry standard, and OpenShift has a leading position in a flourishing market of enterprise Kubernetes-based product offerings. *DevOps Culture and Practice with OpenShift* provides a roadmap for building empowered product teams within your organization.

This guide brings together lean, agile, design thinking, DevOps, culture, facilitation, and hands-on technical enablement all in one book. Through a combination of real-world stories, a practical case study, facilitation guides, and technical implementation details, *DevOps Culture and Practice with OpenShift* provides tools and techniques to build a DevOps culture within your organization on Red Hat's OpenShift Container Platform.

About the authors

Tim Beattie is Global Head of Product and a Senior Principal Engagement Lead for Red Hat Open Innovation Labs. His career in product delivery spans 20 years as an agile and lean transformation coach - a continuous delivery & design thinking advocate who brings people together to build meaningful products and services whilst transitioning larger corporations towards business agility. He lives in Winchester, UK, with his wife and dog, Gerrard the Labrador (the other Lab in his life) having adapted from being a cat-person to a dog-person in his 30s.

Mike Hepburn is Global Principal Architect for Red Hat Open Innovation Labs and helps customers transform their ways of working. He spends most of his working day helping customers and teams transform the way they deliver applications to production with OpenShift. He co-authored the book "DevOps with OpenShift" and loves the outdoors, family, friends, good coffee, and good beer. Mike loves most animals, not the big hairy spiders (Huntsman) found in Australia, and is generally a cat person unless it's Tuesday, when he is a dog person.

Noel O'Connor is a Senior Principal Architect in Red Hat's EMEA Solutions Practice specializing in cloud native application and integration architectures. He has worked with many of Red Hat's global enterprise customers in both Europe, Middle East & Asia. He co-authored the book "DevOps with OpenShift" and he constantly tries to learn new things to varying degrees of success. Noel prefers dogs over cats but got overruled by the rest of the team.

Donal Spring is a Senior Architect for Red Hat Open Innovation Labs. He works in the delivery teams with his sleeves rolled up tackling anything that's needed - from coaching and mentoring the team members, setting the technical direction, to coding and writing tests. He loves technology and getting his hands dirty exploring new tech, frameworks, and patterns. He can often be found on weekends coding away on personal projects and automating all the things. Cats or Dogs? He likes both :)

About the illustrator

Ilaria Doria is an Engagement Lead and Principal at Red Hat Open Innovation Labs. In 2013, she entered into the Agile arena becoming a coach and enabling large customers in their digital transformation journey. Her background is in end-user experience and consultancy using open practices to lead complex transformation and scaling agile in large organizations. Colorful sticky notes and doodles have always been a part of her life, and this is why she provided all illustrations in the book and built all digital templates. She is definitely a dog person.

About the reviewer

Ben Silverman is currently the Chief Architect for the Global Accounts team at Cincinnati Bell Technology Services. He is also the co-author of the books *OpenStack for Architects*, *Mastering OpenStack*, *OpenStack – Design and Implement Cloud Infrastructure*, and was the Technical Reviewer for *Learning OpenStack* (Packt Publishing).

When Ben is not writing books he is active on the Open Infrastructure Superuser Editorial Board and has been a technical contributor to the Open Infrastructure Foundation Documentation Team (Architecture Guide). He also leads the Phoenix, Arizona Open Infrastructure User Group. Ben is often invited to speak about cloud and Kubernetes adoption, implementation, migration, and cultural impact at client events, meetups, and special vendor sessions.

Learning Objectives

- Implement successful DevOps practices and in turn OpenShift within your organization
- Deal with segregation of duties in a continuous delivery world
- Understand automation and its significance through an application-centric view
- Manage continuous deployment strategies, such as A/B, rolling, canary, and blue-green
- Leverage OpenShift's Jenkins capability to execute continuous integration pipelines
- Manage and separate configuration from static runtime software
- Master communication and collaboration enabling delivery of superior software products at scale through continuous discovery and continuous delivery

Audience

This book is for anyone with an interest in DevOps practices with OpenShift or other Kubernetes platforms.

This DevOps book gives software architects, developers, and infra-ops engineers a practical understanding of OpenShift, how to use it efficiently for the effective deployment of application architectures, and how to collaborate with users and stakeholders to deliver business-impacting outcomes.

Approach

This book blends to-the-point theoretical explanations with real-world examples to enable you to develop your skills as a DevOps practitioner or advocate.

Hardware and software requirements

There are five chapters that dive deeper into technology. *Chapter 6*, *Open Technical Practices - Beginnings, Starting Right* and *Chapter 7*, *Open Technical Practices - The Midpoint* focuses on boot-strapping the technical environment. *Chapter 14*, *Build It*, *Chapter 15*, *Run It*, and *Chapter 16*, *Own It* cover the development and operations of features into our application running on the OpenShift platform.

We recommend all readers, regardless of their technical skill, explore the concepts explained in these chapters. Optionally, you may wish to try some of the technical practices yourself. These chapters provide guidance in how to do that.

The OpenShift Sizing requirements for running these exercises are outlined in Appendix A.

Conventions

Code words in the text, database names, folder names, filenames, and file extensions are shown as follows:

We are going to cover the basics of component testing the PetBattle user interface using Jest. The user interface is made of several components. The first one you see when landing on the application is the home page. For the home page component, the test class is called `home.component.spec.ts`:

```
describe('HomeComponent', () => {
  let component: HomeComponent;
  let fixture: ComponentFixture<HomeComponent>;

  beforeEach(async () => {...
  });

  beforeEach(() => {...
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

Downloading resources

All of the technology artifacts are available in this book's GitHub repository at https://github.com/PacktPublishing/DevOps-Culture-and-Practice-with-OpenShift/

High resolution versions of all of the visuals including photographs, diagrams and digital artifact templates used are available at https://github.com/PacktPublishing/DevOps-Culture-and-Practice-with-OpenShift/tree/master/figures

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

We are aware that technology will chage over time and APIs will evolve. For the latest changes of technical content, have a look at the book's GitHub repository above. If you want to contact us directly for any issue you've encountered, please raise an issue in this repository.

Acknowledgements

First and foremost, we'd like to thank those on the front line who dealt with and are dealing with COVID-19 and its impact. Their incredible contributions to maintaining and strengthening our communities cannot be overstated.

We'd also like to thank those in the Open Source community who collaborate and contribute to make all the products we use better every day. This includes the many contributors to the Open Practice Library[1], with special thanks to the individuals who have driven the Open Practice Library from its initial idea to where it is now including Justin Holmes, Ryan de Beasi, Matt Takane, Riley Ghiles, Jerry Becker, and Donna Benjamin. We thank the Mobius Outcome Delivery community who have evolved an inspiring mental model and navigators through complexity with extra special thanks to the founder of this community, Gabrielle Benefield, for all her support in providing the framework that we've anchored this book around.

We'd like to thank our internal and external reviewers for their effort in keeping us on the straight and narrow, correcting us, challenging us, and driving continuous improvement into this book - Chris Baynham-Hughes, Charley Beattie, Donna Benjamin, Jeremy Brown, Margaret Dineen, Cansu Kavili Oernek, David Eva, Oli Gibson, Kari Mayhew, Brid Mackey, Ed Seymour, Mike Walker, and the whole team at Packt Publishing.

Thank you to our many colleagues and customers of Red Hat, Red Hat Open Innovation Labs, and of our previous companies for providing us with our experiences, stories and tips that we have shared in this book.

1 https://github.com/openpracticelibrary/openpracticelibrary/graphs/contributors

On a personal level:

Tim - Thank you so much to my wife, Charley, for being such an immense support and encouragement in helping me write this book and in everything I do. I also want to express my thanks to my extended family including my Mum, Chantelle, Bev, Ivy, Henry, Kieran, and Sharon as well as all my friends and colleagues, past and present, who have all helped develop me and my career. Finally, I dedicate this to my late Dad who I know would be very proud of this and who I miss every day.

Mike - We wrote this book during COVID-19 times. You would think that might make it easier, being locked away, gaining weight. In fact, for everyone globally, it has been a time of turmoil and stress, so I want to thank everyone involved for sticking with it, especially my immediate family who have supported me all the way. I am going to repeat my favorite quote from Harry S. Truman - "*It is amazing what you can accomplish if you do not care who gets the credit*". To the amazing community that has provided the ideas that have gone into this book, thank you.

Noel - To Mags, Michael, and Sean, thanks for all your support and patience. We're still not getting a cat though :-) Also thanks to Noel and Mary, I told you this IT thing would be interesting.

Donal - I have to start by saying thank you to my awesome wife, Natasha Carroll. When we started writing this book, it was just the two of us and now we have a small boy Cillian and Louis the pup. Finding time to carve out to finish the book could only be done with her rallying and support. Thanks to my Mum and Dad for encouraging me to get into IT and leading me down the path I'm on. The experiences shared in this book come from all the amazing teams I've been in over the years. They've shaped my opinions and the learnings shared here, so thank you all.

Ilaria - First thanks to Noel and Tim. When I heard they started a book, I asked if I could help. I was going through a hard time, and they said, "*yes, why not? Why don't you do the illustrations*?" I was not conscious of what that meant and realized only after a few months the effort and how many new things I had to learn and practice! I have also to thank Furo, my husband, and my parents who always encouraged me in trusting I could do it, and I did it ;)

Section 1: Practices Make Perfect

In this section, we are going to introduce the book, where it came from, and how it's organized.

Chapter 1, Introduction – Start with Why focuses on the book's purpose and the target audience. *Chapter 2, Introducing DevOps and Some Tools* explains, in our words, what DevOps is and how it helps speed up the value chain of product development. We'll explore what this chain is and the bottlenecks that DevOps culture and practices address. We'll introduce a couple of important tools that we'll use throughout the book to navigate around the use of many different types of practices we're going to apply. In *Chapter 3*, *The Journey Ahead*, we will introduce how we use real-world stories and the case study we'll use throughout the book that will outline how the remaining six sections of the book are organized.

This will set us up to build a foundation and start a journey of continuous discovery, options, and continuous delivery.

1

Introduction — Start with Why

You've picked up this book and have started reading it – thank you very much!

Perhaps you read the back cover and it gave you just enough information to be inquisitive enough to open the book up and read some more. Maybe a friend or colleague told you about it and recommended it to you. Maybe you have stumbled upon it for another reason. Whatever the reason, we're very happy you've taken some time out of your day to start reading this and we hope you get some value from it and want to keep reading it.

Before going into any kind of detail regarding what this book is about and what it's going to cover, we want to start with why. This is a practice we use to create a common vision of purpose. Why have we written this book? What problems is it trying to solve and who is the intended audience?

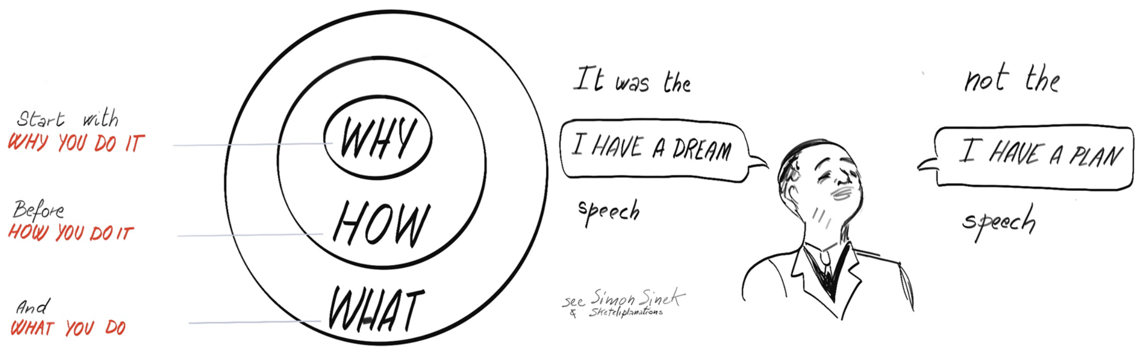


Figure 1.1: Creating a common vision of purpose

Why — For What Reason or Purpose?

While this book may have been positioned as a book about technology, it is, at most, only one-third about technology. DevOps is really all about collaboration. We wrote this book because we want to increase your understanding of DevOps, collaboration, and cultural and engineering practices on a container platform such as OpenShift. We want to make moving to DevOps easier and provide a clearer path for you to apply DevOps using OpenShift. We want to excite you when reading this and give you some inspiration as to how you can apply DevOps principles and practices. We want to equip you to go out and try these new techniques and practices.

As you progress through this book, we want you to continually measure the usefulness (impact/value) of using these new techniques. In fact, every time you try something out, we want you to think about and measure what impact it had.

That impact might be at an individual level: *What impact did trying that thing out have on me or a customer or a user? For example, has it reduced my cycle time to complete a set of delivery activities*? Alternatively, it might be an impact on a team or a department you work in: *Has team satisfaction been increased? What did we, as a group of people, achieve from that*? The impact might even be felt at organizational or societal level: *Has it reduced the number of operational incidents impacting customers*? We believe you will quickly start to see positive effects in all of these aspects. As a result, maybe you'll leave us nice reviews and tell all your friends about this book. If not, perhaps you can pivot and use this book as a doorstop or a monitor stand, which, of course, will give you a different type of value!

If you don't know where to start with how to go about measuring value, read on – we promise we'll cover that.

What we've just done is started using one of the practices and techniques used in writing this book. We have used the **Start with why** practice, which is something we always strive to do with every team or organization we work with.

So, what is a practice? A practice is an activity that helps teams achieve specific goals. It's not just an idea; it's something that you do repeatedly in order to hone or polish a skill. Practices have the following attributes:

- **Empowering**: The practices in this book will help teams discover and deliver iteratively.
- **Concise**: They can be read in a few minutes.
- **Agnostic**: Practices don't require the team to follow a specific framework.
- **Proven**: Practices have been tested in the real world.
- **Repeatable**: Practices can be used more than once.

Hopefully, throughout this book, you'll see examples of us **practicing what we preach** through the experiences, stories, and tips we will share from our real-world delivery experience, which includes stories such as these:

- The story about when we worked with an insurance company to rebuild one of their applications using DevOps and OpenShift but had a **stop the world** moment (a practice we'll talk about in the next section) when we realized we were redeveloping an app that users did not want and were not using!
- The story of when we worked with a European automotive company and kick-started modern application development and agile practices with one of their teams, only for the product owner to question how they were going to prove to management that this was a better way of working when management **only work with spreadsheets and numbers**.
- The story of the telecom company that suffered huge outages and non-functional problems over a festive period and were keen to learn new cultural and engineering practices to drive an auto-scaling and self-healing approach to their infrastructure and applications.

Why Should I Listen to These Folks?

Before you read any more from the four folks writing this book, perhaps it's worth taking a step back and sharing a bit of background as to where all our anecdotes, theories, stories, and tips come from.

We all work for Red Hat. In particular, we are all a part of Red Hat's services organization, which means that we all regularly interact with, and deliver professional services to, Red Hat customers. This ranges from helping with installation and supporting the early adoption of Red Hat technology to driving large transformation programs underpinned by Red Hat technology and Red Hat's culture.

Red Hat's culture is relatively unique as it is entirely based on open source culture and open organizations (of which Red Hat is one of the largest examples). This means that the Red Hat organization is run under a set of characteristics that are closely aligned with open source culture and philosophy. They include collaboration, community, inclusivity, adaptability, and transparency. We highly recommend learning more about Red Hat's open organization philosophy by reading *Jim Whitehurst's The Open Organization*[1].

A lot of the experience that has informed this book and the stories and tips we will share emanate from engagements led by Red Hat Open Innovation Labs (or Labs for short). Labs provides an immersive and open approach to creating new ways of working that can help our customers and their teams develop digital solutions and accelerate business value using open technology and open culture. The main offering provided by Labs is called the residency, which is a four- to twelve-week timeboxed engagement where client's engineers are matched one-on-one with Red Hat's technology and culture specialists.

Between the four authors, we've been involved in over 50 Open Innovation Labs' residencies around the world, in addition to many other professional services engagements. Due to the relatively short nature of Labs residencies, we get to learn very quickly different techniques, different approaches, and different practices. We get to see what works well and what doesn't work so well. We get to build up a huge collection of stories and tips. This book is all about sharing those stories and tips.

Where Did This Book Come From?

The title of this book is an evolution of a training enablement program that the authors have developed named *DevOps Culture and Practice Enablement*. This is an immersive training course run by Red Hat, providing enablement to Red Hat customers, partners, and employees.

We initially created the course because the services area of Red Hat we are working in was growing, and we needed a way to consistently increase the enthusiasm and shared understanding behind the practices and culture we were using globally, with our customers, and within our own organization. We wanted to do this by exploring all of the principal practices we had found to be successful in taking many products to market with our customers. This included practices to help understand the why and drive the discovery of products, as well as practices that would help us safely, securely, and confidently deliver in an iterative and incremental manner. And then there was the third outcome, which was having fun. We really couldn't see the point in all of this if you couldn't have some fun, banter, and enjoyment as you went along – it's one of the key ingredients of that mysterious word culture.

---

1 https://www.redhat.com/en/explore/the-open-organization-book

One of the key success factors behind this was injecting lots of experience and real-life stories into our delivery and using a lot of our practices on ourselves to deliver the course. Every time we run the course, we use the definition of done[2] practice to explain to participants that every practice we are going to teach on the course will be presented in a consistent way, following this process:

1. Introducing the practice with the theory and an overview of what it is, why you should use it, and how to use it
2. A hands-on practical exercise so everyone participating can leave the course having **had a go** at using the practice and having gained some learning and experience from it
3. A real-world example of the practice being used in action on a real customer delivery project or product development initiative

The core practices taught in this course vary from discovery practices, such as impact mapping and event storming, to delivery practices, such as sprint planning and retrospectives. They include a set of practices we've found to be very powerful in establishing high-performing, long-lived product teams, such as social contracts, team sentiment practices, and mob and pair programming. They include the engineering practices that many coming to the course would have most strongly associated with the term *DevOps*, such as continuous integration, continuous delivery, test-driven development, and infrastructure as code.

One of the unique aspects of this course is its appeal to a broad audience. It is not exclusively for technologists or designers. In fact, we embraced the idea of having cross-functional groups of people spanning from engineers to project managers, from infrastructure experts to user experience designers. We felt this course offered the opportunity to break down silos. We intentionally do not run different tracks for different types of people. The aim is for participants to have a shared understanding of all of the practices that can be applied to truly appreciate and enable a DevOps culture.

Having run this course more than a hundred times globally, we've learned volumes from it and have continuously improved it as we've gone along.

Faced with the opportunity to write a new book about DevOps with OpenShift and to apply new learnings and more up-to-date technologies from Stefano Picozzi, Mike Hepburn, and Noel O'Connor's existing book, *DevOps with OpenShift – Cloud Deployments Made Easy*, we considered what the important ingredients are to make DevOps with OpenShift a success for any organization choosing to adopt the technology.

2 https://openpracticelibrary.com/practice/definition-of-done/

The success factors are all based on people, processes, and technology through the application of the many practices we've used with our customers globally and, in particular, the kinds of practices we were introducing and enabling using DevOps culture and practice enablement.

This book's purpose is to enable you to understand and be ready to apply the many different practices – some people-related, some process-related, some technology-related – that will make DevOps culture and practice with OpenShift a success within your organization.

Who Exactly Is This Book For?

This book is intended for a broad audience – anyone who is in any way interested in DevOps practices and/or OpenShift or other Kubernetes platforms. One of the first activities for us to undertake was to get together and list the different personas and types of reader we intended to write for. These included the following:

Figure 1.2: The intended audience

- **Caoimhe**, a technical lead who looks after a team of people who develop software. She wants to learn more about DevOps so she can help adopt great DevOps practices.
- **Fionn**, a project manager who is responsible for a set of legacy software applications and wants to modernize his team's approach to make use of this **DevOps** thing he's heard lots of people talking about.
- **Padraig**, an Agile coach who is very experienced in applying Agile delivery frameworks such as **Scrum** and wants to further his skills and experience with DevOps. He feels that this will really add value to the teams he is coaching.
- **Tadhg**, a user experience designer who wants to better understand what other people in the company's development team do with his designs and how he can collaborate with them to deliver products.
- **Séamus**, who is an IT leader executing his company's technology strategy to adopt containers and cloud-native technology across the company's entire IT estate. He has chosen **OpenShift Container Platform** (**OCP**) as the strategic product to support this. He wants to ensure that OCP generates a fast return on investment and that there is a large uptake across all IT teams in his organization.
- **Aroha**, the CIO of the organization. She wants to ensure that the company's people are aligned with company strategy and getting the very best out of the technology and the organizational decisions being made to drive the strategy. She's motivated for the business to become more agile and adapt quickly if and when market conditions change. She wants to read about what similarly sized organizations in different industries (including in her own industry) have successfully done and what they saw as being the critical success factors.
- **Siobhán**, an infrastructure engineer who has been using Kubernetes for many years and is now part of a team introducing OCP to her organization. She wants to ensure that the platform is configured to support her team's goals and wants to know how she can best work with development teams so that they get the maximum value out of the technology.

- **Eimar**, a project manager who has spent two decades delivering IT projects through up-front planning, tracking deliverables against plans, and managing risks, issues, and dependencies with strong project reporting and stakeholder management skills. She gets frustrated by the amount of time it takes to ship software and not being able to address user needs and fixes quickly. She sees the benefit of moving to a more product-centric approach rather than a project-centric one. She would like to re-skill herself to be a product manager. In doing this, she wants to be able to test and adapt quickly, ship deliverables quicker, adapt to changing market conditions, and also improve performance, uptime, recovery times, and more.
- **Finn**, a system tester who takes great pride in quality assuring software before it is shipped to customers. His business analysis background helps him develop comprehensive testing approaches and scripts and, over the years, he's also led performance testing, security testing, and operability testing. He's keen to learn how he can introduce more automation to his work and branch out to other forms of testing.

From I to T to M

With this book, we want people to move away from being **I-shaped**, where they are a specialist in one skill or one field. We want them to become more **T-shaped**, where they still have a depth of skill and experience in a particular field (such as infrastructure or UX design), but they also have an appreciation and breadth of knowledge across all the other skills that people bring to make up a cross-functional team. This could be a frontend engineer, for example, who also works side by side with the API engineer.

A great cross-functional team is one where the full team holds all the skills and experience they need. They are empowered to take a new requirement from a user or business stakeholder through to production. A team could be made up of lots of **I-shaped** people, but this type of team quickly becomes dependent on specific individuals who can be a blocker when they are not available. For example, if a database change is needed to expose a new API but only one team member has the knowledge to be able to do this, the team can quickly become stuck. If the team is full of more **T-shaped** members, there is a greater opportunity for collaboration, sharing, and partnerships across the team and less reliance on individuals:

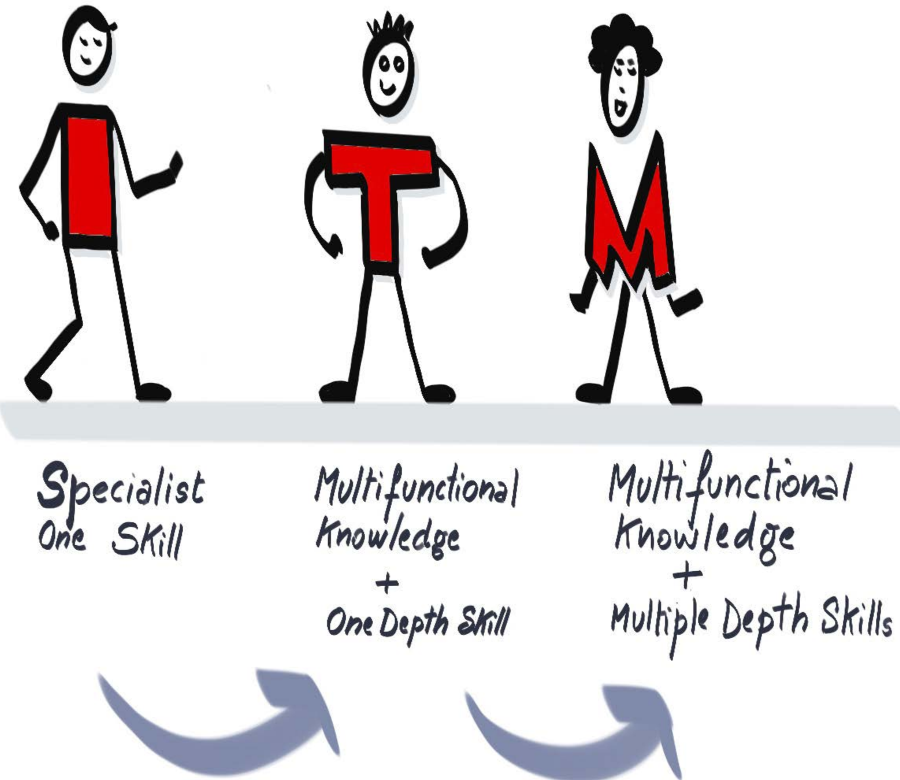


Figure 1.3: Skills transformation

We want this book to help **I-shaped** people become more **T-shaped** and perhaps even become **M-shaped**. **M-shaped** people are inspired to deepen their learning, take it into other fields, and hold multiple skills, thereby building stronger cross-functional teams.

Conclusion

This chapter presented a brief overview of why we wrote this book and who it is intended for.

We introduced ourselves and how we will be using our applied knowledge, experience, and learnings to write this book full of stories and examples.

We examined the different personas we are targeting in this book and how we intend to help move these focused I-shaped people into more T-shaped or M-shaped to build stronger cross functional teams.

In the next chapter, we will introduce DevOps and some tools we will use during the book to organize and explain DevOps practices.

2

Introducing DevOps and Some Tools

What Does It Mean to Be DevOps in a Container World? People have different perceptions about DevOps, what it means, and how it works.

In this chapter, we are going to explain our view on DevOps and the bottlenecks and challenges that DevOps focuses on addressing. We will introduce the idea of a value chain in software product delivery and how we can use different techniques from lean, agile, and DevOps communities to optimize and speed up the value chain.

We will also introduce some tools, such as the Mobius Loop and the Open Practice Library that we will use to navigate our way through the many practices utilized in the rest of the book.

DevOps is a bit of a buzzword at the moment! It seems that for every decade in technology, there is a new buzzword associated with it.

Throughout the 2010s, Agile was that buzzword—*This is going to be an Agile project*, or *We're going to use Agile to deliver this*, or *We're going to use the Agile methodology* were common phrases that many of us have heard. It was (and still is) often used incorrectly about delivering software faster. In fact, Agile is focused more around delivering business value earlier and more frequently and driving a culture of continuous learning. Agile has now officially grown up—it had its 18th birthday in February 2019. Even after all this time, we still love to use the values and principles of the Agile Manifesto[1] created back in 2001.

Containers is another buzzword these days. We see it being used by individuals without them necessarily understanding the full meaning of what a container is and why people, teams, and organizations would benefit by utilizing them.

So, with this book being about DevOps and OpenShift (a container management platform), we're going to de-buzzify these terms and talk about very practical, real world experience and examples of the real value behind DevOps and OpenShift containers.

Let's take a look back in time and see where we believe these phenomena came from.

We all have worked in IT for a number of decades (some more decades than others!). While chatting over a beer and looking back at our experiences of delivering IT projects, we recognized some common characteristics in all our IT projects that have been constant. We also identified a set of gaps in the value chain of delivering IT projects that, for us, seemed to slow things down.

The Value Chain

Every project we've ever worked on has had some kind of end customer or user. Sometimes they have been external users, such as an online shopper wanting to use their mobile app to buy the latest set of Justin Bieber bedsheets! Other times, they have been teams internal to an organization, such as an operations team or a particular department within a company. One common denominator we all agree on is that the objective of our work was always having smiley, happy customers:

1 www.agilemanifesto.org

Figure 2.1: Happy customers — The ultimate goal of organizations

Between us, we have helped many organizations, from the public sector and finance to retail and charities. We've seen it all! As we reminisced, we discussed the end result of some of our projects; we thought about our why – there was almost always some kind of monetary value aspect associated with the reason for us being there. There were other motivations, too, such as increased customer satisfaction, reduced risk, and improved security and performance, but the bottom line is that an essential part of any of our commercial customers' business is to make money and reduce costs.

So, in the end, value was often linked to money in some shape or form. Three of us authors are Irish and the fourth is from New Zealand, so we felt it was appropriate to reflect this as a pot of gold!

Figure 2.2: Profits — A common goal of every commercial organization

The 1990 book *The Machine That Changed the World*, written by *James Womack, Daniel Jones, and Daniel Roos*, first introduced the term **value stream**. The idea was further popularized by the book *Lean Thinking*, written by the same authors. According to them, the value stream is the sequence of activities an organization undertakes to deliver on a customer request. More broadly, a value stream is the sequence of activities required to design, produce, and deliver a good or service to a customer, and it includes the dual flows of information and material. Most value streams are highly cross-functional: the transformation of a customer request to a good or service flows through many functional departments or work teams within the organization:

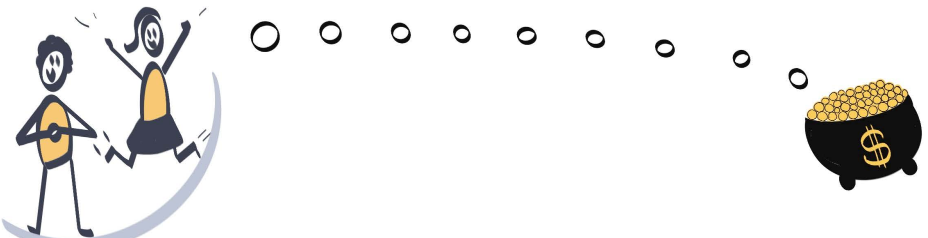


Figure 2.3: Customers dreaming of the pot of gold

Let's visualize this as our customers dreaming of that pot of gold. They're constantly thinking about how they can get the most out of their products or ideas to generate the most gold. So, how do they go about doing this?

The Gaps

We're going to explore the gaps in the value chain between customers and organization's business people, between business people and development people, and between development people and operations people.

The Big List of Things to Do

The first gap in the software development process that we consistently saw was the process of collecting information from end customers and forming a list of customer requirements:

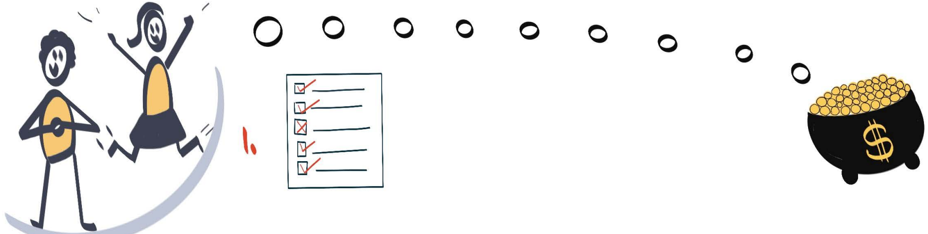


Figure 2.4: Understanding and collecting customer requirements

Our early projects often involved long phases of business analysts documenting every possible requirement they could conceivably think of into epic volumes of business requirements – documents. The goal was to pre-empt every conceivable customer journey or scenario and to cover all the bases by building specifications that included every possible eventuality. Sounds rigid, right? What if we made an incorrect assumption?

Demonstrating Value and Building the Right Thing

The second gap revolved around demonstrating value to customers. Usually, the **project** being undertaken was set up to include all of the features and ideas needed so that they could be released together. Once the project was in production, it would only have a small operations budget to support minor enhancements and problem resolution. Sounds like it might take a long time to get the application into the end users' hands, right?

There are two reasons we call these *gaps*. First, the process was lengthy – months, sometimes years, would elapse between starting a project and signing off on the requirements. Second, trying to collect every possible requirement before delivering anything would mean no real benefit to the end customer for years and often, the wrong functionality was built and delivered to an unhappy customer:

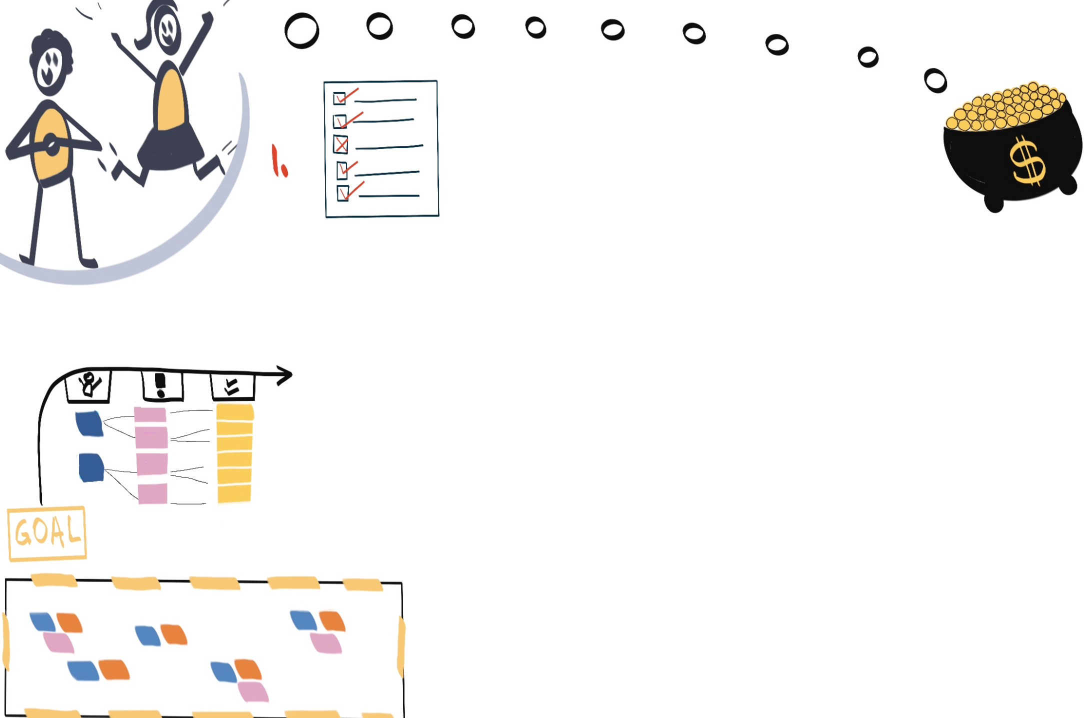


Figure 2.5: Using human-centered practices to understand customer needs

This gap of *not building the right thing* has been plugged in recent years by the emergence of human-centered design and design thinking. These are a set of practices that put the end user at the center of capturing the needs and requirements of a product.

We gather the information by talking directly to users and forming greater *empathy*[2] with them:

empathy noun

em·pa·thy | \ ˈem-pə-thē \

Definition of *empathy*

1 : the action of understanding, being aware of, being sensitive to, and vicariously experiencing the feelings, thoughts, and experience of another of either the past or present without having the feelings, thoughts, and experience fully communicated in an objectively explicit manner

also : the capacity for this

Figure 2.6: Merriam-Webster definition of 'empathy'

In this book, we'll explore how techniques such as impact mapping, event storming, and human-centered design can aid the software development process. We'll also explore other practices to help us define solutions and features and crucially ensure that the solution is connected to business value. We'll show how the act of coupling research activities such as user interface prototypes and technical spikes with experimentation inform product backlogs that are well prioritized according to delivered business value. We will show you how using just enough information can lead to a better-understood product.

How Do We Do the Things on Our List?

Let's consider the second gap in delivering value to users. This gap focuses on moving from a shopping list of TODO items into working software.

The traditional approach is to sign off and confirm a finite set of requirements that have undergone the lengthy process of business analysis and capture. The scope of the project is locked down and a stringent change control process and governance is put in place for dealing with any deviation from the documented requirements.

2 https://www.merriam-webster.com/dictionary/empath

A team of software designers and architects then gets to work, producing a **high-level design** (**HLD**) that will deliver a solution or set of solutions according to the business requirements specified. These requirements also go through a formal review process by key project stakeholders and, once signed off, become the reference source for the solution scope.

Often, different design documents are written in the next phase - detail design documents, program specifications, data designs, logical architecture blueprints, physical architecture solutions, and many more. Each of these is written to support a defined, dated, and signed-off version of the HLD, which itself, is signed off against a defined set of business requirement specifications:

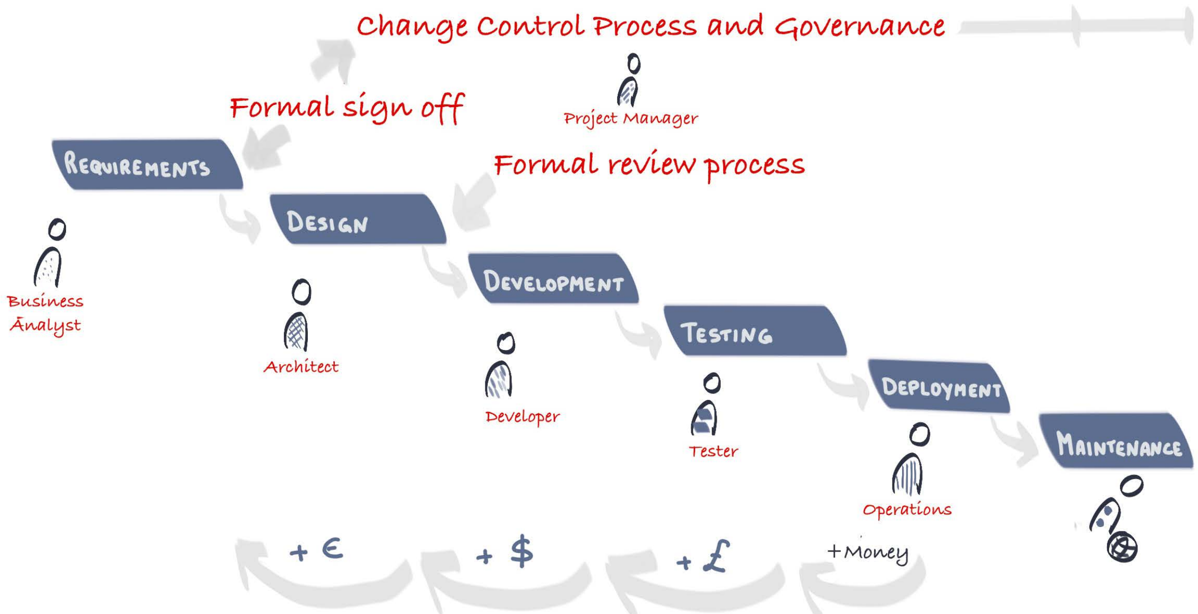


Figure 2.7: Traditional application development lifecycle

Any changes to the earlier documents have direct time and cost implications for reassessing and updating each of the following design documents. Software development teams may have been involved in the production or review of some of these documents. However, they are often encouraged not to start any coding or development activities until these designs have been locked down. Some organizations reduce project costs by not onboarding development teams until this stage. Development is often siloed by function and unaware of the big picture with limited automated testing.

At a predefined point in the project plan, all developers are expected to have delivered their coded components to a testing environment. Perhaps each developer manually builds and deploys their own code to the testing environment. Some larger programs seek economies of scale by setting up build infrastructure teams who do this on behalf of all developers. Once all components had been delivered, a separate team of testers starts executing the hundreds of test scripts they had been writing in the preceding weeks and months to test the solution according to business requirements and HLD documentation. This is the first time some components are integrated and tested together. Of course, problems and bugs drive reworking by development teams and designers to fix such issues.

Just as there are different levels of design documentation, testing often undergoes different levels of testing, with one starting when the previous phase is completed. A test manager would sign off on a set of test results, signaling that the next level of testing could start. Testing would range from a set of component integration testing to wider system integration testing, security and penetration testing, performance testing, failover and operability testing, and finally, user acceptance testing!

The final stage before the big-bang go-live of a solution would often be user acceptance testing, involving a set of focus users and the test system. In many cases, it could often be months or years before this first user saw the implemented system. Once user acceptance of the solution was signed off, the green light was given to deploy to the production environment. Finally, with the software in the hands of real end users, business revenue could hopefully be generated from all this work.

You're probably thinking that this process sounds long and drawn out – well in truth, it was! Many programs hit delays at different points along the way and what started out as a multi-month project plan ended up being years long. For the curious, there is even a list of some epic failures on Wikipedia: https://en.wikipedia.org/wiki/List_of_failed_and_overbudget_custom_software_projects.

Often, business conditions would change during the development period. New feature requests would be generated. During testing, gaps in the requirements would emerge that no one considered during the analysis and requirements capture. The market didn't stand still during development and competitor companies may have started to innovate quicker. The competition would even provide more feature requests, in a process akin to a feature comparison war.

Of course, there was always some kind of change control procedure to handle new scope like this. In a complex program of work, the lead time to get features added to the work plan could range from months to years. In order to get something into production, program executives would simply say no to any more change and just focus on getting to the end of the project plan.

This meant that solutions finally delivered to production were somewhat underwhelming to users several years after the first requirements were discussed. Time and industry had moved on. The biggest frustration of these programs was that they were frequently delivered late, were over budget, and often delivered a solution that lacked user satisfaction or quality.

Stepping back a little, we had this huge gap of converting lists of features into a software deliverable. The process known as **Waterfall** due to the nature of separate phases of work flowing down to the next phase was associated with very lengthy times:

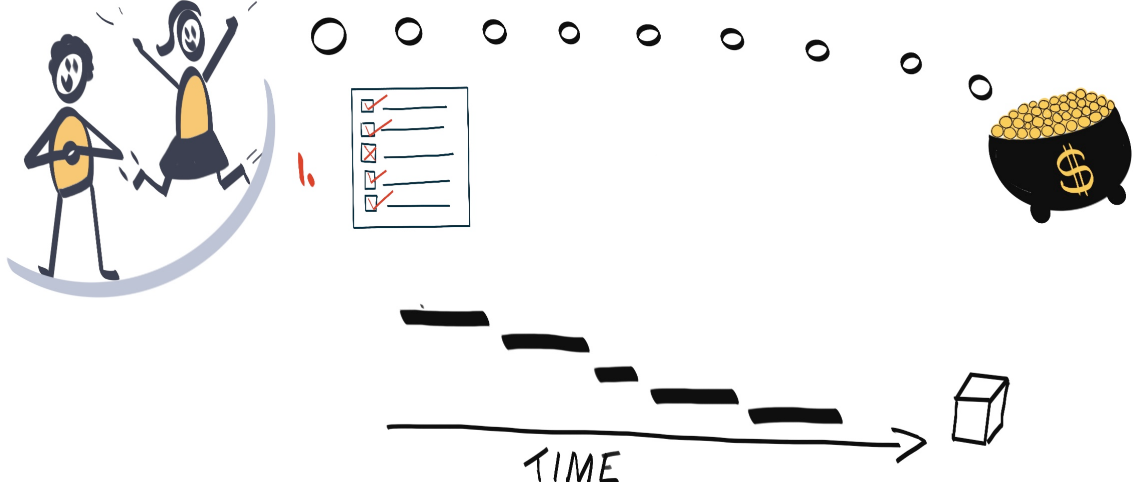


Figure 2.8: Traditional deliverables with its drawbacks failed to achieve customer satisfaction

Let's think about how we plug that second gap with more modern software development processes. How do modern developers manage to translate user needs into working software solutions much more quickly compared to previous ways of working?

The formation of the Agile movement in 2001, led by the 17 IT individuals who wrote the Agile Manifesto, has triggered alternative approaches and mindsets toward delivering software. Many of the individuals involved in writing the Agile Manifesto had been tackling many of the problems described by Waterfall development. Jeff Sutherland and Ken Shwaber had created the Scrum framework for software development, which included delivering small incremental releases of value much more frequently – they used the term **sprint**, which was a fixed timebox ranging from one to four weeks (usually being two weeks), during which a set of events and roles would work together such that big solutions could be delivered iteratively and incrementally. Kent Beck and Ron Jefferies led much of the **eXtreme Programming** (**XP**) movement, focusing on delivering faster releases of value and working on key practices that helped drive more efficiency into review, testing, and release processes, using better collaboration and increased automation:

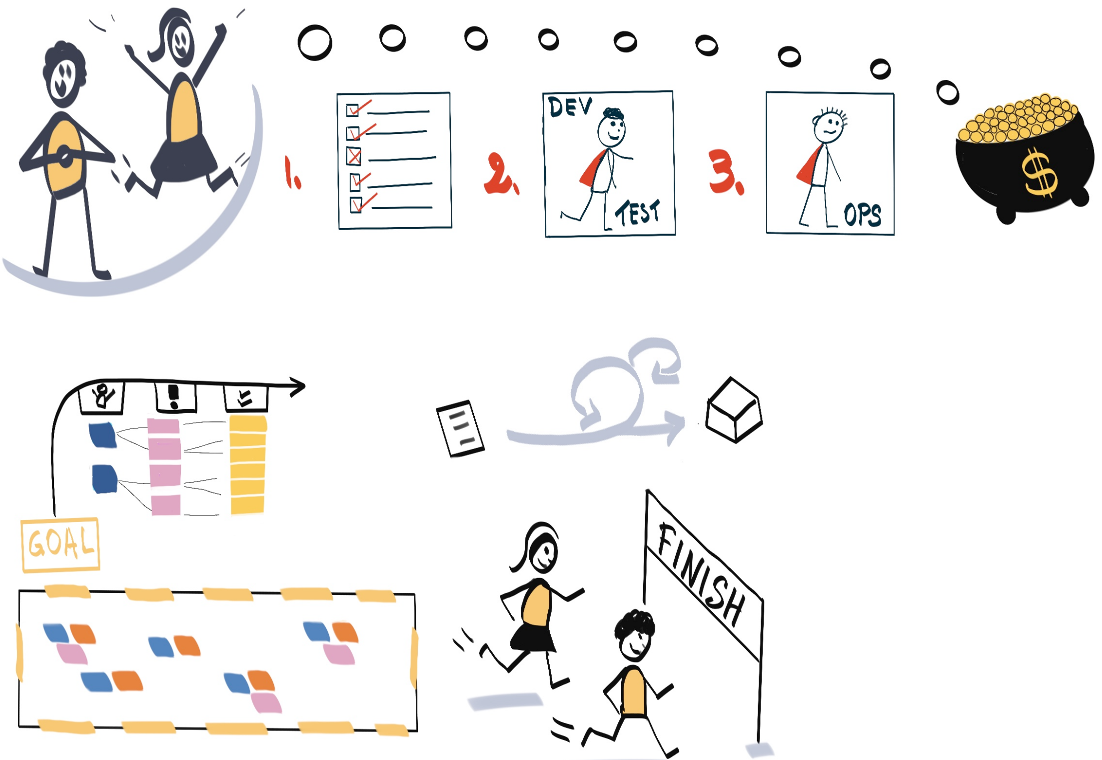


Figure 2.9: Implementation of DevOps practices leading to faster delivery and better products

In this book, we'll show you different software delivery practices and how our experience using a mixture of different practices from Scrum, Kanban, XP, Lean, and some scaling frameworks helps deliver value quicker. All the underlying practices are simply tools to help close the gap between an idea or requirement being captured and it being delivered. This has been an area we have sought to continuously improve to a level where the gaps are minimized and we're working in a mode of continuous delivery.

Development to Operations

There is one more gap to plug in our efforts to optimize the software delivery process. The third gap is the one between development teams and operations teams.

In our Waterfall process, we had reached the point where the signed-off solution exited user acceptance testing and went through a big-bang go-live. So, what happened next?

Often, a whole new team responsible for maintenance and support would then pick up the solution. The people who work in this new team were not involved in any of the design, development, or testing, so additional time would be built into the project plan for knowledge transfer. The delivery team would write lengthy documentation in the hope that this would be a useful resource for future operations teams.

At this point, the package of software would metaphorically be thrown over the wall from the army of developers to the group of operation engineers. The operations teams often had to learn about the software the hard way by investigating production incidents, addressing bugs that were not found previously, and handling new scenarios not considered during the requirement planning stage:

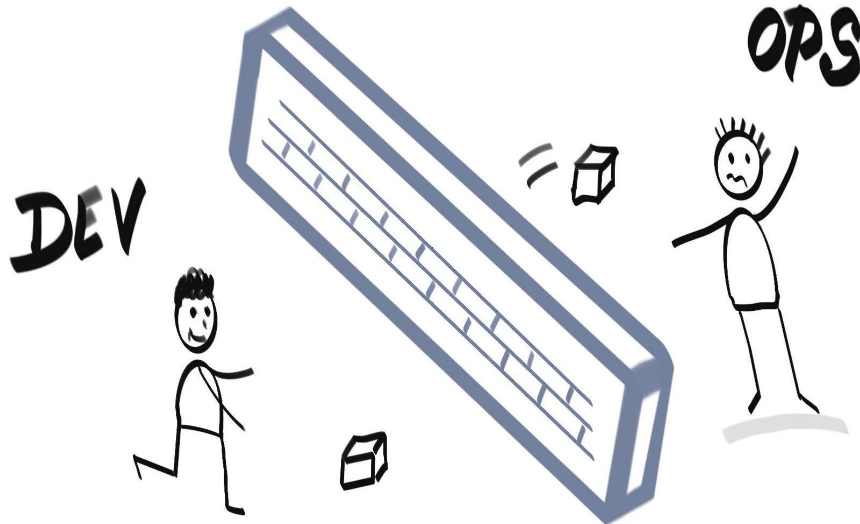


Figure 2.10: Aspiring to bring down the wall between development and operations teams

To plug this gap, we must bring development and operations teams together. Tear down that wall and remove the silos! Bringing down the wall forms new teams that are focused on development and operations activities. These teams are collectively responsible for the whole solution and can design the solution according to each others' needs.

The term DevOps was coined by the idea that we no longer have siloed development and operations teams. In recent years, we've seen various other terms emerge from this idea, such as DevSecOps, BizDevOps, DesOps and even BizDesDevSecOps!

> **Note**
>
> **BizDesDevSecOps** is a bit of a mouthful, so we're going to use the term **product team** to describe it throughout this book. It addresses the ultimate goal of plugging all gaps in the software development process and bringing down all the walls.

Figure 2.11: Plugging the gaps in the software delivery process

Note that we will not use the **DevOps team** term **DevOps team** – the idea of having a team or even an individual purely focused on DevOps runs counter to what the DevOps philosophy is all about – collaboration, cross-functionality, and the removal of silos. How many times have you seen ads on LinkedIn or other sites looking for DevOps engineers? The invention of the DevOps engineer or the DevOps team could be seen as creating just another silo.

People, Process, and Technology

DevOps is really all about collaboration. It's about taking pride in, and ownership of, the solution you're building by bringing down walls and silos and by removing bottlenecks and obstacles. This speeds up the value stream connecting the customer's perceived need to the product delivery.

Technology alone will never solve all your business problems. No matter how good the platform or software product you are evaluating or being sold, unless your organization has learned to adopt the correct balance of people aspects, process changes, and technology adoption, the objectives will not be met.

This book is about finding the right combination of people, process, and technology changes needed to maximize business outcomes on a continuous basis. This requires changes in mindset and changes in behavior. This book will look at the behavioral change that we have seen be most effective with the hundreds of organizations we have collectively worked with. We've observed that such mindset and behavioral change is needed across all roles and that we need to break down the silos we see inside organizations, which, as we saw previously, is what drives the gaps and inefficiencies in software development:

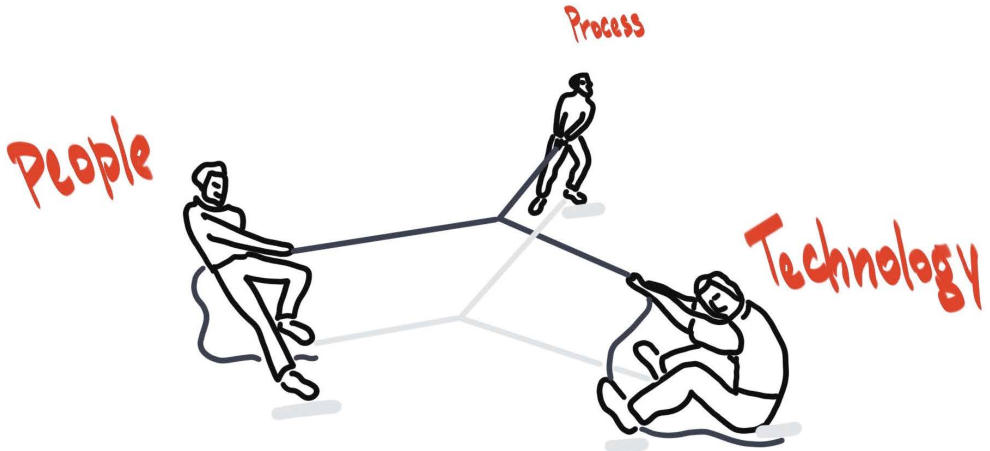


Figure 2.12: A healthy balance between people, process, and technology

Everyone in an organization should care about people, process engineering, and technology in order to drive the desired outcomes. We want to break down the silos between these three pillars and bring them closer together. A reader who may be more interested and focused in one of these three things will get as much (if not more) value from learning about the other two things.

This means a hardcore software engineer or architect can pick up some great insights and guidance on why people, culture, and collaboration are equally important for their role.

Someone who has previously been an expert in project management methodologies and is now learning about more agile delivery practices such as Scrum can also use this book to learn about modern technology approaches such as GitOps, CI/CD, and serverless. They can learn why these are important to understand and appreciate so that they can articulate the business value such approaches bring to organizations.

A leader who is concerned about employee retention can learn how the mastery of these modern tech practices of iterative and incremental delivery strategies can maximize the opportunities for organizational success through the delivery of highly valuable products being used by happy customers.

The Mobius Loop and the Open Practice Library

In this book, we're going to explore lots of different practices. We're going to explain what they are and why we use them. We're going to give you some guidance on how to use them. We're going to share some real-world examples of how we've used them and, where possible, we'll even show them in action. Using our Pet Battle case study (more on that later), we're going to bring them to life in a fun way and we'll share the best tips that we've picked up in the field.

A problem we hit a few years ago when working with our customers' new teams was explaining how and when you might want to use different practices and in what order. What practice should we start with? What practice links nicely to the output produced from a previous practice, and so on?

To help with this, we have made use of an open-source navigator tool called Mobius. This was created by Gabrielle Benefield and Ryan Shriver. There is a huge amount of great material, including a number of open-sourced canvases and artifacts, available at www.mobiusloop.com. Red Hat Open Innovation Labs makes use of this open-source material in all of its residencies and in its DevOps culture and practice enablement courses.[3] We will use it in this book to structure the content and the sections.

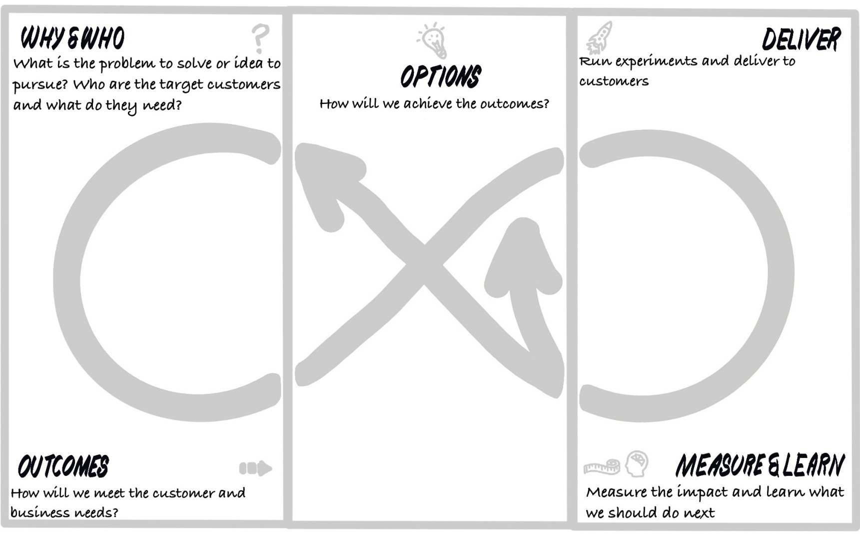


Figure 2.13: The Mobius loop[4]

3 https://github.com/rht-labs/enablement-docs

4 The Mobius Loop (https://mobiusloop.com/) resources by Gabrielle Benefield and Ryan Shriver used here and throughout this book are licensed under CC BY 3.0. Later in the book, images include an additional modification of a foundation layer. For more information please see https://creativecommons.org/licenses/by/3.0/

Mobius is a framework that connects discovery and delivery and can be used to connect strategy to products to operations. The common denominator is measurable outcomes. Mobius is used to understand, align, and share measurable target outcomes so they can be tested and validated.

There are a number of principles that underpin the Mobius navigator:

- **Outcomes over outputs**: We focus on delivering tangible impacts or outcomes to people as opposed to delivering lots of features that may not drive outcomes.
- **Multi-options strategy** (**options pivot**): We look to build a list of options, a list of research initiatives, experiments, and implementation features that can be used to test hypotheses about whether those research initiatives, experiments, and implementation features will indeed drive the anticipated outcomes.
- **Rapid delivery**: We aim to use short iterations of delivery with regular feedback and measurement as we strive toward the idea of continuous delivery.
- **Continuous learning and improvement**: happens throughout the cycle so that our next set of options yield an even better impact on outcomes.

There are **seven core elements** to the Mobius approach across a continuous and never-ending flow. They can be visualized on a single canvas that is open source and made available under a creative commons license at www.mobiusloop.com:

- **Why** describes the purpose. Why are we doing this? What is the problem we are trying to solve? What is the idea we are trying to pursue?
- **Who** focuses on the end users. Who are we trying to solve the problem for?
- **Outcomes** are where we want to get to with these people, the changes in their human behavior that influences big results, and how we will measure the customer and business impacts delivered.
- **Options** are the potential solutions that could deliver these outcomes. They help define the hypotheses we can go on to test and help us find the simplest way to achieve the desired outcome with the least amount of effort or output.
- **Deliver** is the cycle where we run experiments to deliver a solution or set of solutions to users so we can measure the impact.
- **Measure** is where we assess what happened as a result of delivering the solution or set of solutions. We check whether the impact of the solution delivered the desired outcomes and assess how much of an impact we achieved.
- **Learn** is the feedback loop that takes us back to the options pivot. We learn from what we delivered and assess what to do next. Have we delivered enough to make an assessment? Do we go right back around the delivery loop again? Have we reached our target outcomes or invalidated assumptions from our learnings? Do we return to the discovery loop?

Personas such as Tadhg, our user experience designer, would typically spend a lot of time in the discovery loop. Personas such as Caoimhe, our technical lead, would traditionally be focused on the delivery loop. Personas such as Fionn, our project manager, would typically spend a lot of time here establishing outcomes and gathering options. But, as we seek to move to cross-functional teams of T- or M-shaped people, we really benefit from everyone being involved at every stage of the Mobius loop. And Mobius creates a common language based on targeted measurable outcomes.

You can apply the same principles of outcome-driven thinking for strategy, product, and services delivery to enabling business and technical operations – we'll return to this idea later in the book.

Mobius is powerful because it's framework agnostic. It integrates with many existing frameworks and methods you may already be familiar with – Scrum, Kanban, design thinking, Lean UX, Business Model Generation, Lean startup, and many other great frameworks that have surfaced during the last couple of decades. You don't have to reinvent the wheel or replace everything you already like and that works for you.

You can capture key information on a discovery map, an options map, and a delivery map – all of these are open source artifacts available under **Creative Commons** at www.mobiusloop.com:

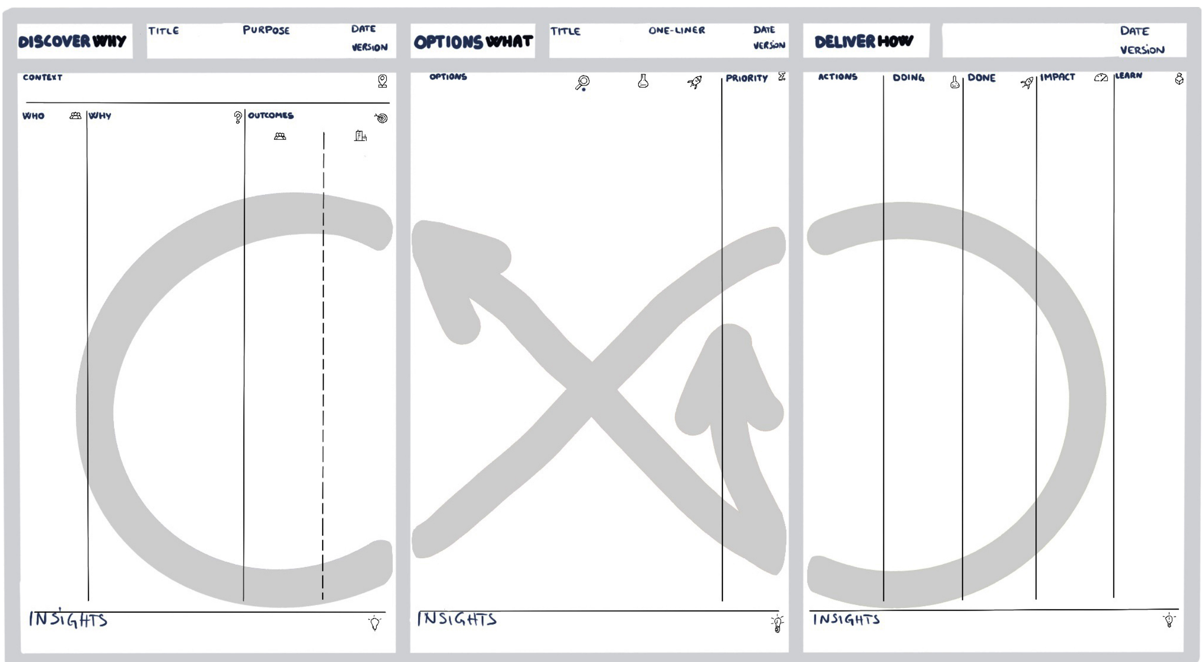


Figure 2.14: Using the Discovery, Options, and Delivery canvases of the Mobius loop

When Red Hat Open Innovation Labs started using Mobius, we placed all of our practices around the Mobius loop. Some practices clearly aligned with the discovery loop and, in particular, the **Why & Who** end of the discovery loop. Practices such as impact mapping, start-at-the-end, and empathy mapping are great at uncovering the answers posed in this section of the loop. We'll get into the detail of these practices in subsequent chapters of this book.

Practices such as event storming and user story mapping were very helpful in establishing and visualizing outcomes on the other side of the discovery loop. Again, we'll look at these practices in detail and share some great examples of their effect.

Practices such as design sprints, how-might-we, and product backlog refinement would help determine and organize the series of options available attempting to drive toward outcomes.

Practices such as sprint planning would help plan and execute the incremental delivery of products toward outcomes. We'll explore these iterative delivery practices and how different Agile frameworks can be used with Mobius.

Practices such as showcases and retrospectives would help with capturing measure-and-learn data from incremental delivery.

We still had a large number of practices that we did not feel naturally fitted into one of the loops or the options pivot. When we laid out all of the remaining practices that we had all used with numerous customers very effectively, we found they fitted into one of two areas. One set of practices were all focused on creating culture and collaboration. The other practices were all technical engineering practices that supported the concept of continuous delivery.

When explaining these practices to others, we talked about these being very important practices to put in place, but not necessarily practices that you would schedule. For example, you will learn that practices such as impact mapping on the discovery loop are important scheduled workshops that you execute and occasionally revisit in the future. Practices such as sprint planning, showcases, and retrospectives on the delivery loop are also tightly scheduled when working in an iterative delivery framework. But the practices associated with culture and collaboration or those associated with technical engineering were more like practices that you use all the time, continuously.

Practices such as social contracts and definition of done are not one-time-only practices where you bring the artifact out on a schedule. These are living and breathing artifacts that teams use all the time in their day-to-day work. Likewise, continuous integration, test automation, and infrastructure as code – these are not the types of practices you schedule one or two times a week. These are practices that you do all the time. They are practices in the foundation of where and how we're working. In order to effectively practice continuous delivery and continuous discovery as presented by the Mobius loop, we need to have a strong foundation of culture, collaboration, and technical engineering practices.

To visualize this, we added the foundation to the Mobius loop:

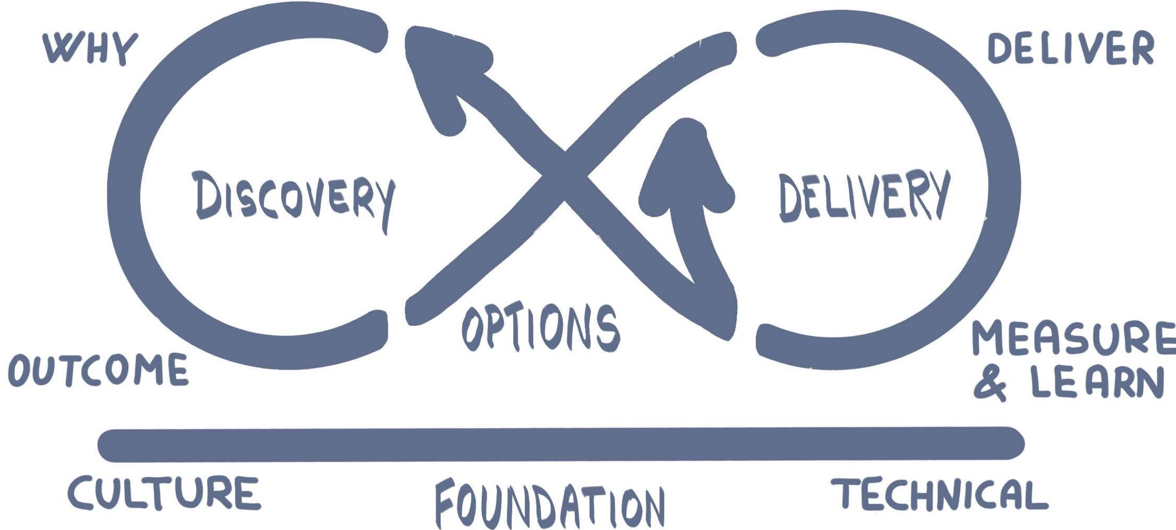


Figure 2.15: Adding a foundation to the Mobius loop

This graphic has become a simple visualization tool that helps us navigate the ever-growing list of practices and techniques we use to achieve continuous discovery and continuous delivery of digital products:

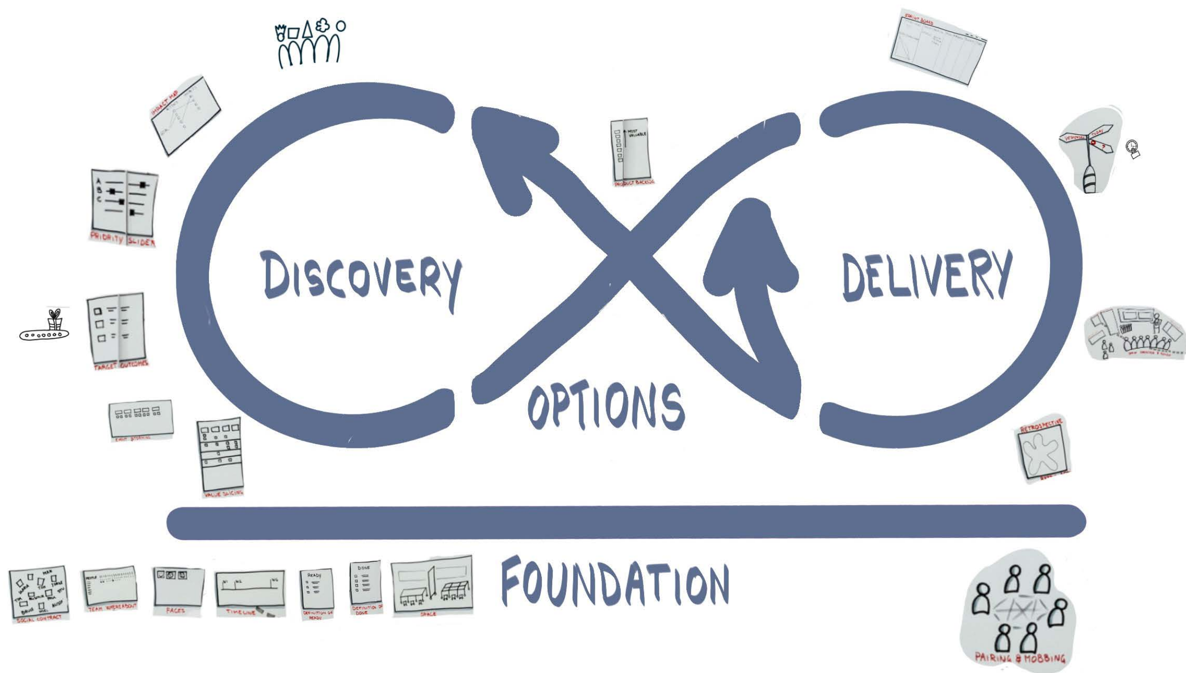


Figure 2.16: Practicing continuous discovery and delivery through the Mobius loop

Open Innovation Labs Residencies involves traveling around the Mobius loop a few times, usually starting from discovery before proceeding to delivery and then pivoting a few times to either more delivery or returning to discovery continuously. We find that, in order for this to be sustainable, you must build a foundation of culture and collaboration and you must build a strong foundation of technical engineering practices.

Open Innovation Labs kick-started an open source, community-driven project called the Open Practice Library. The Open Practice Library is a community-driven repository of practices and tools. These are shared by people currently using them day-to-day for people looking to be inspired with new ideas and experience.

All of the practices you read about in this book have been contributed to the Open Practice Library and, throughout the book, we will use the Mobius loop and the foundation of culture, collaboration, and technical practices as a reference point to determine where and how all our open practices fit together to deliver great DevOps culture and practice with OpenShift.

An important characteristic of Mobius and the Open Practice Library is that it is not prescriptive. It is not a methodology. It does not tell you exactly which practice to use when and where. Think of the Open Practice Library as a box of tools – a really well-organized toolbox with lots of compartments and shelves. The practices have been organized into compartments that help with discovery and, in particular, the *why* and *who*, followed by deriving outcomes. There is a drawer containing all the tools that help form, organize, and prioritize options and how to pivot later in the cycle. There is a portion of the toolbox with all of the tools that help with delivery – whether that be iterative and incremental delivery associated with agile practices or single delivery associated with Waterfall. There are tools to help capture and understand the measurements and learning from delivery. Finally, there is a huge drawer of tools used to establish culture, collaboration, and technical engineering excellence. These are often the first tools we go to grab when starting a piece of work.

Conclusion

In this chapter, we introduced the value chain in software product delivery and explored how traditional ways of working brought inefficiencies, bottlenecks and gaps between users, business stakeholders, development teams, and operational teams.

We explored some of the techniques that have been used to plug these gaps and how a balanced focus on people, process, and technology is needed by all involved.

Finally, we introduced the open-source navigator tool called Mobius that connects discovery and delivery in an infinite loop and can connect strategy to product to operations with a common denominator of measurable outcomes. The Open Practice Library uses mobius on a foundation of culture and technology to navigate between an evolving number of open practices – many of which will be explained in subsequent chapters.

In the next chapter, we're going to outline how we'll approach the rest of the book by introducing our case study and the structure for the remaining sections.

3

The Journey Ahead

As we conclude the first section of this book, this chapter will explain the journey we intend to take you through the remaining sections.

This will include how we intend to not just tell you about practices and techniques but also show them in action and apply them. We'll introduce a fun case study and real world stories to do this.

One of the challenges of writing a book intended to be read by a diverse group of people with different skill sets and backgrounds is how to write it in such a way that means it can be consumed, understood, and appreciated by all. From tech leads, infrastructure engineers, and OpenShift specialists, to Agile coaches, user experience designers, and project managers, to IT leaders and CXOs, we want you to grasp a shared understanding of what's behind all the practices being taught and the principles that underpin then.

The topics covered are going to range from how to capture behaviors in an empathy map using human-centered design practices to considering observability within applications using performance metrics. It will look at ways to help product owners prioritize value versus risk while also addressing instrumentation for applications, image tagging, and metadata!

Similar to the definition of done practice we use on our DevOps culture and practice enablement course, we're going to use a few different approaches in this book to help you with your journey:

1. Explaining the culture and practice
2. Showing the culture and practice
3. Applying the culture and practice

To explain the culture and practice, we will introduce what the practice is and why and where we've chosen to use it, and give some guidance on how to use it. In some ways, this is the easy part.

We have a saying among us that we prefer to **show, not tell**. It's easy to research and write a load of words. It's far more compelling to visually show a practice in action and the effect it is having. To show the culture and practice, we have a few techniques:

1. As much as possible, we'll aim to make use of visualization techniques such as sketch notes, diagrams, and other charts. You will have seen a few of these, beautifully drawn by Ilaria Doria, in this section already, and hopefully, they have helped bring the words to life a little.
2. Where we can show you a practice in action through photographs or reproduced artifacts, we will do so. Where possible, we have made the diagrams and other visual artifacts open source, and they are available at https://github.com/PacktPublishing/DevOps-Culture-and-Practice-with-OpenShift/.
3. We find stories and real-world examples the best way to explain a practice and the value it brings. So, from time to time, we will break away and tell a story that one or more of the authors have experienced connected with these practices. We'll visually tell these stories on their own by having a box around the story. Let's start with one now:

A Story about Telling a Practice

In December 2005, I was working in the billing workstream of a large telecommunications billing system replacement program in the UK. I'd been working on this program for 18 months already at this point. It was a 10-year program to replace all legacy billing systems with a more modern COTS software and introduce some new business capability enabling flexible and changeable product management.

I lead the billing interfaces workstream and was responsible for the delivery of interfaces between billing systems and third parties such as banks, BACS, and the treasury.

Our workstream was having our Christmas dinner in a pub near the office. We'd chosen this pub because most of us had been to the same pub 12 months previously for last year's Christmas dinner. It was funny to see so many of us in the same place around the same time of year 12 months on.

When I was there, I reflected on the last 12 months and what had been achieved by our expensive team of consultants during the period. It dawned on me that 12 months ago, we were entering the design phase of a major release of the program. We were running a series of workshops over several weeks to map out the different work products and deliverables required for the release.

12 months on, we were still in that design phase. A workstream of over 60 people had spent a year writing, rewriting, refining, and writing again design documents, more design documents, variations of design documents, technical clarification notes against design documents, and even change requests against design documents. At this point, no code had been written, no tests had been run, and no software had been released. All we had produced from 12 months was lots of design documents and lots of meetings.

I remember feeling somewhat underwhelmed by the impact of what we'd achieved in the last year. I said to myself then, *There has to be a better way of delivering software.*

Finally, we want to apply some of the culture and practices for real. To help us do that, we are going to use a simple, fun case study about a small start up organization going through some of the challenges and hurdles associated with creating a DevOps culture and establishing DevOps practices. This story will represent an anonymized account of some of the things we've seen in the field with our customers using these practices.

We'll regularly return to this story of applying DevOps culture and practices using OpenShift in shaded boxes. Let's get this rolling with the backstory – we hope you're ready for this!

PetBattle – the Backstory

Pictures of domestic cats are some of the most widely viewed content on the internet[1]. Is this true? Who knows! Maybe it's true. What we do know is that they make a great backstory for the example application we use in this book to help explain a number of DevOps practices:

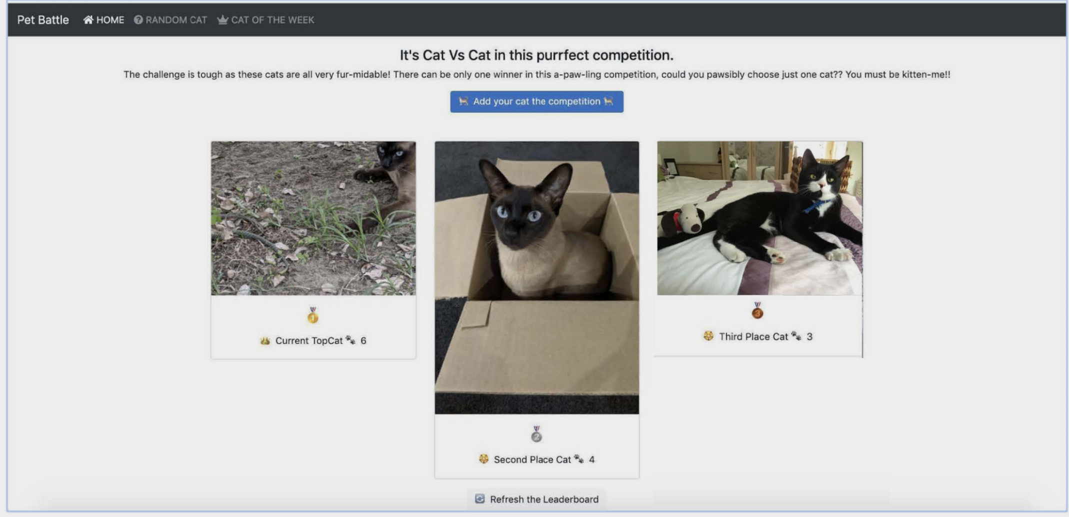


Figure 3.1: PetBattle — The backstory

PetBattle is a hobbyist app, started for fun, hacked around with so that the authors can **Cat versus Cat** battle each other in a simple online forum. A **My cat is better than your cat** *type of thing*. There are very few bells and whistles to the initial architecture – there is a simple web-based user interface and an API layer coupled with a NoSQL database.

1 https://en.wikipedia.org/wiki/Cats_and_the_Internet

PetBattle begins life deployed on a single virtual machine. It's online but not attracting a lot of visitors. It's mainly frequented by the authors' friends and family.

While on holiday in an exotic paradise, one of the authors happensed to meet an online influencer. They date, they have a holiday romance, and PetBattle suddenly becomes Insta-famous! Nearly overnight, there was a drastically increased number of players, the PetBattle server crashes, and malicious pictures of **not** cats start appearing on the child-friendly application.

Back from holiday, the authors suddenly find themselves needing to earn a living from PetBattle and decide that developing a business and a production-ready version of the hobbyist app is now a viable thing to do.

The scene is set for the PetBattle founders to go on an exciting journey embracing DevOps culture and practice with OpenShift.

What about Legacy Systems?

People often associate Agile and DevOps with greenfield, brand-new development and see it as only applicable to start-ups and those with the luxury to start again. *What about legacy systems?* is a common question we get asked.

We'll show throughout this book that the Mobius loop and the foundation can apply to any kind of project and any kind of technology; greenfield or brownfield, small web app or large mainframe, on-premises infrastructure delivery or hybrid cloud technology.

We tend to start our journey on the Mobius loop at the *discovery* part (after building the base foundation of culture, collaboration, and technical practices). But you don't have to start there. In fact, you can start anywhere on the loop. The most important tip is to make sure you regularly travel around all parts of the loop. Do not get stuck in delivery loops and never return to discovery to revisit hypotheses and assumptions previously made. Do not get stuck in discovery where you're moving so slowly, you're getting stuck in **analysis paralysis** and risk missing market windows and never delivering value. Most importantly, never forget to keep building on the foundation of culture, collaboration, and technical practices.

Borrowing Brilliance

Before we start to dive deeper into the detail, we should take a moment to point out that we did not write or dream up any of the practices in this book. The practices in this book and in the Open Practice Library are a growing list of contributions of some brilliant minds. We have borrowed that brilliance and will attribute it to the brilliant minds that came up with it. We hope we have attributed everyone correctly and any omissions are purely accidental.

What we have attempted to do with this book is show how these practices, when connected together, have delivered some highly impactful outcomes for organizations and show some of the underlying principles needed to enable those outcomes.

What to Expect from the Rest of This Book?

So, you've almost made it to the end of this first section. Thanks for sticking with us so far! We hope you're feeling suitably enthused and motivated to read some more and have enough trust in us to want to read what we've written.

If you need just a little bit more information about what to expect, here's a short overview.

Section 2 — Establishing the Foundation

In this section, we'll look much deeper into the importance of establishing a foundation both culturally and technically. We'll look again at the purpose motive – the **start with why** that we kicked off this book with and how that should be the starting point for any product or any team. We'll look at some of our favorite and most powerful practices we've used to help create the foundation and culture of collaboration – social contracts, stop-the-world andon cords, real-time retrospectives, creating team identity, and getting into the habit of visualizing everything and having a cycle of inspection and adaptation. A relentless focus on creating an environment of psychological safety is a key success factor when establishing the foundation. We'll explain what this is and how we can help achieve it.

We'll explore how executive sponsorship can enable and impede a successful foundation and explore deeper what it means to be open in terms of technology and culture. We'll look into how Agile decision-making works and some of the useful tools and practices that can help with this. And we'll look at the adoption approach and how to convince the doubters and skeptics!

From a technical foundation perspective, we're going to share some of our most successful approaches, including the visualization of technology through a big picture, the **green from go** philosophy, and how we treat **everything as code**. We'll introduce some of the baseline metrics we've used to measure the success and impact of DevOps culture and practices. We'll even set the scene for some of the technical practice trade offs and approaches to consider when creating your foundation – GitFlow – versus Trunk-based development, setting up development workflows, considering different types of testing, and setting up an environment for pairing and mobbing.

To show and not tell, establishing the foundation is about turning the picture on the left into the picture on the right:

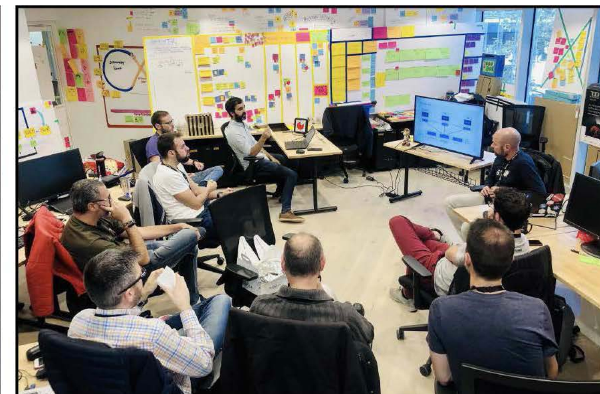


Figure 3.2: Collaboration within the organization

Section 3 — Discover It

Here, we'll dive into the discovery loop of Mobius and look at some of the best ways to use it. We'll share some of our favorite and most impactful practices from the Open Practice Library that have helped us in the discovery loop, including impact mapping, human-centered design, and event storming.

We'll look at how this relates to technology and the idea of emerging architecture and enabling true continuous delivery.

From a business perspective, we'll explore the difference between outcomes and outputs and how we're trying to move from the idea of more features being better to creating powerful outcomes with fewer features. We'll explore some practices for how we can continuously measure outcomes and how we can radiate information from the entire discovery loop on open source canvases.

To show and not tell, we'll look at moving discovery from looking like what you see on the left to what you see on the right:

Figure 3.3: Practicing discovery through impact mapping, human-centric design, and event storming

Section 4 — Prioritize It

Here, we'll dive into the options pivot of Mobius and see why living, breathing, and always-changing options are important. We'll explore practices such as user story mapping and value slicing that help us with this and share some of the gotcha stories we have of where this has been misunderstood and misused. We'll look at how we go about building that initial product backlog using discovery that leads to options pivot practices. We'll look at different types of items that end up in product backlogs, which range from research work to experimentation work and implementation work. We'll look at some economic prioritization models and how to assess the trade-offs between value and risk with the mindset of continuous experimentation and continuous learning. We have lots of stories to share – some with a specific focus area and some with a thin thread of learning across many areas.

To show and not tell, we'll see how prioritization can go from looking like what's on the left to what's on the right:

Figure 3.4: Using the Options pivot to prioritize backlog items

Section 5 — Deliver It

In this section, we'll look at Agile delivery and where and when it is applicable according to levels of complexity and simplicity. We'll also look at Waterfall and the relative merits and where it might be appropriate. We'll explore different agile frameworks out there and how all of them relate to the Open Practice Library and Mobius loop. We'll explore the importance of visualization and of capturing measurements and learning. Technology-wise, we'll look at how advanced deployment techniques now available help underpin some of the experimentation and learning approaches being driven.

To show and not tell, we'll see about getting delivery from looking like the picture on the left to something like the picture on the right:

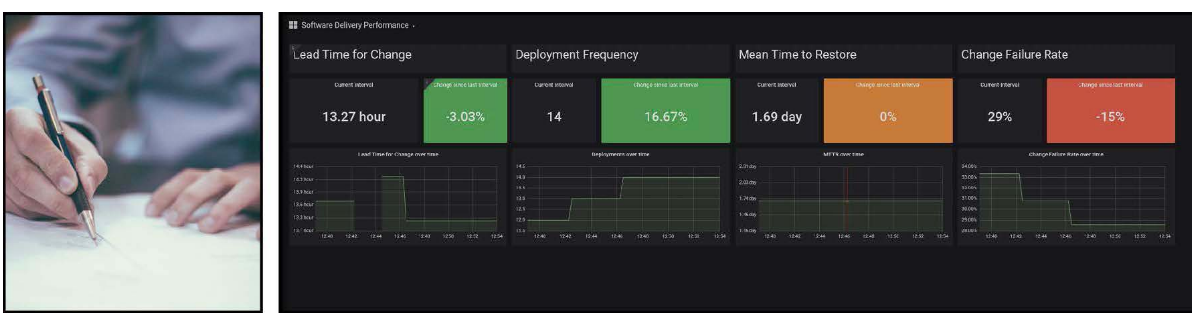


Figure 3.5: Practicing delivery through visualization and measurements

Section 6 — Build It, Run It, Own It

This section really focuses on technology as an enabler and why it is important to have an application platform.

We'll return to the philosophy of everything-as-code and look at Git and Helm as enablers for this. We'll dive deeper into containers and the cloud-native (the cloud, the platform, and the container) ecosystem. We'll explore OpenShift and Cloud IDE, as well as pipelines that enable continuous integration, including Jenkins and Tekton. We'll explore emerging deployment and configuration approaches, such as GitOps through ArgoCD, with guidance on how and where to store configuration. We'll explore advanced deployment techniques, such as A/B testing, feature toggles, canary deployments, and blue/green deployments, and how these are used with business outcome experimentation. We'll look at non-functional aspects of DevOps practices, including **Open Policy Agent** (**OPA**), the scanning of images, DevSecOps, BaseImage, and chain builds. We'll look at some functional and non-functional testing. We'll explore operational aspects such as app chassis, image tagging, metadata and labeling instrumentation, Knative and serverless, and observability in terms of business versus app performance metrics. We'll reference Service Mesh and focus on operators for management and day 2 operation considerations.

To show and not tell, we'll explore taking building and running from being what you see on the left to what you see on the right:

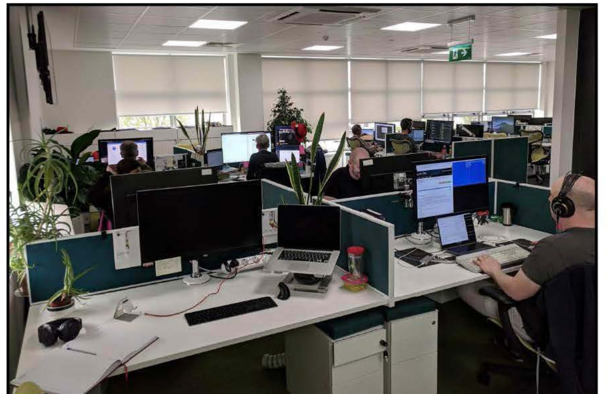


Figure 3.6: Creating the right environment for doing DevOps

Section 7 — Improve It, Sustain It

As we come out of the delivery loop, we'll ask, Did we learn enough? Should we pivot or go round the delivery loop again? We'll see how we're entering a continuous learning cycle – not a one-time thing. Assumptions are proven or disproven during delivery loops. We explore the world of technical debt and how we can bring qualitative and quantitative metrics from the platform, the feature, and the app dev workflow to help radiate this. We'll seek how to take measurements and learning from delivery back into discovery artifacts, such as event storms, metrics-based process maps, and user research.

We'll learn how to blend everything covered in the discovery loop, the options pivot, the delivery loop, and the foundation to help sustain this way of working. This is what enables double-loop learning for continuous discovery and continuous delivery.

Long lived, cross-functional product teams learn to build it, run it, and own it. In this section, we'll look at some of the practices that help them sustain it.

What role does leadership have to play in all of this? We'll show how to visualize the connection between leadership strategy, product development, and platform operations, all being driven by intent and informed by information and metrics.

We'll explore approaches to scaling everything described in the book and how a culture of following principles is more important than pure religious use of practices.

What about Distributed Teams?

When you look at the photographs in the previous section, you may have noticed that the world that we're moving towards involves people collaborating together. We are in the same space, around big white boards with lots of colourful sticky notes.

A common question we receive from leaders, executives and customers is how easy is it to apply these practices such as Event Storming or Social Contracting when people are not in the same location.

During the last couple of decades, there has been a steady increase in offshore and nearshore development models. Some organisations have been restructured resulting in different parts of the organisation being located in certain areas of the world. In some situations, this has resulted in a more siloed organisation with larger walls between different parts of it due to geography, time zones and lack of face-to-face collaboration.

Our answer to whether the people, process and technology practices we're going to explore in this book can be used with distributed teams is **yes, they can**.

However, the speed at which a distributed team will discover, deliver and learn is very unlikely to ever be as fast as the same team working together co-located. The ability to learn and learn fast is foundational to the whole way of working. Therefore we always seek opportunities to find the fastest way to learn and remove bottlenecks that might slow down a team's learning. Working distributed in nearly all situations we've observed is a bottleneck.

Until 2020, questions around whether use of these practices can be applied with distribution have been motivated by offshore teams, availability of skills and, ultimately, cost. Large System Integrators have spent the 2000s investing billions in nearshore and offshore development centres so it's understandable why they will want to do everything possible to enable teams in those centres to be able to use agile, lean and DevOps practices. For Agilistas, this can be frustrating as the focus of agile is very much on switching the conversation to be about value rather than cost and how to continuously learn and improve in order to optimise value.

The year 2020 saw a new and significantly enhanced motivation for distributed teams - the COVID-19 global pandemic. We were in the early stages of writing this book when the pandemic was declared and governments started talking about lockdowns and other severe restrictions. From March 2020, most if not all people reading this book will recall that their working and personal lives changed significantly. The vast majority of IT development and operations was suddenly performed from employees' homes. The opportunity to co-locate for any kind of in-person collaboration was severely restricted by companies, by governments and, given health risk, was a detractor for individuals themselves.

Like most, we had to pivot our own work and modify our own ways of working with our customers during the pandemic. Regardless of preference or prior thinking on distributed working, we now had to explore and invest in distributed working practices. For us personally, this meant the launch of the Open Innovation Labs Virtual Residency and other services to be provided remotely and distributed.

When we released this first section of this book as an early preview for feedback, we were strongly encouraged by our readers to explore distributed use of practices more than we were originally planning to. So, we will. In each of the sections of this book, we'll include a section to share our experience of applying practices with distributed teams. This will include stories and experiences from the field during 2020 when we did this and the many learnings we got from doing so. Where relevant, we'll also include details and links to digital templates that have supported us.

A note of caution. Just investing in a tool like Mural or Miro and having access to a bunch of templates will not enable you to carry on as normal with all the practices and techniques you may have used in a room. We've already outlined the importance of getting the balance of people, process and technology change to get successful DevOps Culture and Practice. When switching to using distributed mode - teams need extra and continued focus on people, process, and technology.

Some Words about the World of 'Open'

The term open has been used several times in this book already and it will be used many times more. We work for an open organization, a company built on open source principles and characteristics. We're using the experiences of Open Innovation Labs to tell many of our stories, and all the practices we're using are captured and will continue to evolve in the Open Practice Library.

Figure 3.7: Default to open

We strongly believe that open culture and open practices using open technology makes the best cocktail for successful transformation.

Conclusion

In this chapter, we introduced PetBattle and the backstory of the hobbyist app that will form our fun case study we'll use throughout this book.

We also introduced how we'll regularly break out into real stories and examples from work we've done with our customers.

Finally, we set out the remaining sections of the book and what we'll explore in each of those sections.

Our introduction is complete. Let’s start working our way round the Mobius Loop and exploring our most used practices. Before we get onto the loop, we’re going to the foundation the loop will stand on. In the next chapter we’ll start by building the very important foundation of culture.

Section 2: Establishing the Foundation

In *Section 1*, *Practices Make Perfect*, we introduced DevOps and the practices and tools we're going to use to navigate around the Mobius Loop, which we also introduced. Before we get onto the loop, we're going to build a foundation for the loop to stand on. This is a foundation focused on building culture and technology:

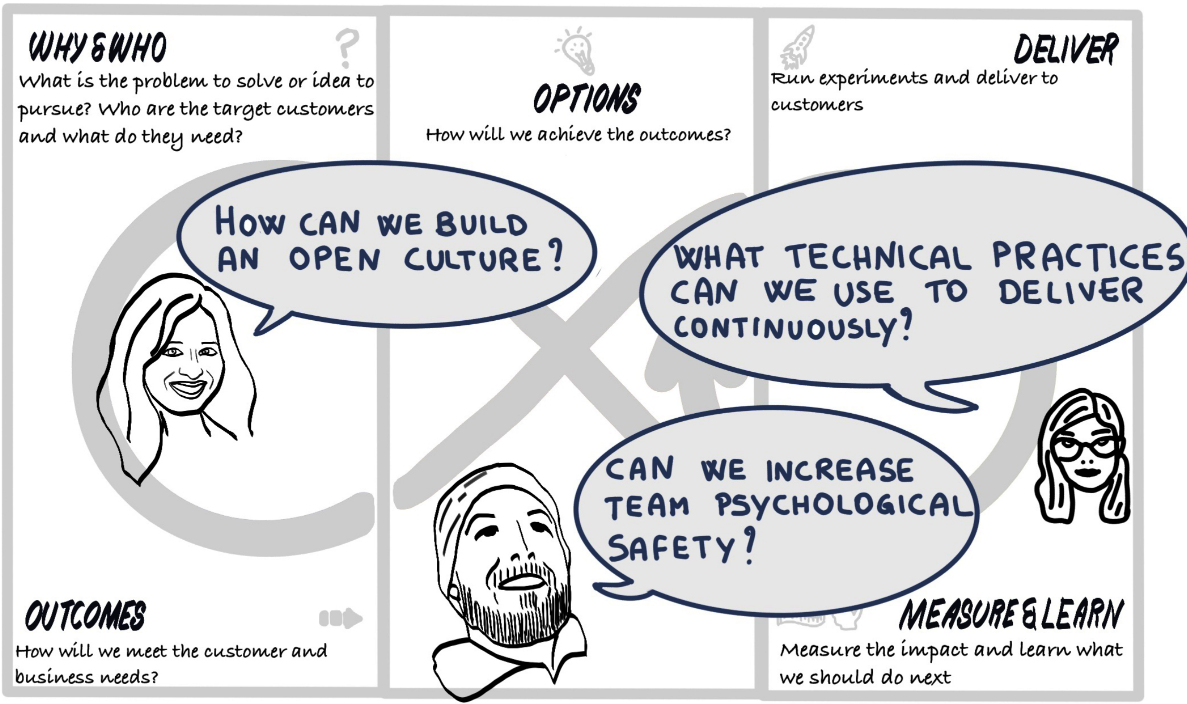


Figure 4.0.1: The Foundation – setting the scene

When you hear the word **Foundation**, what do you think of? A lot of people will think of the foundations of the building you are currently sitting or standing in. As you know, you cannot just turn up and start building a house on top of some land. I mean, theoretically, you could. But not if you wanted something to stand the test of time and more importantly meet building regulations! First, you need to build a solid foundation. That means going beneath the surface of where you're going to build. The taller the building, the deeper and stronger the foundations need to be.

When you think about it, this kind of applies to anything in life. When you lay a solid foundation, incrementally building on top of it has a much higher chance of success. Think about your education, your health, your fitness, your career, and your life. The successes you enjoy are because of the foundations and what you have been able to build on top of them.

What happens when you build on top of a weak foundation? Well, generally, it's not good news:

Figure 4.0.2: Building on a weak foundation

When working with organizations to build applications to run on **OpenShift Container Platform (OCP)**, we see much greater success and return of business value when those organizations invest time and expertise to build a foundation and a solid approach for their development and operations.

In *Section 1*, we introduced DevOps and some tools we're going to use throughout this book—namely the Mobius Loop and the Open Practice Library. The Mobius Loop acts as a navigator tool for teams seeking to apply DevOps on a journey of continuous discovery and continuous delivery. The principles we've just discussed about needing a solid foundation before building anything also apply in the field of software design. We therefore added a foundation in our Open Practice Library. Before we even think about putting any teams, people, or products onto that Mobius Loop, we need to build a foundation. And not just a flimsy and minimal foundation – we need to build a rock-solid foundation. One that is going to support fast movement and an increasing number of folks jumping on the loop above it. What do we mean by a foundation? We mean a foundation of culture and technical practices:

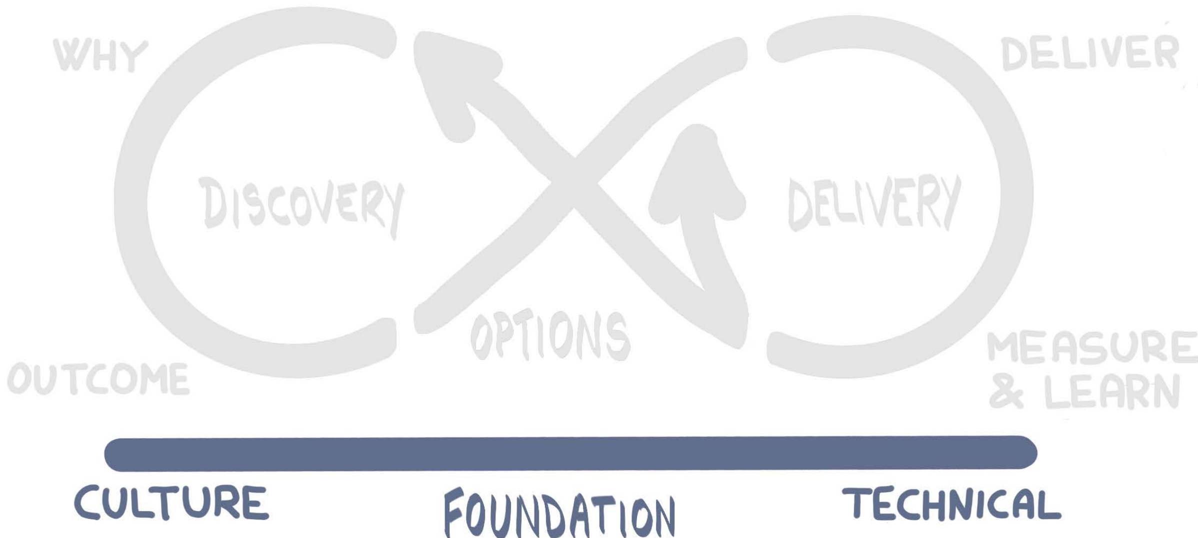


Figure 4.0.3: Focus on the foundation underpinning the Mobius Loop

At Red Hat Open Innovation Labs, we meticulously drive an outcomes-based delivery approach. The Mobius Loop acts as our navigator. It is a visualization tool that helps us to navigate, identify, and articulate the practices that we use at different points of our DevOps journey.

When we are on the **Discovery Loop**, we identify and use practices that help us answer the question of Why – why are we embarking on this journey? What problems are we trying to solve? Who are we trying to solve them for and what do we know about them? What is our great idea? We also use practices on the Discovery Loop to help us identify and set target measurable outcomes for the business and their customers.

When we are at the **Options Pivot**, we use practices to identify how we are going to achieve measurable outcomes. What are the ideas we have that we could implement to help us get there? What are the hypotheses that have resulted from our discovery that we can test, run experiments on, and conduct research? How can we prioritize these options so we deliver value more quickly?

When we are on the **Delivery Loop**, we are using practices to do the work identified on the Options Pivot – implementing the features, running the experiments, and conducting the research. We crucially also use practices that allow us to take measurements and capture learning about the impact of our delivery. And, as we return back into the Options Pivot, we assess what we should do next based on those measurements and learning.

The Mobius Loop is this infinite, continuous journey of continuous discovery, Options Pivots, and continuous delivery of measurable business outcomes that matter. We use practices on the Mobius Loop that typically have defined start and end points. The practices are designed to help a team advance around the loops – for example, in their Discovery process, to make their way to options and a decision point.

When creating the Open Practice Library – a toolbox to store all of these practices – we found that many of them did not necessarily fit within the Discovery or Delivery Loops. Their nature was ongoing, or continuous. For example, we use many practices and techniques that help establish and drive a culture of collaboration. We use tools that help increase the energy, autonomy, and empowerment of product teams. We use practices that help drive an environment built on information radiation, transparency, and continuous learning. The concept of the **Foundation** beneath the Mobius Loop was added to explain these kinds of practices. These practices are designed to make it easy for teams to do the right thing by default. They are practices that we use to build a foundation of culture. There are also many technical practices that we use on an ongoing basis. The first instinct might be to associate these practices with delivery and argue that they should belong to the Delivery Loop. However, there is a subtle difference given that practices sitting on the Delivery Loop tend to be time-boxed, scheduled bursts of activity that help drive delivery. Practices such as Sprint Planning, Showcases, and user acceptance testing events (all of which we'll explore in detail in *Section 5, Deliver It*) tend to be scheduled at a particular time in the week or iteration. There are a host of practices that you would not schedule in the same way that you would, for example, a Sprint Planning session. These include practices such

as continuous integration and continuous delivery, Infrastructure as Code or, in fact, Everything as Code, and practices around automation, such as test automation. You don't do a couple of hours of continuous integration every Wednesday morning! You don't schedule a period for Infrastructure as Code at the end of the week. These are things you do all the time, and doing them all the time is what makes them so powerful. The **Foundation** beneath the Mobius Loop is also used to explain these kinds of practices – practices that help build a foundation of technical engineering excellence.

Similarly, Discovery-type practices tend to be focused events run at a particular time, sometimes multiple times as teams do multiple iterations of the Discovery Loop. The practices that we'll examine in detail in *Section 3*, *Discover It*, are all executed by a group of people gathering to focus on that activity for a period of time. Practices more focused on generating culture, such as creating a team charter and social contract to define ways of working (which will be introduced in the next chapter, *Chapter 4*, *Open Culture*) do not sit on the Discovery Loop. This is because teams get the most value out of these practices when they use them all the time. They don't get their social contract out for half an hour on Monday morning and then never consider it until the next Monday morning! This is why we build them into the **Foundation**.

Now an important aspect of the Foundation and, in fact, the whole Open Practice Library, is that it does not prescribe or mandate any one practice over another. This is not another methodology or framework that spells out, in a prescriptive way, the tools and practices that someone or some team must use to achieve an outcome. We use the Mobius Loop and the Foundation to visualize the different practices we are choosing to use. There is no right answer to say how many practices you should use when building a foundation of culture and collaboration.

We find that if you don't focus on or prioritize these practices at all, teams struggle to achieve outcomes, remain stuck in old ways of working, or fall back into the status quo when attempting change. Many organizations send their staff on a two-day Scrum training course and try to implement it by the book, only to struggle to see the impact of moving toward business agility. Scrum is primarily focused on the Delivery Loop and Options Pivot and organizations are not considering the cultural change or technical practices needed to sustain and promote the benefits that Scrum practices provide.

In this section of the book, we will show you how to build an initial foundation of culture, collaboration, and the required technical practices, so that your house will stand for a very long time!

You may have taken a peek at the Open Practice Library following its introduction in *Section 1*, *Practices Make Perfect*. If you applied the filter to look at all the foundation practices in the Open Practice Library, you will see that there are a lot! You may be wondering, *Do I need to use all those practices*?

The answer is no. The foundation of the Open Practice Library is a growing selection of tools and practices that will help grow an open culture. You don't need to use them all but you should start with at least a few. You may even have a few of your own! Ask yourself whether you are achieving an open culture and whether the tools you are using are working well for you.

4

Open Culture

Many development teams, operations teams, and even cross-functional product teams have a tendency to jump straight into the technological aspect of the project. *Let's set up pipelines*, *let's automate our infrastructure*, *let's write some code!* It's such a natural temptation to dive straight in without investing at least some time in establishing a cultural foundation. It can also lead to ineffective and unnecessary complexity.

Don't worry, we will get to the technology in *Chapter 6*, *Open Technical Practices – Beginnings*, *Starting Right*. But, before we do, we want to talk about open culture and open leadership. In this chapter, we are going to explore what open culture is and why it proves to be such a critical enabler for development and operations.

Open culture stems from a performance-oriented approach to collaborative work. It provides motivated teams of individuals with an environment where they can be continuously inspired to work together and feel a sense of autonomy, mastery, and purpose. We are going to explore what these three elements entail and provide some good and bad examples of them.

We're going to dive into our Open Practice Library and explore some of our favorite practices that we have used to establish a foundation of culture, such as social contracting, stop-the-world cords, and real-time retrospectives. These are practices to help establish team identity, understand team sentiment, and visualize all of our work in a very transparent manner.

We have an array of stories to share about using these and other practices and we'll return to our PetBattle organization to see how they go about establishing an open culture as they move from being part-time hobbyists to establishing a high-performing team ready to take over the pet battling world!

Why Is It Important?

Culture refers to shared customs or behaviors among members of a group of people. Open culture refers to the custom of defaulting to open principles of transparency, collaboration, community, inclusivity, and adaptability. Studies have shown that high-performing teams need psychological safety, and open culture reinforces the behavior patterns that allow for this.

What do we mean by psychological safety? William Kahn introduced this term in 1990,[1] explaining *Psychological Safety was experienced as feeling able to show and employ one's self without fear of negative consequences to self-image, status, or career. Dr Amy Edmondson* provided further explanation in her 1999 paper,[2] where she stated, *Psychological safety is a belief that one will not be punished or humiliated for speaking up with ideas, questions, concerns, or mistakes*. This was taken from her 1999 paper that got picked up by the Google Project Aristotle in 2013. That Google study found that psychological safety was (somewhat surprisingly) the most important factor for building high-performing teams.

Tom Geraghty, *Transformation Lead for Red Hat Open Innovation Labs*, recently redefined this as a part of his own research and transformation services work with his customers. He explained, *In a group context, psychological safety manifests as the belief that other members value your contributions and concerns and will not harm you, actively or passively, for expressing them. It creates space for group members to take calculated risks, admit vulnerability and acknowledge mistakes without fear of negative consequences.*

Open culture and psychological safety are essential ingredients of any organization. They are the enabler for an environment where people and teams feel an echo of trust with the freedom to explore new ideas, conduct experiments, learn by doing, and share freely with colleagues, peers, leaders, customers, users… in fact, everyone!

---

1 Kahn, W.A., (1990). Psychological conditions of personal engagement and disengagement at work. Academy of management journal, 33(4), pp.692-724.

2 Edmondson, A., (1999). Psychological Safety and Learning Behavior in Work Teams Amy Edmondson. Administrative Science Quarterly, 44(2), pp.350-383.

Conversely, a closed culture has blockers and barriers all over the place that prevent these kinds of behaviors. Teams fall back into protective and secretive mode. When there is a fear of sharing bad news, or the idea of asking for feedback fills someone with dread, we end up in an environment where we protect what we're doing from the outside world. We don't want to share our code with others in case they shoot us down or ridicule us for not getting it right. We don't want to show our managers when we are behind the plan for fear of being mentally beaten up and being told we have to work nights and weekends to get back on track. We don't want to show features to business people or end users because they might tell us that what we've done is not right, which will trigger more re-work (and more angry managers).

By creating these types of environments, what we are really doing here is delaying the inevitable. Think about how many IT programs have run into problems that have only surfaced toward the end of the development timeline. Teams suddenly have to work above and beyond to meet a deadline. What was missing was an open, honest, and transparent view of work progress as the project developed.

Have you ever heard of the watermelon effect on IT projects? Nice and green and healthy on the outside but when you start to peel back the skin and look inside, it's red everywhere! An example we've seen is where team leads and project managers write reports with their own RAG (Red, Amber, Green) status and they all get passed up to a more senior project manager who summarizes everyone's reports with their own summary RAG status. That gets passed up for even more senior executive steering governance and an even higher level (or more diluted) set of information with a RAG status presented to senior customer stakeholders.

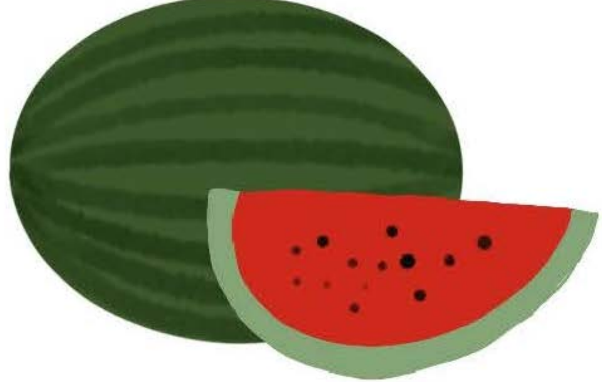


Figure 4.1: The watermelon effect

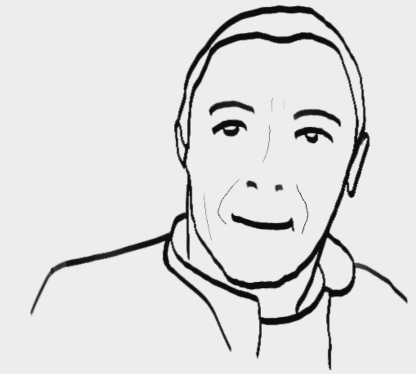
Information Radiators

The best way to encourage an open culture is by having information openly available to all team members and other interested stakeholders.

The term **information radiator** was coined by *Alistair Cockburn* for any of a number of handwritten, drawn, printed, or electronic displays that a team places in a highly visible location, so that all team members, as well as passers-by, can see the latest information at a glance. In Cockburn's book *Agile Software Development*, written in 2001, the information radiator formed part of an extended metaphor that equated the movement of information with the dispersion of heat and gas. We'll look at the many different types of information radiator we use with our customers and teams – counts of automated tests, velocity, incident reports, continuous integration status, and so on.

We also sometimes use the term **information refrigerator** – where information becomes cold and sometimes out of date. The information refrigerator is somewhere you need to go looking for information rather than it being readily available and accessible. The refrigerator may even be locked! Information stored in files on shared drives, JIRA repositories, and other digital tooling all risk become refrigerated, so it's up to us to keep them warm and radiating.

Can You Make Those Red Lights Go Green, Please?



Working with a UK retail organization, my team had instigated the idea of transparent, information radiation.

The project had been reporting green to the business stakeholders for months. But there were some key dependencies and blockers preventing progress to the delivery teams. Our team had set up TV monitors to highlight that none of our services could get access to pre-production or production servers. Some simple automated tests would show no traffic getting through the network infrastructure and therefore all tests were failing. These test results were being shown on a big monitor in the team's room. The monitor, which was very visible to lots of people, just showed big blocks of red.

> As stakeholders had been, to date, reporting a healthy, successful program with no big issues, having big monitors showing lots of red gave an opposite and worrying message. I was asked if we could change these *traffic light* information radiators for a day because the CIO was coming in. If not, could we just turn the monitors off?
>
> This was a classic watermelon effect.
>
> NO – don't falsify the real-time information radiator that the team has set up.
>
> DON'T hide the information. Instead, address the infrastructure issue that the information radiator is highlighting. That'll automatically turn the dashboard green the next time the automated tests run.
>
> In an effort to continue empowering our team, we chose to ignore this request.

When we have an open culture, we have the opportunity to regularly inspect the health of everything – the health of the product, the teams, the investment, the stakeholders, the users, and the technology. An open culture means that when we do a health inspection and it does not give us positive news, we welcome the bad news with engagement and conversation. We do not berate people for giving bad news. Instead, we talk about what we have learned and what we need to do to make things better. What do we need to change? What should we adapt? Inspection and adaptation are healthy indicators of an open culture. The more open the culture, the greater freedom we have to inspect and adapt as we go along.

Before we get into a few specific practices that help build an open culture foundation, let's share a few thoughts on what we mean by that word **culture** and how motivation within teams is what can really drive culture up.

Culture

Culture is a mysterious energy. You can't see it but you can certainly feel it. You can feel when it's particularly strong within a team or in a physical space. If you've ever known what great open culture feels like, you'll also quickly know when you can't feel it.

We often liken open culture to the force in Star Wars. The force was strong in young Luke Skywalker in Star Wars IV – A New Hope: he wasn't wearing a t-shirt that said so, but others could feel it when in his presence. Open culture is like that. You'll know it when you've got it and you'll know when it's getting stronger. Your job is to regularly assess, sense, and check how strong the open culture is and, if you feel it could be stronger, explore more cultural foundation practices to strengthen it. We'll explore different techniques and practices that help measure and learn from the impact of these changes in *Chapter 13, Measure and Learn*.

In the remainder of this chapter, we're going to share a handful of practices we've had the most success with in establishing a cultural foundation during Open Innovation Labs residencies.

Motivation

According to Dan Pink, author of the number 1 New York Times bestseller *Drive: The Surprising Truth About What Motivates Us*, people are not motivated in the ways we expect. He argues that *Organizations historically have incentivized employees the wrong way by offering rewards (money) and creating a culture of fear and punishment for underachieving. When work requires any cognitive skill or knowledge, then these methods do not work.*

Through his research, he shows there are three things that motivate people beyond basic task completion:

- **Autonomy**: The desire to be self-directed and the freedom to take ownership
- **Mastery**: The desire to get better at something
- **Purpose**: The desire to do something that has meaning or is important to you

Figure 4.2: Autonomy, mastery, and purpose

Creating an open culture in your organization should embody these principles. Open source software development is built on the pillars of autonomy, mastery, and purpose. Examples include using open source code bases, collaborating on public discussion forums, and having transparent decision-making records; these practices make it easier for people to find purpose, gain expertise, and take ownership of their work. It eliminates dependencies on people and on scarce or hidden information. People across the globe are willingly contributing to open source projects such as *OKD* (the community distribution of Kubernetes that powers OpenShift), thus improving the product for all. Sometimes they are contributing because it is part of their paid work, but quite frequently they contribute because they want to. They are working on these projects because of the deep personal fulfillment that they gain through helping a community build something better that benefits everyone, being self-directed or listened to, and building something for the future; they also want to become a better coder by implementing and testing something on a public backlog.

In order to build this kind of ethos in your organization, leadership needs to set the environment for people to achieve these things. They should empower teams to make decisions and adopt self-organization and self-correction, thus providing autonomy. They should embrace and encourage continuous improvement and enable team members to continuously learn and improve themselves and their team, providing mastery. Leadership should remove obstacles and silos that prevent team members from connecting with and appreciating business purpose. They should enable regular and direct access to business stakeholders, customers, and leadership to drive up the sense of purpose to their work. Allowing employees to focus on autonomy, mastery, and purpose will have a twofold effect. Teams will have the freedom and *drive* to create brilliant products. With greater responsibility for the product, people will start to work harder because they truly believe they are listened to and are connected, and they will want to work on making the product better.

Let's get back to our PetBattle case study and see how some of the employees felt autonomy, mastery, and purpose. Specifically, let's look at a few events that happened in the early days of establishing PetBattle that helped achieve this open culture.

PetBattle — Creating Autonomy, Mastery, and Purpose

Connecting the engineers to the end users is a great way to create a sense of purpose. The PetBattle engineers do not want features or tickets dropped on them with no view of how the feature connects to the vision of the product. To combat this, they do regular demos with end users and generate some feedback. This connectivity allows them to build empathy with one another and see how the application is being used in real life.

Mary (a self-confessed cat lover who plays PetBattle regularly) was being asked to use the app while Ciarán (a member of the PetBattle engineering team) observed her. Mary was struggling to understand why her cat was not appearing immediately in the competition after she'd uploaded it. Ciarán knew she had to vote for another cat before hers would appear in the list, which seemed obvious to him. A few days after this feedback, Ciarán sees a ticket in his backlog to have the uploaded cat appear in the voting list immediately. Ciarán sees the purpose of this ticket immediately and starts thinking about the effect this feature will have on Mary, who plays the game regularly.

Ciarán and the other engineers working on PetBattle – Aidan, Emma, and Jen – recognized that having access to Mary's feedback was really rewarding, especially when they had the opportunity to deliver new features to Mary. *It's so much better than my previous projects, when I would be asked to code to a specification and not really understand what it was that I was helping to build*, said Jen in a recent retrospective meeting. Aidan suggested it would be even better if they could meet users like Mary every now and again, perhaps working with them to do some testing and hear all their other ideas for future development.

During that same retrospective meeting, Aidan called out his personal highlight from recent weeks working in the PetBattle engineering team. He said he loved how he could just release his new features and bug fixes when he was ready. He didn't have to wait for a release window or need a change review board to meet. He could just push the button and the awesome technical foundation practices that Ciarán and Emma in particular had focused on putting in place meant he could ship with confidence. He was learning to create feature toggles around his new code, which was even better as it empowered the business product owner to decide when to

switch on the feature. Aidan thinks the team should look to explore more advanced deployment capabilities like this.

Meanwhile, Emma's highlight was having the opportunity to share openly with the wider community some of the technical foundation tips she'd picked up. She has written several blog articles and has been invited to present at a DevOps conference in the new year. She said that, in her previous projects, she wasn't allowed to share so openly the work she had done. She felt she had some great techniques to share back with the wider technical community. Many folks in this community had helped and inspired her own development, so she was very happy to share back. Everyone agreed they should seek opportunities to do these kinds of things – who knows, maybe it will even help promote and advertise PetBattle as a product and, as they grow the team, a really cool place to work!

Dan Pink, author of *Drive: The Surprising Truth About What Motivates Us*, gave an awesome talk at the **Royal Society for Arts** (**RSA**) in 2010 all about what really motivates us. The RSA produced a short video[3] from this talk, which is awesome – we've shown it to many teams and leaders. It's openly available on YouTube and we really recommend watching it at least once to explore this area of autonomy, mastery, and purpose further.

At Red Hat, autonomy, mastery, and purpose are values that underpin much of our open culture. We're going to explore tactical practices that help create this. As we move into product discovery and product delivery, you will find that all practices here are also helping realize this culture.

Social Contracts

We, the four authors, are in a very fortunate position where we get to help kick-start new product teams. We use a range of practices to help bootstrap these teams and get them off to the best start possible. We've come to realize that one practice in particular is foundational – social contracts.

3 https://youtu.be/u6XAPnuFjJc

The social contract practice is a simple yet highly effective way to enable team autonomy and self-accountability for engagements. A social contract is created by and for the team. It looks to codify the behaviors and expectations of the team. It also provides a mechanism for the team to visualize and share its desired behaviors with management and other stakeholders.

A social contract is not some big expensive consulting engagement but a simple set of guidelines and behaviors put forward by a team for how they would like to interact with one another. It promotes autonomy and self-governance within a team.

One thing we've learned from previous engagements is that if we don't invest time and use tools such as social contracts upfront with new teams, we risk running into all sorts of cultural, teaming, and communication problems over time. *Figure* 4.3 shows an example of a social contract.

Figure 4.3: An example of a social contract

Do I Need One? If So, How Do I Build One?

In modern software development, we're all about tearing down walls. The wall between development and operations teams was only the beginning! Open organizations want their teams to act with more autonomy and so give them shared purpose around their products. Teams react to this by often trying to embody the *You build it, you own it, you run it* mantra. But how can we kick-start this change in people's behavior? How can we accelerate our teams to be high-performing, especially when starting with a new team whose members know nothing about each other? This is where our friendly social contract comes into play.

When we are kicking off with a new team, we get them to come up with a team name. Team identity is very important in setting up the ownership part of a high-performing team culture. We'll explore team identity further later in this chapter.

The next step is simple: get the group to spend a few minutes thinking about the best teams they've ever worked in previously or the best products they've been a part of creating. With this in mind, they should think of all the amazing characteristics and behaviors they can remember and capture them on sticky notes.

Conversely, think of all the terrible projects and what things created that toxic environment for the team to operate in. With some ideas of behaviors that they want to adhere to, the group can discuss them further to get a shared understanding and home in on the specific language to use.

A good social contract will contain concrete actionable items and not fluffy statements. To really tease these out, lead by giving examples of how someone would embody the statements they've come up with and try to capture that. For example, *be open* could be a good seed for an item in a social contract, but perhaps lacks specificity. We can explore this by getting examples of where we are, or are not, adhering to it. Teams may then come up with things like *Everyone's opinion and contribution is valuable*, *Give space to quieter people*, and *Actively listen to others the same way I want to be heard.*

With some items in the contract, the group signs the contract and hangs it high and visibly. It is now the responsibility of the team to abide by it and call out when others do not.

Figure 4.4: Another example of a social contract

The preceding example is from a cyber security company we worked with for 4 weeks. This team had a mix of developers, designers, and site reliability engineers. Some of the things they included were as follows:

- **Core Hours (10.00 to 16.00)**: This is the team's agreed collaboration time. This is not the hours they work, but more the time in which they will have sync ups, meetings, or pairing and other collaboration. On this engagement, one of the team members wanted to miss the early morning traffic, so if she came in a tiny bit later her commute would take her a lot less time. Also, another member of the team had childcare responsibilities so getting out at a reasonable hour would have made his life easier.
- **Mob to Learn / Pair to Build**: This is a simple mantra that describes how the team wanted to interact when writing code, tests, or even documentation. If there are new things to be tackled, such as that scary new microservice in a language or framework that's new, do it as a full team. Get everyone on the same page from the beginning and ensure that you're not creating individual heroes with all the knowledge in the team. Pair on implementing features to raise skills across the board.
- **Have a Weekly Social**: Celebrating success is an important thing for lots of teams. Taking the time as a group to get away from our desks, socialize together, and eat together helps build positive relationships. These events can lead to improving team morale and creating a positive culture around the people working on a product.

Now we've considered the key tenets of social contracts, let's look at how to integrate them with other workplace practices.

To support gaining consensus on ideas and behaviors, use grouping techniques such as **Affinity Mapping**, the **Fist of Five**, and **Dot Voting**. These are simple yet very powerful practices and open facilitation techniques that help drive alignment, consensus, and the inclusion of all people involved. We will explore them in more detail in the next chapter about open leadership.

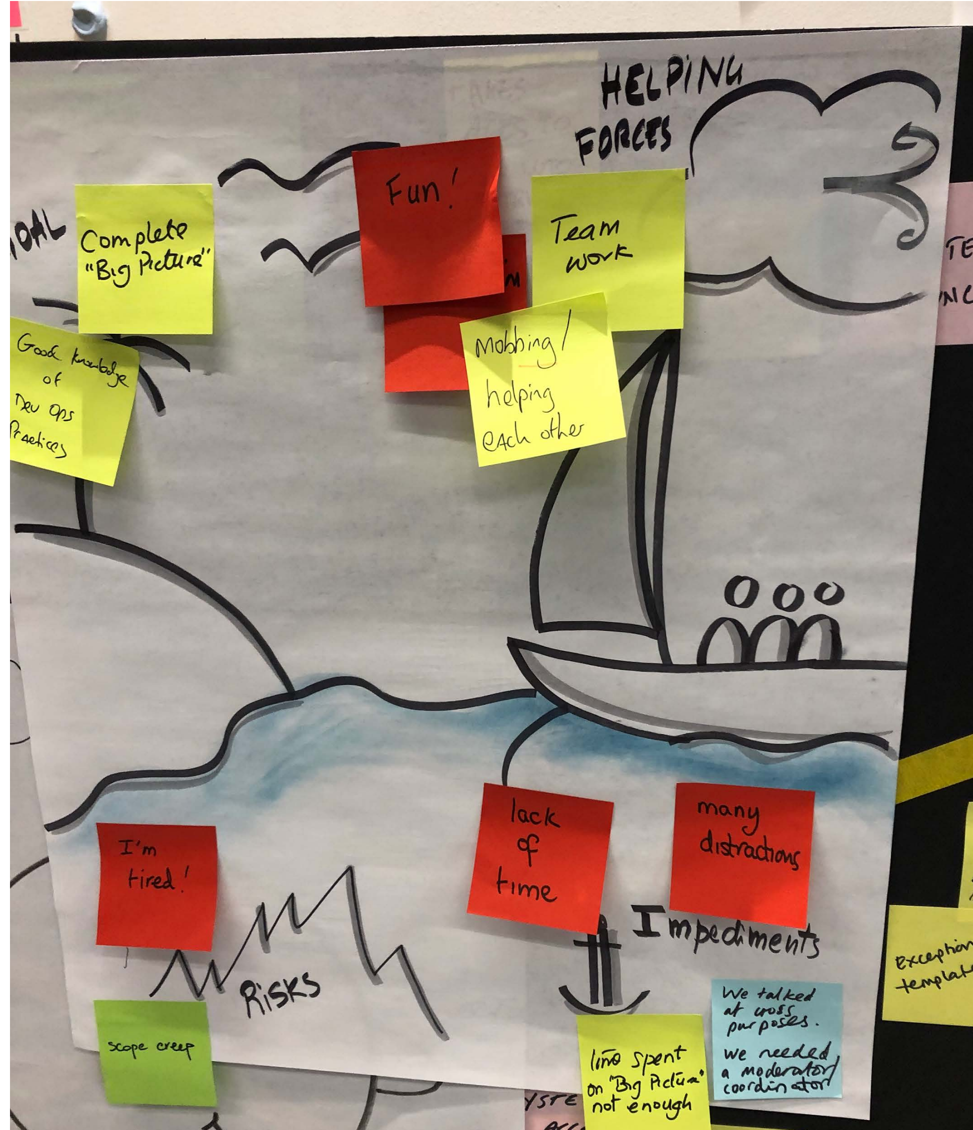


Figure 4.5: Retrospective learnings driving social contract updates

When the social contract is first created, it represents a snapshot in time. It represents a best guess of how a team should interact and is often written when we know the least about each other or our habits. It is a useful tool to accelerate a team to a comfortable position built on trust and psychological safety. However, it is not a fixed or static thing. Items in there could be invalid or possibly missing. A great thing a Scrum Master or Agile coach can do is bring the social contract along to a team's retrospective (a practice we will explore in more detail in *Section 5 - Deliver It*). It provides a great opportunity for the group to inspect and adapt it, possibly updating it with new ideas or improvements.

It's OK to Be Wrong

When working with the cyber security company, we found a great opportunity to update our social contract. There were two developers on that team that were quite hot-headed and both always liked to be right. Often, they could not agree on the approach to take for any given solution, which was creating a toxic vibe within the team and reducing morale. Through a retrospective, the team identified this issue. After some discussion and getting the issue out into the open, we updated our social contract with one simple phrase: *It's OK to be wrong.*

This had a profound effect on the team going forward. The competition within the team started to evaporate as we could focus less on *who was right* and more on *does the solution work for our end users*. We looked for opportunities for both developers to have their voices heard by trying more than one implementation of a feature. Building both solutions and evaluating each, both developers then started to pair on building out each other's solution, thus creating a better bond between the two. This also had the net benefit of writing a much better solution by often combining ideas from both individuals.

Social contracts are a powerful and easy-to-implement practice – they promote team autonomy, self-governance, and psychological safety. If your team does not have one, it's not too late to build one. Most importantly, keep it visible to the team and don't forget to revisit it when needed.

Social Contracting for Distributed People

We mentioned at the end of *Section 1, Practices Make Perfect* that we would consider how our practices work in a more distributed environment where people are not physically in the same place. Using the social contract practice with a new virtual team can be tricky as we tend to use social contracts at a time when relationships, trust, and rapport have not yet formed.

The challenges we've seen include not having everyone engaged or contributing to the social contract. Hearing from the same voices puts those not so involved in the background. What follows are some recommendations on how to approach social contracting with distributed team members.

Using a virtual whiteboarding tool such as Miro or Mural with everyone connected and looking at the same artifact will set you up for the best possible success. As shown in *Figure* 4.6, we have created some templates, which are openly available and can be used as a starting point for your social contract. You can download this from the book's GitHub repository.

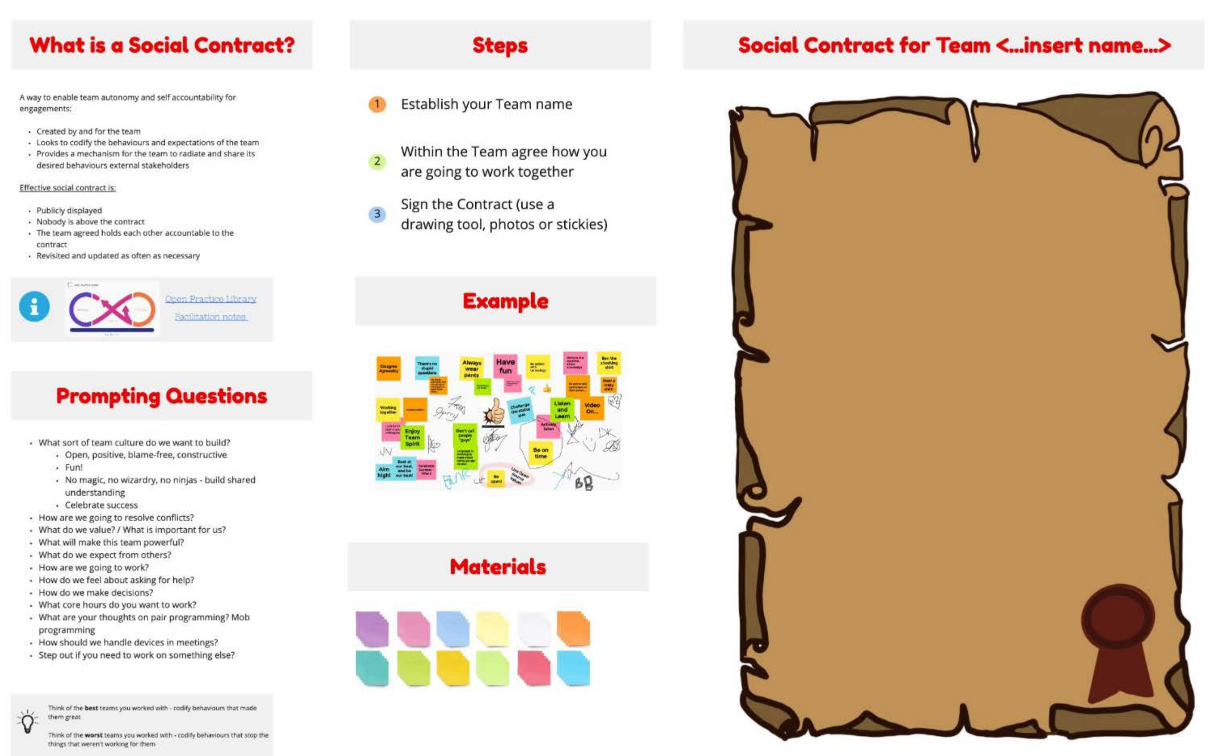


Figure 4.6: Digital social contract template for a distributed team

Consider having a *pre-canned* social contract that is good for remote meetings, then allow people to add/change based on their thoughts:

- Everyone should *sign* their name with a sticky note or virtual pen.
- Add remote working elements to the social contract (for example, mute on entry to calls, turn your webcam on, and so on).
- Establish global communication norms and individual teams' exceptions: response time, writing style, tone, and so on.

Some good examples of communication norms we've seen in our virtual engagements include the following:

- One conversation at a time.
- Assume positive intent.
- Do not delete or move others' contributions.
- Stay focused on our goal, working iteratively.
- Respect breaks.
- Avoid other distractions such as email and chat.
- A contract can be updated based on needs.

Figure 4.7 shows an example social contract created by a team distributed with their suggestions and signatures:

Figure 4.7: An example social contract used by a distributed team

You can learn more about the social contract practice by going to the Open Practice Library page at openpracticelibrary.com/practice/social-contract.

Creating a social contract over a video call with everyone remotely takes an extra level of facilitation skill. There are some great tips for facilitating this and other practices remotely in a blog post contributed by *Ilaria Doria* and *Marcus Maestri* at https://openpracticelibrary.com/blog/facilitation-tips-for-remote-sessions/.

Stop the World

The **Stop the World** event or **Andon Cord** is another of our favorite practices that we use regularly in our engagements and is a DevOps superpower.

John Willis explained the origins of the word **Andon** in his *ITRevolition blog post*[4] – in Japanese, it comes from the use of traditional lighting equipment using a fire-burning lamp made out of paper and bamboo. This idea was later translated for use in manufacturing in Japan. The **Andon** became used as a signal to highlight an anomaly (that is, a flashing light). This signal would be used to amplify potential defects in quality.

4 https://itrevolution.com/kata/

The Andon Cord and Psychological Safety

The first time I heard of the *Andon Cord* was when reading *The Phoenix Project – A Novel About IT, DevOps and Helping Your Business Win* by *Gene Kim*, *Kevin Behr*, and *George Spafford* – if an employee on the car production line suspects a problem is happening, they pull the cord and the whole production line immediately stops. By avoiding passing failures downstream, Toyota credits this disruptive behavior as being *the only way we can build* 2,000 *vehicles per day – that's one completed vehicle every* 55 *seconds.*

About eight years later, I heard Gene Kim talk about the relationship between the Andon Cord and team psychological safety. When you pulled the Andon Cord in the Toyota factory, the line manager ran up to you and THANKED YOU for stopping production so that whatever the problem was could be resolved. No-one would pull the cord if they thought they were going to be punished for it.

Yes ... Good DevOps you borrow, great DevOps you steal!

The process of stopping a system when a defect is suspected goes back to the original Toyota System Corporation and something called Jidoka. The idea behind Jidoka is that by stopping the system, you get an immediate opportunity for improvement, or to find the root cause, as opposed to letting the defect move further down the line and be left unresolved. The Jidoka concept was pioneered by the original Toyota founder, Sakichi Toyoda, also known as the father of the Japanese industrial revolution and also the founder of the original Toyota Systems Corporation (before they manufactured automobiles).

We use the *Stop the World* event during our engagements with customers. When somebody on the team identifies that the engagement is off-course, the full team works together to find a solution that is actionable and acceptable to all and progress can then resume.

There are a lot of hidden benefits to using Stop the World events within your team. In the Toyota factory, the line manager would physically go and see the station where the cord had been pulled and ask: *how can I help you*? Immediately the issue is treated as a priority and by going straight to the area where the problem is first raised, the process becomes evidential and fact-based. By thanking the team member who pulled the cord, this encouraged a *safety culture*: factory management was saying, *You have saved a customer from receiving a defect*. Whenever your application tests fail the pipeline, think of this! At their core, Toyota believed failure created learning opportunities and that failures are good things because of that.

We're Just Rebuilding the Same Experience. Stop the World!

We spent 6 weeks delivering a dual-track Open Innovation Labs residency to a customer in the Nordics in 2018. (A dual-track residency has one track/team focused on building a brand-new production OpenShift platform while the other team builds and maintains the first business applications to run on it. The two tracks collaborate closely together and work in the same physical space.)

For this client, the application being built was a modernization of an existing mobile application used by customers to buy new travel insurance products. The goal of the dual-track residency was to learn new skills around DevOps, develop microservices, and explore the OpenShift product while improving their customers' overall user experience.

Four days into the residency, the team was using a practice called event storming (which we'll explore in detail in *Section 3, Discover It*) to map out the end-to-end business process and discover the emerging architecture. This had been running for a couple of days.

Riley, our UX Designer, had a sudden moment of realization as he looked at the in-progress Event Storm. All the conversation and visualization was focused on capturing the EXISTING application flow. They were going down the path of just rebuilding exactly the same experience but with some new technology and approaches.

Riley pulled the Andon Cord, which, in this space, was a big bell. He stopped the world. It was loud and disruptive and intended to interrupt everyone and grab their immediate attention. Everyone stopped what they were doing and gathered in a circle. Riley explained his concerns and everyone agreed that this approach was not going to help much with the target outcome of improving the customers' overall user experience if they just designed exactly the same application again.

So, as a team, they agreed that they would use the Event Storm to capture assumptions that needed to be tested, ideas for fresh experiments, and lots of questions where there were unknowns that they should explore with end users.

The world was started again and the six weeks that followed took a hypothesis- and experiment-based approach.

That story showed the user-centered motivations behind stopping the world. A second story explains how a more technical team member stopped the world.

Losing Track of Original Purpose

The team here was running a number of experiments, technical spikes, and research efforts to maximize learning during a 6-week Open Innovation Labs residency as a precursor to launching their new cloud platform using OpenShift.

The team had agreed on a number of architectural principles for the residency.

During the third sprint, one member of the team, Tom, noted an increasing number of side conversations and drawings happening on whiteboards. These had nothing to do with the work committed in the current sprint. They had nothing to do with the part of the business process the team was developing user stories for.

The team's burndown chart was well off-course and it was looking increasingly unlikely they would achieve the current iteration's goal. As Tom investigated further what all these side chats were about, he learned there was a lot of discussion about completely different scenarios and thinking about architecture in 2-3 years' time.

A little frustrated, **Tom stopped the world**. There was a *stop the world* bell in the labs space, which he rang.

Figure 4.8: A team member stopping the world

The team were all familiar with the practice but slightly alarmed to hear the bell for real. They gathered around and the team had an open, honest conversation. Tom shared his concerns and frustrations. The team re-visited their current commitments and reminded themselves of some of the

decisions previously made during priority sliders and target outcomes practices. They agreed to put some items on the product backlog to park these discussions.

Figure 4.9: Team gathers after the world has stopped to fix the problem

Within just 15 minutes, the team had re-aligned, re-focused their priorities, and were renewed with energy and commitment.

One wonders if Tom had not aired his concerns and frustrations and just let them sit in the back of his head, how long would this have gone on? What ripple effects might there have been? What would the overall impact have been?

You can learn more and collaborate about the Stop the World practice by going to the Open Practice Library page at https://openpracticelibrary.com/practice/stop-the-world-event/.

Real-Time Retrospective

In seeking to establish an open culture and kick-start an ethos of autonomy, mastery, and purpose across the spaces we work in, we need to have feedback loops in place so we can sense if we're moving in the right direction.

So far, we've just introduced a couple of very simple practices that help to build the foundational culture. But will they work? How will you know? What if the new team is muttering bad things under their breath about how the social contract was fluffy and that Stop the World thing *will never work in a place like this*. Hopefully, this isn't happening, but if it is, you're probably heading in the opposite direction to an open culture by creating side silos and more closed behaviors and thinking.

We need to create a channel for folks to feed back on their own use of the practices, even first impressions. Do they like these things and think they're helping to create the foundation desired? Are there things they already foresee as barriers and blockers?

We've mentioned the practice of the retrospective a couple of times already and we'll be coming back to it several times during this book as the concept behind the practices brings the most important philosophy of continuous learning and continuous improvement.

But the practice being introduced here is a little different. Rather than scheduling in retrospectives to start capturing feedback on our early foundation activities, could we have a more continuous and *always on* self-service approach to capturing feedback?

The real-time retrospective technique shared by Emily Webber provides a simple, visual tool for anyone to add retrospective feedback on any item at any time. Rather than waiting for a formal feedback event or some kind of survey (which, let's be honest, is not everyone's cup of tea), the real-time retrospective enables faster, more accurate, and more direct feedback from everyone involved.

To make use of this technique, find a wall or surface that is long enough to handle the amount of feedback you are expecting. Make a long line to represent time.

Draw a happy face above the line, a sad face below the line, and a surprised face along the line (this represents the type of feedback that you wish to receive – positive, negative, and surprised).

Explain to your participants your goal of using this practice and how to use your prepped area. Here are the key things to share:

- Where to find materials for them to contribute.
- How the timeline begins at the left and continues until the timeframe you designate as the end.
- Any and all feedback is welcome.
- One sticky note per feedback item.

Cycle back and monitor constantly to review your feedback and make appropriate improvements based on them.

We find that making some simple adjustments early based on a real-time retrospective's comments can really sow the seeds for autonomy, mastery, and purpose. Even reacting to temperature control, improving catering, or switching people's positioning around the workspace can have a great positive impact because the feedback came from the people and they see their ideas being responded to quickly. This is what we mean by empowering teams.

As always, we like to show more than tell, so let's look at a real example of where we used a real-time retrospective.

Figure 4.10: A real-time retrospective used throughout a five-day DevOps Culture and Practice Enablement workshop

This was used during the delivery of a five-day immersive enablement workshop – DevOps Culture and Practice Enablement. Given the short nature of this, a real-time retrospective is particularly useful because you don't want to learn feedback at the end of the week (or even the end of one of the days) if there was a simple improvement or resolution that could have been applied much earlier.

You'll see that a lot of the stickies in this example are beneath the middle line and represent the *sad* feedback. Does this mean this delivery was going really badly? No, not necessarily. When we facilitate real-time retrospectives, we actually tell people not to worry about giving us lots of *negative* feedback. While it is nice to read the *happy* or *positive* contributions (and it helps affirm what we should do more of), it's the improvement ideas that we really want to capture. How can we make ourselves better and how can we do that quickly?

You may not be able to read the low-level detail but a few great examples of early feedback captured on the real-time retrospective that were addressed in the second day included the following:

- Having more frequent breaks and reducing the long lecture times
- Ordering more vegetarian food
- Adopting a group policy to *move on* when discussions were starting to *rabbit hole* (separately, the team came up with visual cues to call out when these things were happening)
- Getting some more monitors to help with the collaborative nature of technical exercises
- Going outside for some fresh air at least a couple of times each day

These are all great feedback points from attendees and very easily addressable. The continuous feedback and continuous improvement journey starts here!

One of the things that make the real-time retrospective practice very powerful is that it is always on display and always in sight of the people contributing to and using it. We often use large eight foot by four-foot portable foam boards to put the real-time retrospective on so we can move it around the room and purposefully have it in places that team members have to walk by regularly. This encourages them to contribute, reorganize, or simply see what others have been feeding back.

If you're working with a distributed team using digital whiteboards and video conferencing, we can easily start a real-time retrospective. We have a good template that you can use as shown in *Figure 4.11* and available to download from the book's GitHub repository.

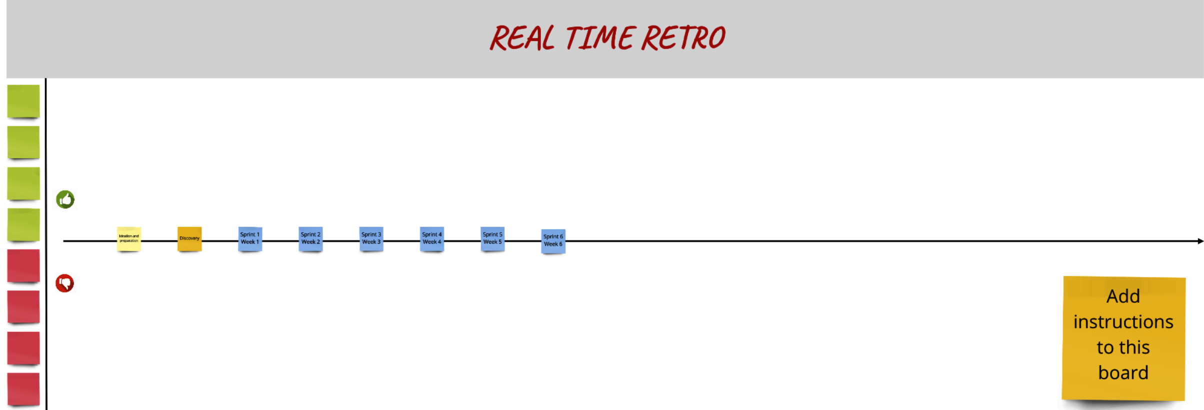


Figure 4.11: Digital real-time retrospective canvas for a distributed team

The challenge here is getting the adoption and usage to continue. Digital artifacts such as this risk becoming **information refrigerators** where information goes to die and is only found when someone opens it up, resulting in it being cold and out of date! Perhaps information should have *use-by* or *best-before* dates! To avoid this, we need strong facilitation. We should encourage all our team members to have these digital artifacts visible at all times. We should, as a team, regularly do a virtual walk-the-walls, and Engagement Leaders and facilitators should encourage contribution to the real-time retrospective where it makes sense.

You can learn more and collaborate about the real-time retrospective practice by going to the Open Practice Library page at https://openpracticelibrary.com/practice/realtime-retrospective/.

Social contracts, stop-the-world Andon cords, and real-time retrospectives are three great practices to start building the foundation of culture. If well facilitated, the conversations generated from these practices alone should start to drive a sense of team autonomy and team identity. We're going to explore team identity further as it is a crucial ingredient of great team culture.

Team Identity

Think about the teams you've worked with in the past. How many of them have been really great experiences that you enjoyed waking up to every morning? How many of them were terrible and you couldn't wait for it all to end? Have you ever worked with a group and thought, *That was amazing – I wish it could have gone on forever*! If so, what were the things that made working with that team so amazing?

Team identity, morale, and cohesion are all linked and are critical for building high-performing teams. They are all a state of mind, meaning the formula for creating awesome teams is a bit more abstract and a lot less formal. You cannot force a team to have high cohesion just by slapping people together, by adding one-part outgoing individuals and two parts hard-working individuals! It must form more organically than this.

Individual team identity can have an infectious enthusiasm and spread to other teams in the organization. When there is a level of energy, team culture, and shared identity evident, other teams want to be like that team. New teams should form to be like that team. And, to avoid the risk of a team's identity becoming their own silo, we can introduce ways to connect teams together through communities of interest and communities of practice. We'll explore this more in *Section 7 – Improve It, Sustain It*.

Socializing

There are lots of little cultural nuggets and practices that we've used to try and accelerate team forming and get us to a performing state quicker. If team members naturally get along with each other or have shared interests, this can infect the whole group and create bonds between everyone.

Figure 4.12: Team social away from the office

Getting to know people on a personal level and sharing interests easily spills over into the day job. Humans by their very nature are social creatures. That's why the first thing we always try to do with our teams is socialize together. Take that moment to get away from the office and relax. Sometimes that can be in the pub after work or even during the day while sharing lunches. In fact, on a previous residency, we've even had things like a weekly curry buffet for the team to use the time to socialize together. On one occasion we went twice in a week when one of the team was off sick on Curry Day!

If going to the pub after work does not suit, flip it on its head! We've often had teams go for a big tasty breakfast together first thing in the morning! It's a great way to start the first day of a new sprint with fresh eyes and fresh eggs! Socializing can take many forms, but it's important to stop and celebrate the work that's been done. Going out to play a game together or hit the bowling alley for an hour helps the team build friendships and get to know one another.

It may sound super fluffy, but teams that share in activities like this will always help each other out on the day job, even if they're unfamiliar with the tasks being tackled. Team members will go the extra mile for one another when they've established such relationships.

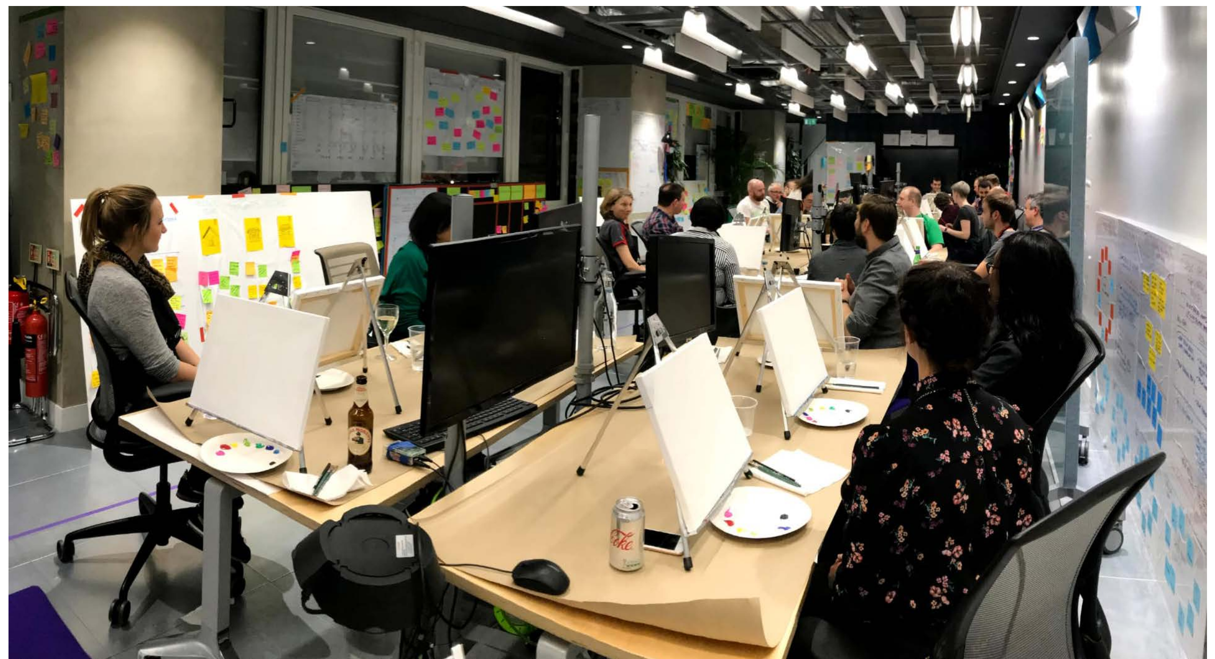


Figure 4.13: Team social in the office

Social music playlists provide a simple way for teams to get to know one another. Creating a shared playlist for all to be able to add songs they like, or upvote and downvote others, can create a very positive vibe within the team. Getting to know people's taste in music, or lack of taste in some cases, can open up new connections with people who may be shy or less likely to engage in a social outing. We've used Outloud.dj for social playlists, even on training events!

Network Mapping

Network mapping is a handy practice for individuals to get to know each other. The practice is simple – line up all the people in your group so each person has a partner they've never met before.

Each person writes their name on a sticky note. Set a timer for three minutes for the pairs to write on a new sticky note something they have in common. With that in place, shuffle the group and go again with a new partner capturing something they have in common.

Two sticky notes are usually enough for each person to get started with this activity, but you could go around again, especially if the group is small. Select one person to go first and have them introduce the people they met. Pop their name sticky note on canvas along with the thing in common and draw a line connecting them. Pass over to the next person and have them introduce the other person they met, continuing to connect the people to the interests. With all in place, allow the team to draw additional lines they have with others in the group, forming a large spaghetti map! It's a simple practice, we know, but it helps to accelerate new groups getting to know one another.

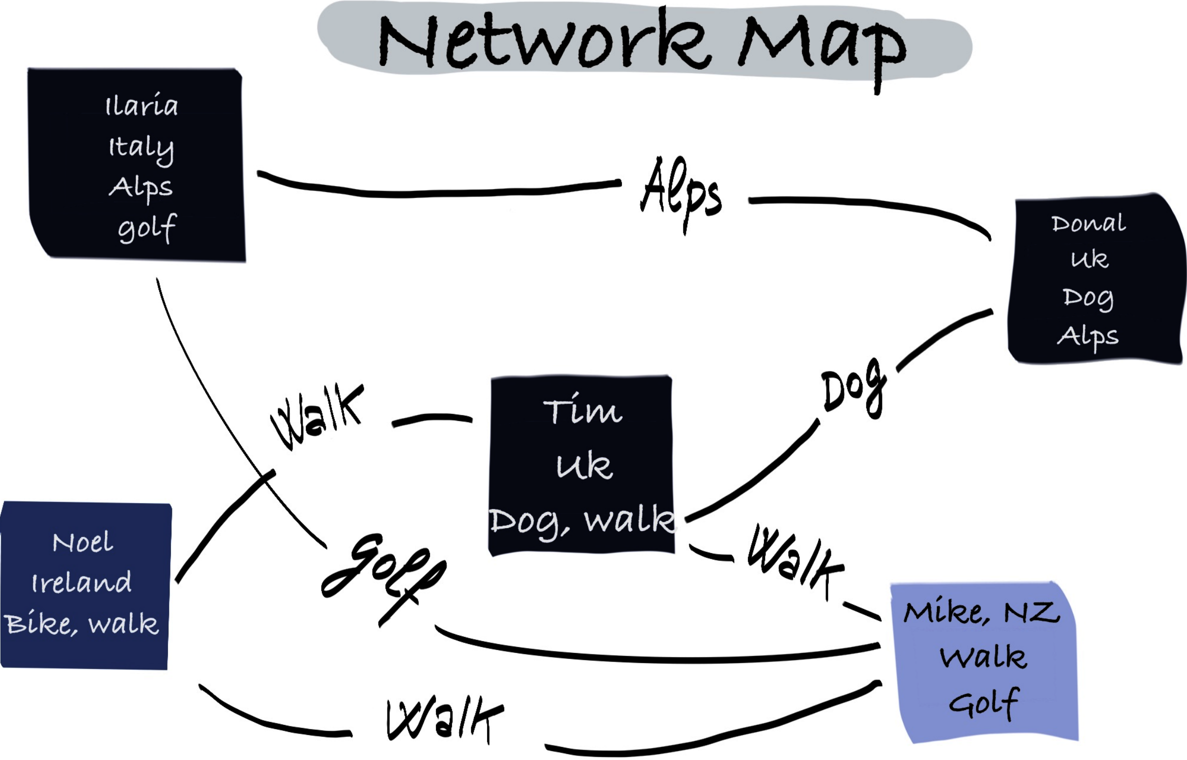


Figure 4.14: Network map example

You can take it one step further to demonstrate metaphorically what happens when you break up a team that's already high performing. With the group standing around in a circle, connect each person who has the thing in common with string or wool. This should form a giant web; take the social contract and lay it on top of the web. To demonstrate the negative effect that swapping in team members or removing team members has on the group's inner workings, cut all the strings connecting those individuals. You will see the social contract is no longer supported and falls away to the floor, symbolizing the effect that moving team members in and out has on the collective.

You can learn more about and discuss the network mapping practice by going to the Open Practice Library page at openpracticelibrary.com/practice/network-mapping.

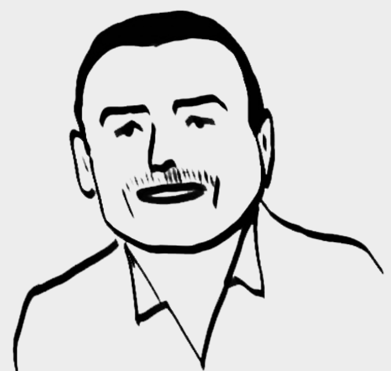
Team Logo and Prime Directive

Another practice that could appear a bit fluffy – but that we stand by – is creating a team name and logo. It strengthens the team's ability to contribute to the team's identity and culture. As the group gets to know each other and starts to build in-jokes, take the time to create a team motto or design a logo that can be printed on t-shirts!

Figure 4.15: Team identity with some customized t-shirts and a team name

Let's look at a real world example where a team name and a team logo kick-started team identity:

Team Name + Building a Team Logo = the Beginning of Team Identity



Some of the best teams we've worked with have co-created logos for their team and worn them with pride. While working for a space and defense customer, we had a team logo designing competition. We took a crest and broke it into segments with each team member doodling something appropriate for the team. We dot voted on all the pieces we liked and collaboratively built the crest from the top four pieces.

Team Space Force was born just like that!

Figure 4.16: Team Space Force logo

A team with a strong identity will feel that they have purpose. All of these little practices and many more can help a team form, storm, norm, and perform as outlined in Tuckman's stages of group development[5]. It's important for teams to be allowed the space and bandwidth to get to know each other. When given space, a team will be happy to self-organize and share problems, and will want to succeed as a unit.

The responsibility of leadership is to just get out of the way and help facilitate the team. If the cost of a pizza and a few beers each week makes a better product or a happier team, it's a negligible cost in the long run. Supporting a team to create bonds like this helps to foster a positive culture. Creating this environment is key for a team to adopt a mantra such as—*You Build It, You Run It, You Own It*.

Creating a Team Identity with Distributed People

Admittedly, during the COVID-19 pandemic, this has been and continues to be one of our biggest challenges and concerns. We knew how special and magical team identity could be and the impact it had in creating an awesome, psychologically safe environment for us to work in. Creating this with a bunch of people we'd never physically met nor were likely to meet any time soon felt impossible!

A variety of online ice breaker activities can help kick-start a distributed culture and are widely encouraged at the beginning of every meeting. Something fun that gets everyone saying something and generating a few smiles up front! One example is to have everyone introduce themselves and share a superpower they have and a superpower they would like to have. All team members are superheroes after all!

Regular break-out sessions and opportunities just to chat informally should not be underestimated. Think about all those conversations you have by the water cooler or coffee machine, over lunch or over a beer at the end of the day. Those conversations still need to happen when the team is distributed.

Building a collective identity may feel like just a nice thing to have but it really drives social bonds within the team. All of the tips mentioned earlier around team name, team logo, t-shirts, and so on, along with having virtual celebrations for events such as birthdays, are arguably even more important in distributed working to build a strong, high-performing virtual team. Even the act of going through this can be done in a collaborative and inclusive way.

5 Tuckman's stages of group development

Finally, don't underestimate the value and importance of being social while having lunch or breakfast together. This is something we build into our Open Innovation Labs residency experience because we see the strength it generates in relationships. So, while it may feel a little awkward, we still do it with our virtual teams. We have our team members show us around their houses, meet their kids and pets, and create fun selfie videos. We have virtual socials – order a pizza and chat over a glass of wine, watch a movie online, and even go to a virtual escape room activity.

A really strong team identity and a high-performing team often result from working in a great team space, full of energy, color, and information that the team is engaging with. Earlier in this section, we introduced the information radiator concept (and the information refrigerator). Let's explore this idea of information radiation further.

Radiate Everything

Have you ever walked into a room housing a high-performing team and found that it just feels different? It's hard to capture why it feels different in text, but you just get a sensation of cohesion and openness all at the same time. You may see sticky notes on a wall showing things being worked on or tracking tools such as a burndown chart visible to all who enter showing the team's progress so far. Possibly you see things such as build monitors showing dashboards of code being built and deployed or tests executing, or even live stats of the applications' and platform's current usage!

All of these things are information radiators, and they are probably one of the most important things we have in our kit bags.

An information radiator serves one purpose – to show whoever passes by the latest information. Radiate all the things, from test scores to retrospectives, architecture diagrams, and even things that could hurt the team! Be proud of them and use them as talking points in an open environment. In a truly agile environment, we emphasize open and honest communication. Hiding behind a tool or burying a problem does not adhere to the principles of transparency. An information refrigerator is exactly that – it's the deep freeze where information goes to die. How many times have you had to request access or sign in to something to find the information you needed? How often has that information become stale or not been updated?

Besides the dashboards and sticky notes we like to have on display, a few other information radiators are good to have in your foundation kit bag:

1. **Visualization of business process design**: The following figure shows an example of this that utilized a practice called event storming. We'll be going into much more detail on this in the *Discover It* section but, as a teaser, you can see the amount of information being radiated – all of it was generated through collaboration and conversations.

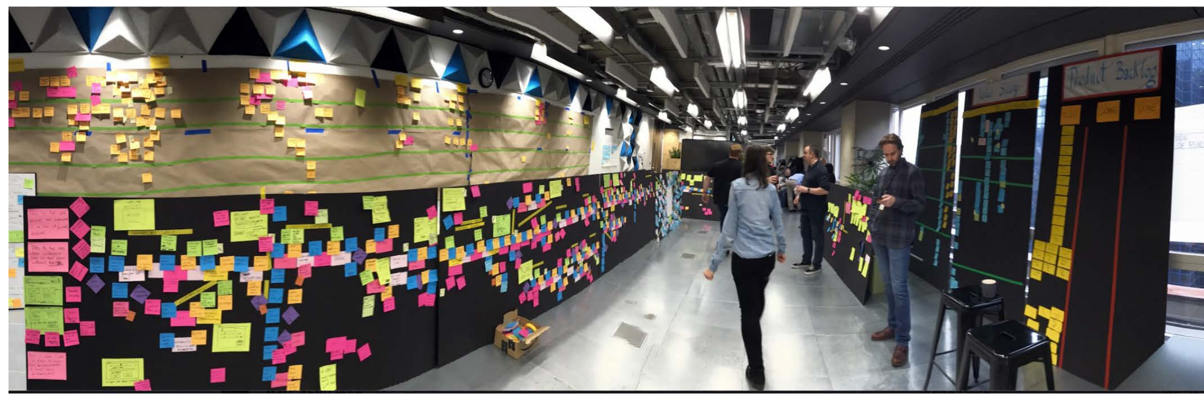


Figure 4.17: Event storming

2. **Visualizing what the team is working on now, what they've worked on previously, and what they might be working on in the future:** The following figure shows lots of boards of sticky notes providing the team with this information.

Figure 4.18: Information radiators on every wall and every window

3. **Our tools** (which we will introduce in *Chapter 6*, *Open Technical Practices – Beginnings*, *Starting Right*) can radiate lots of real-time information about the status and health of the software products we're building. Build status, test scores, and even animations showing the evolution of source code commits can provide helpful information to team members. The following figure shows some big monitors we use to radiate this information.

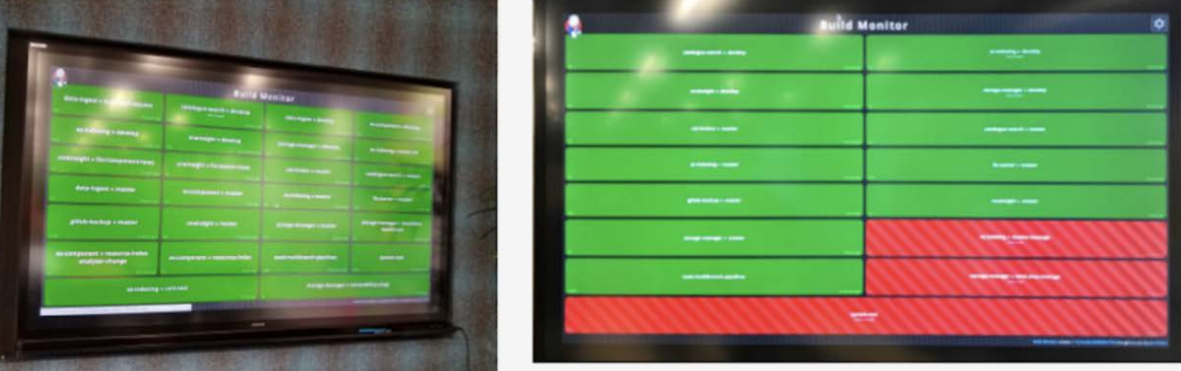


Figure 4.19: Real-time information radiators focused on technology

In the following section, we'll see how teams can practice this when working in distributed environments.

Radiating Everything When Distributed

Once again, it can be challenging to emulate this environment full of information radiators and artifacts when everyone in the team is in a different location. It is challenging but not impossible. Having some time and investment to set up workspaces well and equip them with the right tools and facilities will reap huge benefits and returns.

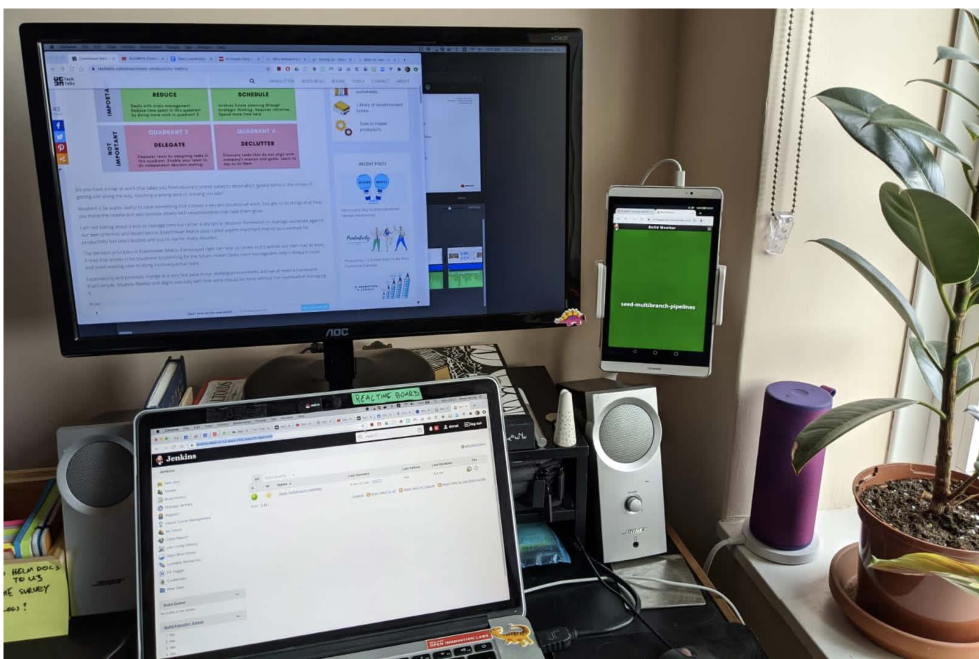


Figure 4.20: Setting up information radiators at home

Having a second (or even third) monitor can help have information radiators oozing warm information to you at all times. You can set windows up so you have different applications giving you real-time information and also have your digital whiteboard and video conferencing tools running. The benefit of using digital whiteboards is you will never run out of wall space!

In the preceding example, you'll also see a portable speaker (great if you have a team social playlist running!), a small tablet that is being used to show a live build monitor, wall space, and sticky notes – still useful for ideating and planning your own work. There's even a plant helping oxygenate the space to generate creativity and clearer thinking!

Check out many more tips (and add your own) by visiting https://openpracticelibrary.com/blog/guidelines-for-setting-up-a-home-working-space/.

A further set of information radiators we use to visualize a team's health and mood are supported by team sentiment practices and are a key enabler of high-performing teams.

Team Sentiment

Team sentiment practices track the mood of a team and provide a mechanism for team members to rapidly feed back, either publicly or anonymously, when their mood has changed. Team sentiment practices enable problems to be identified, radiated, and addressed early. They enable the team to quickly discuss where one or more team members are suddenly troubled by the direction their work is going in and can also provide an information radiator to assess the overall team's health and feeling.

One popular approach to introduce a team sentiment information radiator is mood marbles. To use this, you will need a number of containers to hold enough marbles so there is one for each team member participating. When we say containers, it can be a simple transparent box (in Red Hat, we love containers and OpenShift is all about containers, so we've even found a way to get containers to track our feelings!) or even just a drawing of a container on large flip-chart paper. You'll also need two different colored marbles (can be sticky notes) to start with. Other colors can be introduced.

Mood marble containers are placed in a strategic place where all team members will see and walk past them regularly (for example, near the door or on the way to the bathroom).

Team members are encouraged to think about their current mood. Start with positive (often this is green) and negative (often this is red), and each team member adds a marble to the container that reflects their mood.

Figure 4.21: Mood marble container radiating team sentiment information

Team members are regularly reminded they can change their mood marble at any time by removing their previous marble and replacing it with a different color. With a container of all team members' mood marbles, the information radiated is on the team's overall mood.

Blending Team Sentiment with Other Practices

Having a container of balls turn from green to red very quickly will cause more harm than good if we don't use other practices to recognize and address the change. The change could be explained by information on the real-time retrospective. Perhaps this is serious enough for Stop the World.

Either way, a retrospective is an excellent forum to surface, discuss, and address concerns that may be showing up in the team sentiment box. It can also be interesting to examine trends and see how team mood is changing over time. Team sentiment may also be impacted by other changes happening in the work environment or technology.

It can be interesting and very powerful to aggregate team sentiment data over time. We can use this to identify trends and patterns and draw conclusions as to how, for example, making some tweaks to technology practices, to other cultural practices, or to leadership behaviors can have an impact on team mood and performance. *Figure 4.22* illustrates mood distribution over time.

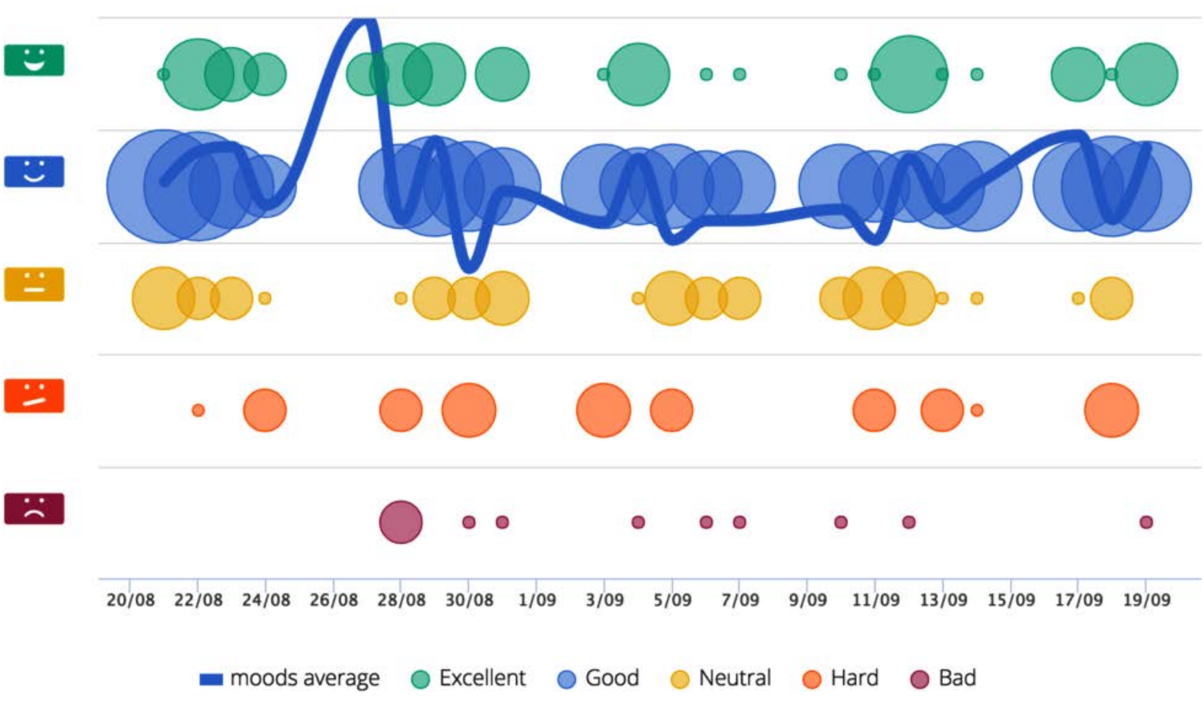


Figure 4.22: Team sentiment analysis

Let's look at another real-world story of one of our authors, where he learned that sometimes these practices can achieve different outcomes but still very helpful ones!

Team Sentiment Achieving a Different Purpose – Banter!

I have found that measuring team sentiment can be an extraordinarily tough thing to do in practice, especially when you are confronted with teams who are predominantly of one sex – for example, all male. I am a man and I can confidently say that men in particular generally do not want to share their *feelings*, especially in terms of the mood of the team.

On one particular engagement, we had a great female engagement lead who described the practice of mood marbles and was fully expecting the nearly all-male team to whole-heartedly participate. After several days of no activity on the mood marble board, the sticky note shown on the left appeared.

It took a bit of explaining that no, it really was not representative of the deeply held feelings of a frustrated teammate, it was just good old banter! In the days that followed, many different colors and funny drawings appeared in the mood marble container. So, the practice did (kind of) work!

Many people will have had to try team-forming with groups of people who have never met each other or are not in the same location. This is particularly true during the COVID-19 pandemic. Let's look at an example of how we used the same practice digitally.

Team Sentiment with Distributed People

The same practice can be done using a digital tool and again emphasizes the importance of keeping such information radiators warm and regularly inspected.

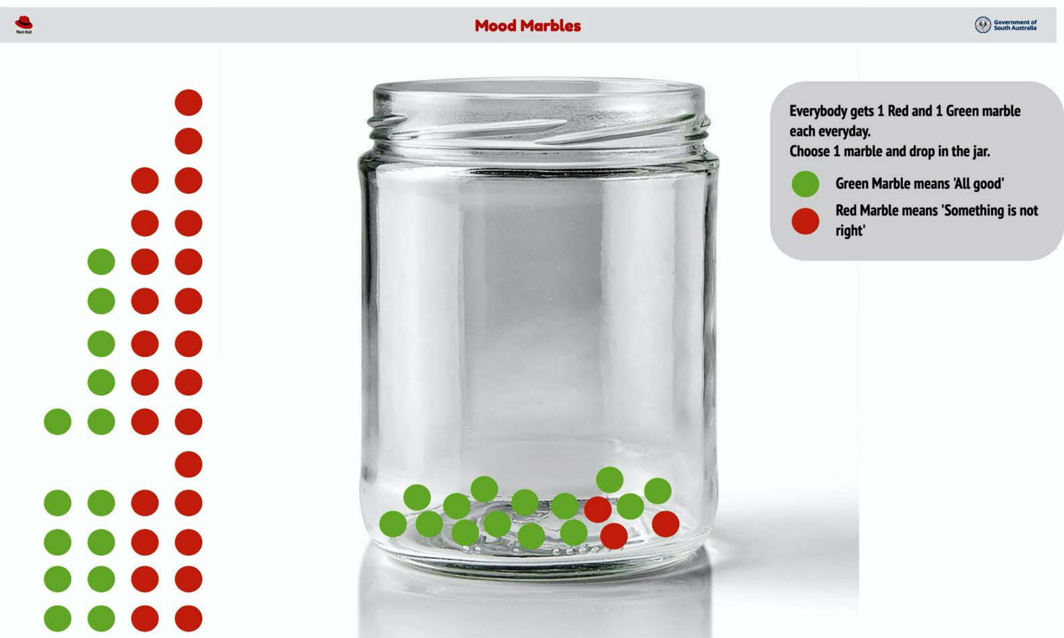


Figure 4.23: A digital mood marble container for distributed team sentiment capture

You can learn more and collaborate about Team Sentiment practices by going to the Open Practice Library page at https://openpracticelibrary.com/practice/team-sentiment/.

Radiate Failures

Having information radiators, dashboards, and mood marble containers show green can be satisfying and positive validation. But when they are red is when they are more powerful. When they radiate that something is wrong or failing, it presents a feedback loop to the people around to react and respond to.

Radiating Failure – as Useful (If Not More) as Radiating Success

Not all information radiators need to be flashing green all the time or show positive things. Sometimes the negative can be a great way to start a conversation.

When working for a retailer some years ago, there were multiple suppliers contributing to the solution. Each one was responsible for a different tier of the architecture and my team was responsible for some of the middleware used to serve data in a mobile-friendly way from a broker.

We were repeatedly being challenged by the retailer for being late in deploying our production server – however, none of the other teams was ready either! When we did deploy our stack, the broker layer was not connected up either, so our server was talking to something that did not exist!

To try to bring the conversation back to the teams that were now causing the blockages, we created a simple dashboard of TrafficLight jobs. It was a simple piece of automation that just did a smoke test of each service, to check its connectivity. We labeled the dashboard Production Monitoring and got it up on a projector in our project room – so whenever someone entered, they would see a board with lots of red on it.

The lesson I took from it was simple – one way to get things moving is to have a dashboard with lots of red on it labeled Production Monitoring! This dashboard helped us steer the conversation toward the fact that our piece of the architecture being in place on its own was useless – and that we as multiple suppliers for the retailer needed to work together to get the end-to-end solution working and not just our small parts.

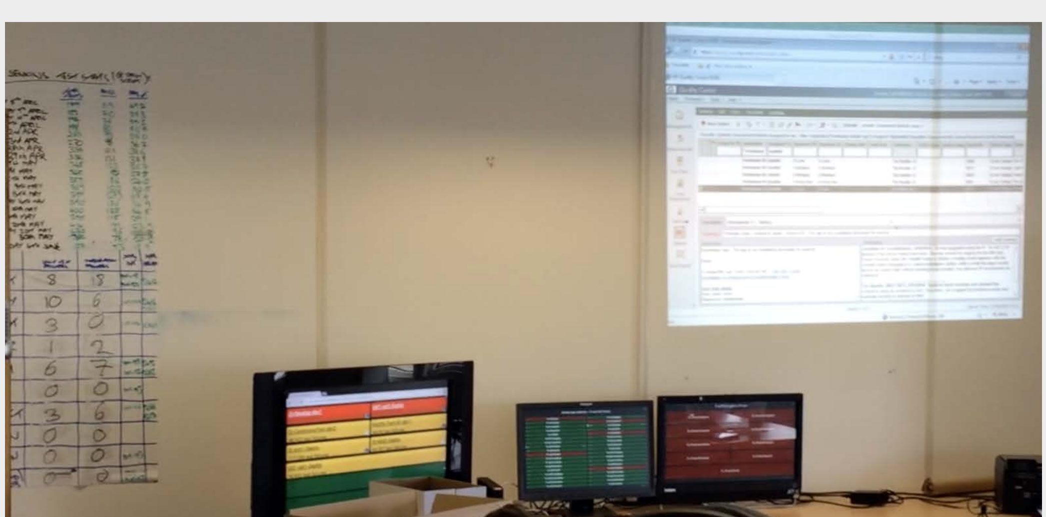


Figure 4.24: Starting to make technical environmental information visible

If you're looking for some great ideas for things you can visualize in a creative way, we cannot recommend this book enough – 96 *Visualization Examples* by Jimmy Janlén![6] It has some amazing ideas for displaying information that's critical for the team to be able to respond to. And it has a super bonus section on the correct way to peel a sticky note!!

You can learn more and collaborate about the visualization of work practice by going to the Open Practice Library page at https://openpracticelibrary.com/practice/visualisation-of-work/.

6 https://visualizationexamples.com/

Inspect and Adapt

By now we've introduced lots of practices and techniques to help the team collaborate and get to know each other. But there is no point in doing them once and walking away without learning from them.

There is nothing to be gained from the data they provide if it's not responded to. It's very easy to slip into this pattern. We've seen it all before – the team does a retrospective and comes up with a really great discussion but then captures no actions to fix things. Even worse, they capture actions but they are not adhered to or given any time to solve.

What happens next? The same problems appear in the very next retrospective and the team morale will slowly slip as nothing gets fixed. Time to break the cycle! Respond to the information. Be a doer.

You've got all these tools in place, but they're completely useless if ignored. They may look pretty – but they are useless if not actioned. We need to respond to the change when the tool or practice tells us that is the right thing to do, and we need to be prepared to pivot and change direction if that's what the information is telling us to do.

To be able to inspect something (technical), you have to monitor and continuously collect metrics as a fundamental tenet built into every service and every tool.

Metrics are built into OpenShift and other products. Insightful dashboards allow you to find things before your customers do. They also allow you to radiate real information about the product, technology, and team in a way that cannot be ignored. The data doesn't lie, and having it displayed so publicly and transparently can help unblock and maintain team autonomy and mastery. Where the team is blocked and radiating the information to explain why, holding regular *walk the walls* sessions with stakeholders and management who can make decisions to unblock and change the information can really drive inspection and adaptation for the wider good. If there is a problem, if there is a snag, if there is a blocker, make sure it is clearly being radiated and have the people who can fix it see that information regularly. If it's not being seen, make it bigger, share it more, add some sound – this is your opportunity to make sure these things are seen and heard.

PetBattle — Establishing the Cultural Foundation

Aidan, Ciarán, Emma, and Jen, the four engineers who have committed to work on the next evolution of PetBattle, met with Dave (a UX designer), Susan (a quality assurance consultant), and Eoin, who is leading the upcoming engagement to take PetBattle to the next level. Eoin had brought coffee and donuts and set the meeting up as a *kick off for success* session. Some of the team had worked together before. Jen, Dave, and Susan were brand new.

Eoin had rented some cool downtown office space that was going to be PetBattle HQ for the next few months. In the room, there were lots of movable desks, portable whiteboards, and swivel chairs, as well as bean bags and comfy foam seats. There were also some huge foam boards (a couple of inches thick and 8ft by 4ft in size) stacked up at one end of the room. All the furniture was pushed over to the other side of the room and Eoin pointed out that the team could configure the space however they wanted. Everything was on wheels and portable. Even the plants were on wheels!

There were also several brand-new large monitors and other IT accessories and loads of unopened boxes from Amazon that Eoin said were full of every color and every size of sticky note imaginable. He'd also bought Sharpie pens, timers, colored card, painters' tape – you name it, he'd bought it. Eoin had been busy!

On one wall, Eoin had prepared a *Kick-Off Backlog*, and there were a number of sticky notes representing items he wanted to cover in the next few days. The team looked at items on the backlog and added a few things of their own that they hoped to cover.

After coffee and a bit of chat, they started a short ice breaker where everyone had to introduce themselves and tell three things about themselves – two of them truthful and one a lie. Over the course of the week, the team would have to work out what everyone's lie was.

The next item was to form a social contract. Eoin explained how it worked and asked everyone to write at least one sticky note of something they would like to add. The social contract started with a discussion about the core working hours, which they agreed would be 10 AM to 4 PM – all collaborative activities would be scheduled inside this time box. Emma suggested a *Be on time* social contract item, which everyone agreed to. Dave suggested a *relaxed dress code*, which everyone was very happy with. During the next 30 minutes, the team got into some interesting conversations that resulted in the social contract comprising the following as well:

- Have fun!
- Ensure everyone is heard
- Show empathy
- It's ok to be wrong
- Mob to learn and pair to build
- Be open
- Have a social evening out every week
- Breakfast and lunch as a team three times a week
- Be empowered

Eventually, the team agreed this was a good start but they would all continue to reflect and think about how to improve the social contract. They all signed it and Eoin hung it on the wall near the team.

Jen suggested the team try using a real-time retrospective – something she had seen at a recent conference and was keen to try. Everyone had done retrospectives before but they were not familiar with the real-time aspect. It seemed like a good idea and they agreed to give it a go.

The next items on the backlog were *introduction to the room*. Eoin had set up a mood marble container by the door and explained how the practice would work. The team looked a bit skeptical (Aidan made a comment that these were not the kind of containers he thought he'd be working with today) but, in the spirit of being open and experimental, would give it a go.

At the other end of the room was a big rope with an old bell at the top. There was a big sign next to it that said STOP. Susan offered to explain how this worked and how it could be used to *stop the world*. The team seemed very happy with this – it made a lot of sense. In fact, most of them were thinking of many scenarios that had happened to them in previous jobs where this would have been very useful!

The team was introduced to the rest of the room. There were lots of placeholder signs on different parts of the wall. One had now been filled with the social contract. There were headings for Impact Map, Event Storm, value slices, product backlog, and sprint 1. Eoin explained that, while the room may feel empty now, in just a few days it will be filled with information.

The next item on the kick-off backlog was to configure the space. This was a great activity to do as a team and was really in the spirit of being empowered. The team wheeled in tables and discussed how they'd like to be set up with a mobbing corner and pairing stations. They set up every monitor Eoin had purchased. They even attached some Raspberry Pi mini-computers to each one. Aidan added a PetBattle digital sign to each one! The music was on, the space was configured, and it looked great. The volume of the team over lunch started to rise. There were more laughs and banter.

The team had started to form. The cultural foundation was in place and would only strengthen from there.

Conclusion

In this chapter, we introduced the foundation from a culture and collaboration perspective and the importance of building this foundation before we do any technology or try to do any discovery or delivery of product development. To achieve strong business outcomes when using DevOps and OpenShift, we need to have high-performing teams developing and operating software products.

The starting point to creating a high-performing team is enabling an open culture where the team members feel psychologically safe and can achieve a sense of autonomy, mastery, and purpose in everything they do.

To help us realize this and build the foundation, we explored several of our favorite practices that have enjoyed the most success in kicking off new teams – social contracts, a *Stop the World* system, gaining team identity, radiating as much information as possible, including team sentiment, and starting a cycle of regular inspection and adaptation.

We told a fair number of stories and shared some memories of applying these practices. And we returned to our newly forming PetBattle team to see how they went about starting their cultural foundation on their first day together.

Some of the practices in this chapter may work for you and your team. Some may not and you'll want to throw the tool back into the box and choose something else. There are many, many more you can explore at https://openpracticelibrary.com/tags/foundation, or you can contribute your own. Remember, the precise practices and tools you use are not as important as the fact that you are investing time in establishing a cultural foundation.

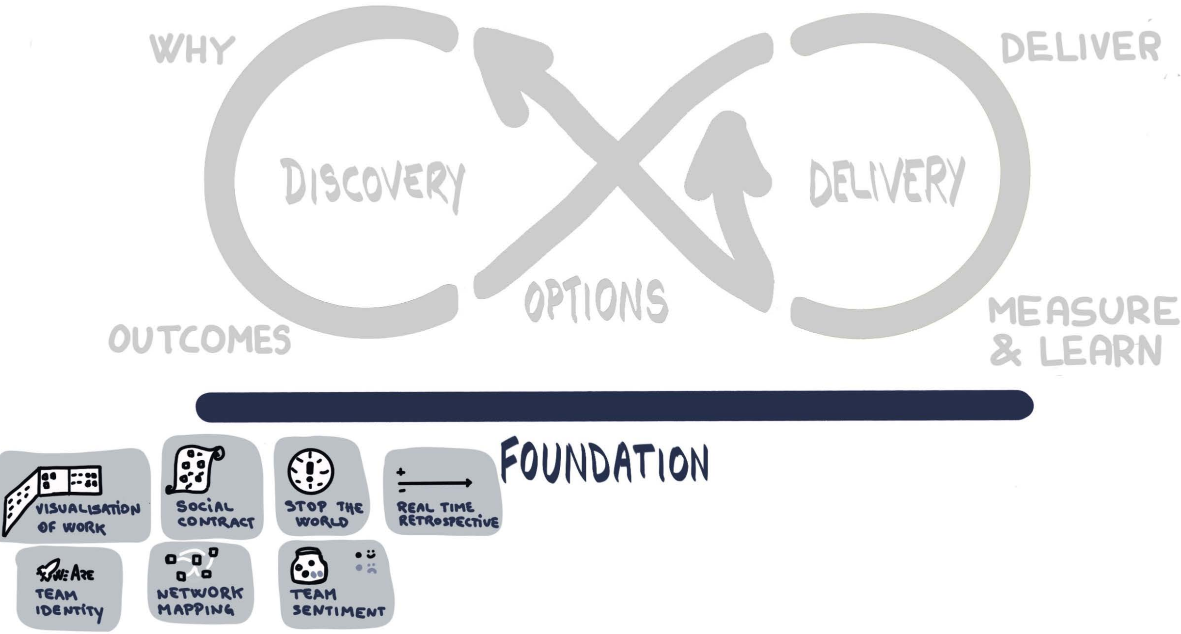


Figure 4.25: Adding the foundation of open culture practices

Of course, a foundation can be improved upon and strengthened. In the next chapter, we'll explore how leadership can help this from the outset and how open physical environments and spaces can strengthen it. In *Section 7*, *Improve It*, *Sustain It*, we'll explore further how the foundation can be strengthened to support scalability and sustainability for product teams adopting the open way of working.

5

Open Environment and Open Leadership

In the previous chapter, we explored what it means to have an open culture and how we can enable teams to build this into their way of working from the ground up.

When we talk about having empowered, self-organizing, self-directing teams, many enthusiasts will say that management is supposed to just get out of the way and disappear! Some will say if teams have true empowerment, surely they don't need to be managed and they don't need managers.

Differentiating between leadership and management is important here. We want teams to manage the organization themselves from the bottom up and for leaders to set direction and intent that enables that behavior throughout the organization. This is how open organizations such as Red Hat are led.

Jim Whitehurst, former CEO of Red Hat, defines an open organization as *an organization that engages participative communities both inside and out – responds to opportunities more quickly, has access to resources and talent outside the organization, and inspires, motivates, and empowers people at all levels to act with accountability*.

His diagram below articulates the difference between conventional organizations that are managed *top down* and open organizations that are led and organized from the bottom up.

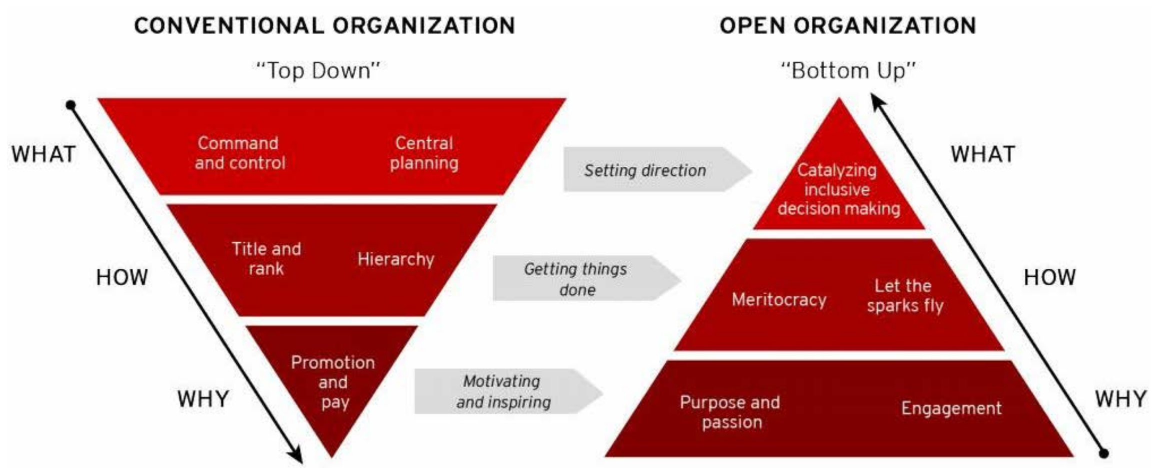


Figure 5.1: Differences between conventional and open organizations

So, what is leadership's main role in all of this? What can leadership do that helps deliver lasting change within a learning organization? We are going to tackle these questions by first taking a look at the Kodak case study, the company that invented personal photography but ultimately missed the digital revolution. What lessons can we learn from it? In the *Changing an organization* section, we'll discover that leadership can achieve far better execution by moving the decision-making to where the information is. We'll then learn how to use Priority sliders and forced ranking to help us prioritize where we can focus our efforts. In the final *The space* section, we'll take a look at creating the right physical environment for our teams to flourish within.

The Kodak Problem

You don't have to be a CEO or a business management consultant to understand that there are lessons everyone can learn just by sharing stories about different organizations and how they succeed or fail in a rapidly changing world. The lifespan of public companies has decreased markedly in the past 50 or so years.[1] Why is this? History is littered with examples of organizations that have failed to adapt to changing customer needs. Let's take a look at possibly one of the most well-known of these stories in more detail and understand what went wrong.

The Eastman Kodak company invented personal photography. Up until the turn of the twentieth century, to take a photo, you had to go into a studio and have someone take your photo. Kodak sold this awesome user experience in a *box* for $1 – not a lot, really. The real money was to be made in processing the film to produce the pictures, which usually took a week or two. By the end of the twentieth century, Kodak had been hugely successful as a company and a brand.

However, they missed a trick – by the turn of the twenty-first century, digital photography had started to take off. Kodak was there in digital, but they believed that photography was a *chemical process* thing – the money was made historically from chemically processing the film. By the time they realized the market had shifted, it was all too late for them. Rising tech giants in Japan and Asia were far better at making digital tech than they could ever be. Kodak filed for Chapter 11 bankruptcy in January 2012.

Did you know that a man named Steve Sasson was the inventor of the Charged Couple Device, which formed the basis of another invention of his – the digital camera. Guess what – he worked for Kodak!

1 https://www.amazon.com/Creative-Destruction-Underperform-Market-Successfully/dp/038550134X

That's right. He invented the digital camera in 1975 and his bosses at the time buried it – they said no one would ever want to look at their photos on a television set. Steve also invented the first Digital Single Lens Camera in 1989, but again, at the time, Kodak marketing thought that releasing the product would interfere too much with their film processing business – so it was buried as well.

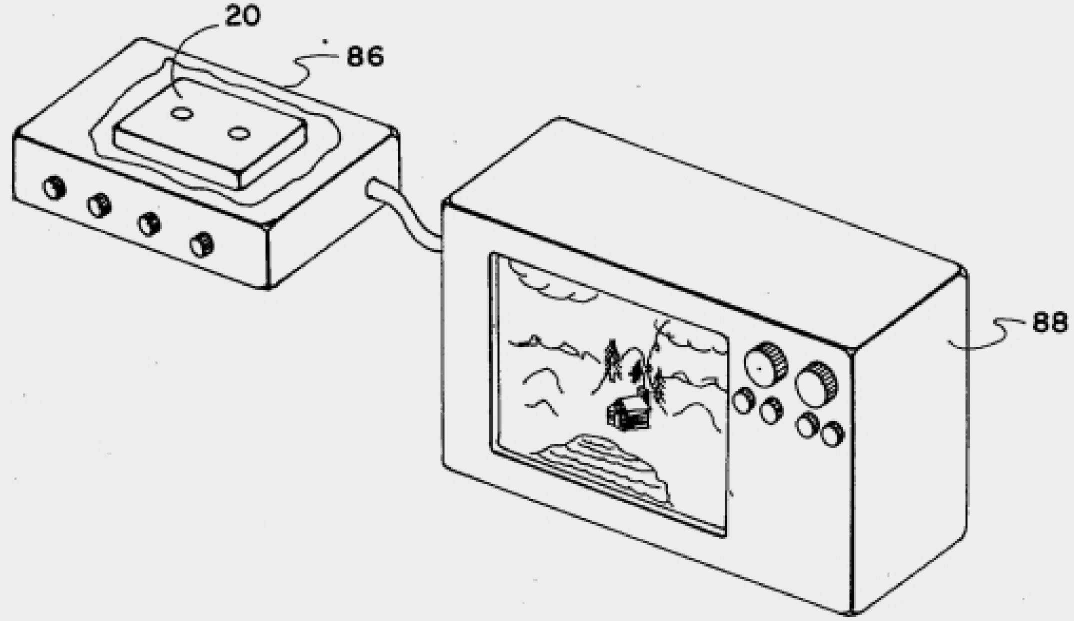


Figure 5.2: Digital Camera patent image from 1976

Now, Kodak made billions on their digital patents; however, these expired in 2007. The company just could not pivot fast enough to change the customer experience, which their competitors were delivering through digital, and eventually filed for bankruptcy protection in 2012.

This provides a fine example of the need for continuous learning, continuous measures, and a continuous appetite to pivot.

Learning from History

There are many lessons we can learn from the Kodak story:

- Organizations must adapt to their changing customer needs.

  This is easier said than done. Human understanding and cognition are heavily influenced by our cultural beliefs and upbringing. We just don't see what others see. In some ways, this is what happened at Kodak. Their background and company history were from a chemical engineering viewpoint; indeed, Eastman Chemicals (which was spun off from Kodak) is still a very successful company today. At the time, Kodak's management was incapable of seeing the transformative change to the user experience that digital photography represented.

- The Kodak story shows us that innovative change can come from anywhere within an organization.

It often requires a different point of view from leadership for internal views to be promoted successfully as a business strategy. The old command and control structures themselves need to change so they are better able to encourage and embrace innovative change.

Open Leadership

Shabnoor Shah, Open Leadership Global Lead and Coach for Open Innovation Labs, explained to us how Open Leadership is a new refreshing gaming-changing way of leading in a digitally transforming world. The foundations of leadership are rooted in the open-source way of thinking, working, and behaving.

A unique aspect of open leadership is that, because it is really a mindset and a way of being, it is not restricted to top levels of management in the hierarchy. Open Leadership can be practiced by anyone and at any level in the organization. However, when leaders lead openly, the impact is significant and palpable in shaping an open, positive, and progressive organizational culture. The results are reflected in employee happiness, well-being and engagement, customer satisfaction, and overall profitability and success of your organization.

The guiding principles of Open leadership and open organizations are transparency, inclusivity, collaboration, community and participation, adaptability, meritocracy, and releasing early and often. The overarching belief of the open leadership mindset (at Red Hat) is to *default to open because open is a better way*. This is supported by four supporting beliefs that everyone has something to contribute, everyone has untapped potential, everyone has the responsibility to lead and everyone benefits when we (all) put the organization first.

Changing an Organization

Organizational charts are usually drawn top-down in a hierarchy, and they tell you nothing about the nature of the company or its interactions. Let's redraw our organizational chart to see if we can better represent the interactions that might be occurring. Our customers are drawn as our roots, the foundations that allow the whole organization to survive. Next, we have the different business units drawn as petals that interact using business processes to achieve their goals. The company president is a rain cloud who is downward-looking, shielding the organization from the external board and stakeholders, who are represented by the outward-looking face of the company, the CEO.

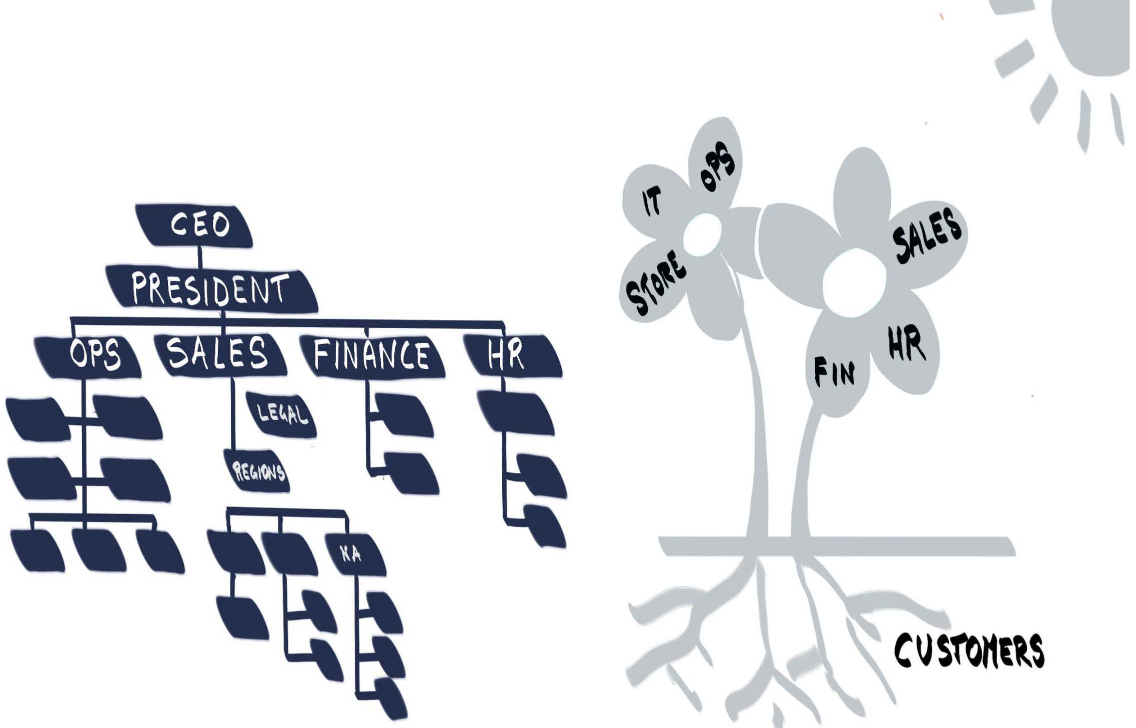


Figure 5.3: Organizational charts rethought

Humans are biological organisms. Organizations should also be treated like organisms rather than the product of their organizational chart. It makes sense – organizations are the product of the people they employ. So, when looking at how organizations change, we should ask the question *how does change occur with humans*? If the majority of people find change unsettling, what makes a person go through change rather than accepting the status quo?

People make changes when they are insecure, unhappy, or unsatisfied with their current situation. Similarly, companies that are under stress are going to be forced to implement change to survive. It is insightful that when putting into action the practices described within this book, that it will feel *uncomfortable*. Often, team members will describe the changes as *hard* and *difficult* to make. This is actually a good thing! Change is hard. Companies (just like humans) need to embrace creativity to be innovative – so they can deliver awesome experiences to their paying customers. A successful transformation isn't something that happens for only a focused period of time, with no changes happening after that. Embracing innovative change as a sustained way of operating is the challenge.

Leading Sustainable Change

Change within an organization is not sustainable without funding and executive sponsorship. Kodak failed to foster and action the great ideas that came from within. Within an organization, it is the organizational leaders that make the rules of the game that everyone else must follow. Importantly, leaders set the environment and the context that teams must operate within. This is akin to the *farming analogy*. To be a successful farmer, you must create an environment for your crops to grow. Organization leaders must create the right conditions to make innovative transformation flourish and become the norm. In a talk, we recently heard *Simon Sinek*, a British-American author and inspirational speaker, say this – *There is one fact that is not disputable – your customers, stakeholders, and employees are all humans. You cannot lead a company. You can lead people.*

It is often unclear that leaders must take risks to change the current way of working. Often the risk is also a career risk for leadership. They are putting their job on the line to embrace the change. So, it is important that the first or initial scope of improvement decisions is made carefully. The change cannot be so big that it risks organizational failure, and must be meaningful so that customers and stakeholders feel a real business impact – your product gets *oohs* and *aahs* when it is delivered.

Does any of this sound familiar in organizations you have worked in?

- We have many unhappy silos within the organization.
- We are one team with many clans.
- There are too many hand-offs.
- There is too little accountability.
- We have misaligned incentives.
- We are punished for failures.
- We have people with egos who don't want to share.
- There are barriers to change – this is the way we have always done it.

If you identified with any of these, they are all traits and characteristics that require executive sponsorship to change. It is the leadership's role to:

- Create a shared purpose and vision.
- Give permission to *Change the Rules*.
- Remove unnecessary gatekeepers and roadblocks.
- Encourage actions that *Get Work Done*.
- Decentralize decision-making.
- Demonstrate transparent communication.
- Break down barriers between silos and departments.
- Measure the system as a whole.
- Make *Transformation* everyone's job.
- Value results based on delivering organizational outcomes over process adherence.
- Help build an engineering sanctuary where trust exists between the business and technology.
- Demonstrate results with metrics and data.
- Encourage teams to own the full engineering stack.
- Build transparency into the full development process.
- Take the risk to start the first transformational change.

This is a long list. In the next section, we'll tackle three ways of bringing about these traits by taking some advice from a submarine captain!

Achieving Greatness

In Captain David Marquet's book, *Turn the Ship Around*,[2] he defines leadership as *embedding the capacity for greatness in the people and practices of an organization, and decoupling it from the personality.*

Giving Intent

In it, he talks about his experience of being a captain of a nuclear submarine and how they learned not to follow the leader into disastrous situations. He vowed never to give another direct order but, instead, to set intent. Rather than giving instructions, give intent. Officers stopped requesting permission all the time, which meant psychological ownership shifted towards them. Marquet talks about the two pillars that supported this idea of giving control – technical competence and organizational clarity.

Moving Decisions to Where the Information Is

Moving the authority to make decisions to where the information is means, in the software space, software engineers can decide when to ship software and to do so when ready. The best time to make a decision is the last responsible moment before it must be made. By delaying, we are afforded the most opportunities to gather as much information as possible. By doing this, you get faster and better-quality execution of decisions than if central leadership were to make them.

Setting the Environment

This can be a huge mind shift to many but, when reading Marquet's book and watching the ten-minute *Greatness* video on YouTube[3] (another video we love to show our customers and, in particular, their leadership), it makes a lot of sense. This mindset provides the leadership foundation for at least one team to work autonomously and with purpose. We'll explore the scalability of this mindset later in the book.

---

2 https://davidmarquet.com/turn-the-ship-around-book/

3 https://youtu.be/OqmdLcyES_Q

Figure 5.4 represents the sketch produced during the ten-minute video and has several important messages.

Figure 5.4: Inno-Versity Presents: "Greatness" by David Marquet

If you haven't yet watched the video that produced the graphic above, please take the time to do it now. It is inspiring, thought providing and sets the leadership context for the rest of this book.

How Do We (as Leaders) Convince the Doubters?

Every company is an IT company, regardless of what business they think they're in – Christopher Little, Software Executive, DevOps Chronicler

If you are not in IT, it is fair to say that IT is often viewed as *getting in the way* of an organization's business goals. As Steve, the CEO from the *Phoenix Project* book by Gene Kim[4] says – *IT Matters. IT is not just a department that I can delegate away. IT is smack in the middle of every major company effort we have and it is critical to almost every aspect of daily operations.* Often it is the non-IT part of an organization that is the hardest to get on board with a new way of working that puts them front and center with the IT department to help drive outcomes of increasing customer success.

4 The Phoenix Project, A Novel About IT and Helping Your Business Win – Gene Kim, Kevin Behr, George Spafford

No Computers in the Company! The 1990s or the 1890s?

In the 90s, I was working at a software house, writing trading systems for banks in London. My partner was working at a traditional London law firm as part of the accounts team. She told me a story that I couldn't quite believe: there were NO computers in the company. All business was transacted on paper and B2B was still done by a fax machine. I was astounded that it was even possible to do business without IT!

Can you think of any similar examples today? No, me neither.

A business product owner and business SMEs are key people supporting our cross-functional teams. One of their key tasks is to represent and communicate with all of the product stakeholders and customers and share those conversations and outcomes with the team. By helping the team decide on what should, and more importantly should not, be delivered, the product owner vastly improves customer satisfaction because the software that represents the most customer value can be worked on and delivered first.

By being part of the cross-functional team, product owners gain a shared understanding of the technical challenges involved in writing and operating the applications that deliver business services to their customers.

Ultimately, it is through this collaboration and shared understanding between technical and non-technical team members that we get business leaders on board with IT. By successfully delivering customer and organizational outcomes, the product owner can show success to their management and in so doing, own the IT challenges with their technology counterparts.

Now that leadership is enabling and supporting our product team's efforts, we are going to switch gears next and take a look at the practice of using priority sliders, which can help our team prioritize what product areas or team building topics are most important.

Priority Sliders

Priority sliders are a great, simple tool to have in your kit bag! Like most of the practices we're exploring, they're really just a tool to help us facilitate a conversation and drive some shared understanding. We use them to drive team consensus in the direction we should go for a given time length.

Running the practice is easy. Just get a small area of a whiteboard and do some brainstorming around key topics of focus for your engagement. They could be things like:

- **Functional Completeness**: How important is being 100% complete on some piece of app functionality, or are we looking for some sort of thin thread through all the functional areas?
- **Security**: We know security is important but how much time do we want to invest now in hardening our software?
- **Skills Acquisition**: Happy, motivated individuals make great teams. Making sure the team has all the expertise to build, operate, and own their software could be important.
- **User Experience**: Are we building what we want or what we think our customers want?
- **Test Completeness**: There will always be some testing. How important is it for us to automate this right now or should we focus on test automation from the beginning?

It is important to call out that these topics are not product features. They are product areas or team-building topics, without which we would not get a great product. For example, perhaps you want to drive more sales through your application so you decide to implement some form of push notifications to drive your customers directly to your app. Push notifications are not something you would add to your priority sliders, but perhaps market penetration could be. This topic could include a raft of additional features or experiments you could run. It's good to have a few examples of these ready and the team should always ensure they know what is meant by each item on the slider before you begin.

With your list in place, get yourself some Post-Its and Sharpies and write them up in one column. For example, let's say we use the five headings from above. With five items, we now need a scale numbered from one to five for each one. The more items your team is prioritizing against, the higher your scale will go.

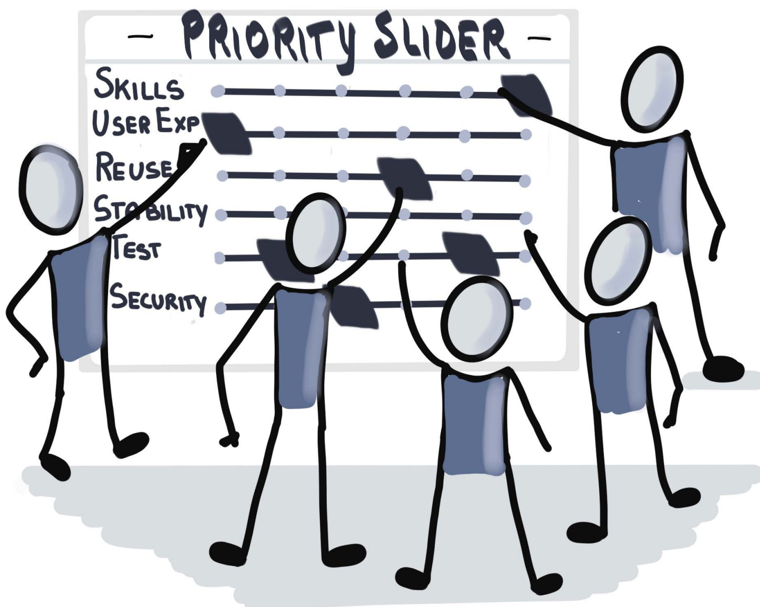


Figure 5.5: Priority sliders

With the scale in place, the team needs to decide what they think is the most important item relative to the other items. The key with priority sliders is that no two items can take up the same priority – so if 5 is your top, then User Experience and Security cannot be on the same level. Each item must be traded off against another – hence some teams call this practice as using *trade-off sliders*.

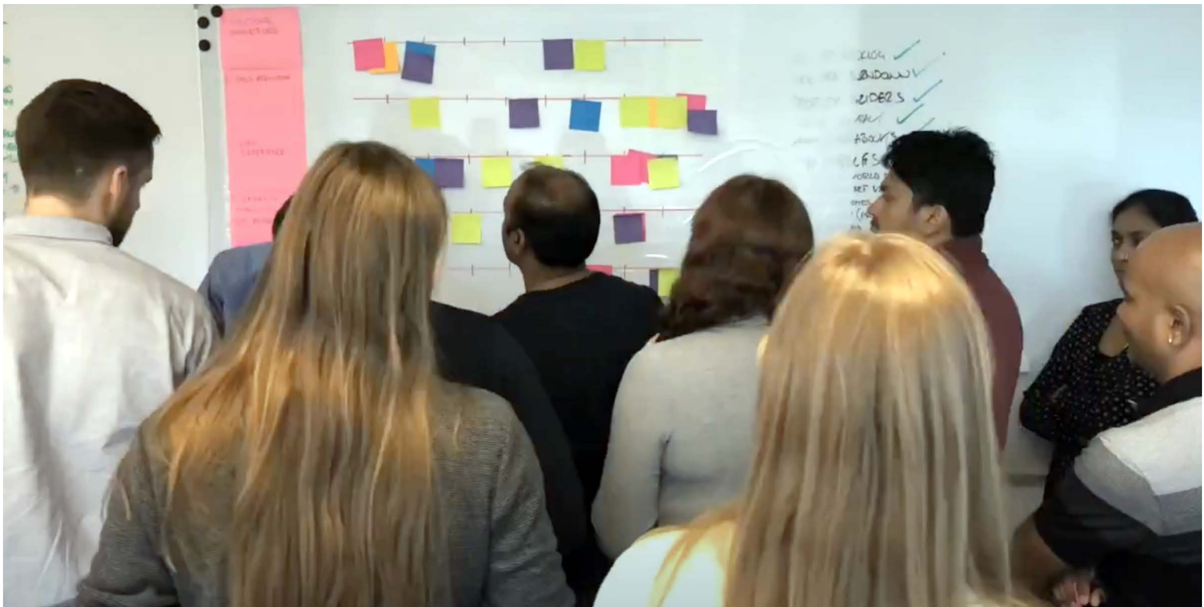


Figure 5.6: Priority sliders – collaboration in action

In the picture above, we can see a team discussing a priority sliders session in progress. Trying to figure out as a team how to force rank the options leads to a lot of great discussions. Below we can see two complete sets of sliders. It's important that everyone agrees on whether 1 or 10 is the highest importance!

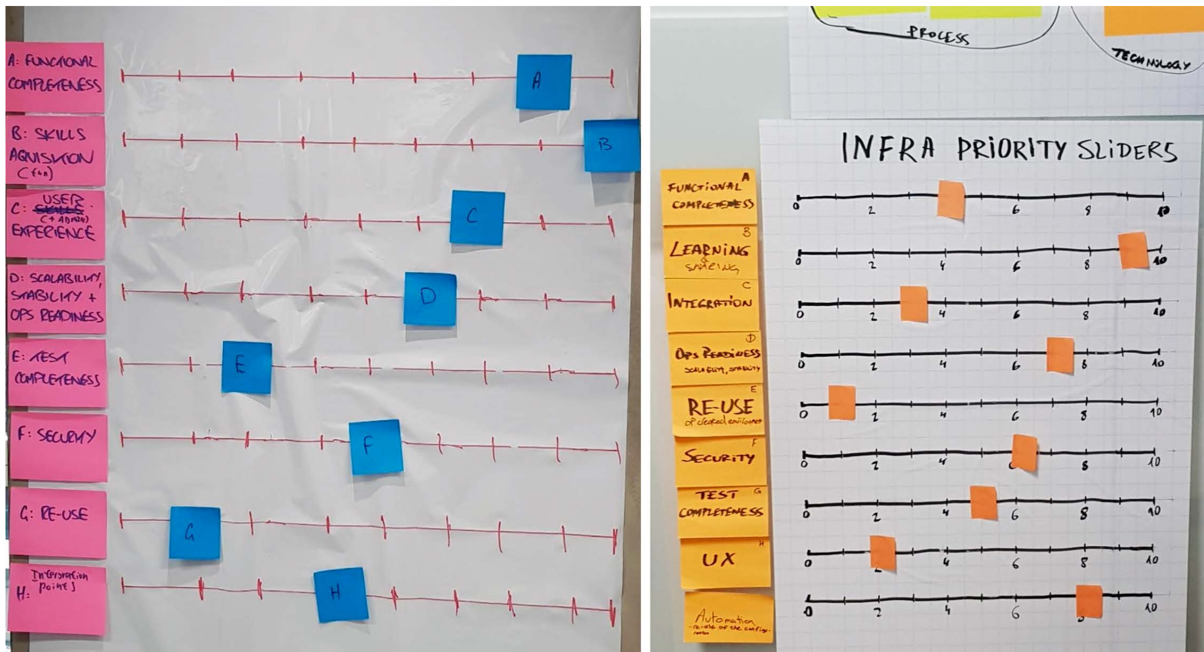
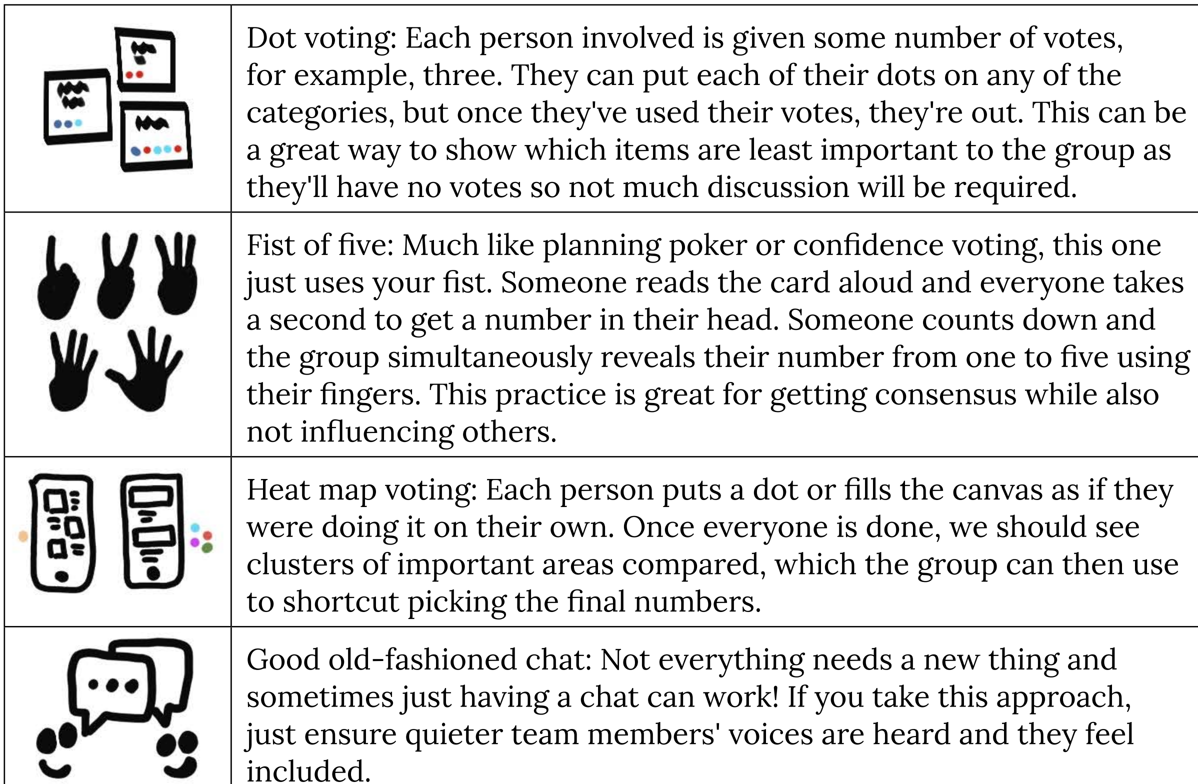


Figure 5.7: Priority sliders examples

There are many ways for a group to gain consensus – they could just have a good old-fashioned chat about each item and agree one by one as they go down the list. This can work well for some groups but for others, a more formal approach can ensure all people participate. Here are a few ideas:



	Dot voting: Each person involved is given some number of votes, for example, three. They can put each of their dots on any of the categories, but once they've used their votes, they're out. This can be a great way to show which items are least important to the group as they'll have no votes so not much discussion will be required.
	Fist of five: Much like planning poker or confidence voting, this one just uses your fist. Someone reads the card aloud and everyone takes a second to get a number in their head. Someone counts down and the group simultaneously reveals their number from one to five using their fingers. This practice is great for getting consensus while also not influencing others.
	Heat map voting: Each person puts a dot or fills the canvas as if they were doing it on their own. Once everyone is done, we should see clusters of important areas compared, which the group can then use to shortcut picking the final numbers.
	Good old-fashioned chat: Not everything needs a new thing and sometimes just having a chat can work! If you take this approach, just ensure quieter team members' voices are heard and they feel included.

Table 5.1: Methods for encouraging stakeholder participation

With the completed canvas, we can get a sense of our upcoming priorities. These could be set for a few weeks or for longer but like all practices, this is not a one-time thing! It should be returned to once some work has been done in any given area on the canvas, checking whether it is enough and whether our priority has changed.

Great product owners and scrum masters could take the priority sliders with them to sprint planning events. When deciding what to bring into sprint or what to omit, being able to shortcut those decisions by referring to the canvas can save time. Like all these practices, it's very important to keep them visual at all times and, of course, this is much easier when everyone can be in the same room. Of course, there are still excellent ways to run a priority sliders session with a distributed team, which we'll look at next.

Running Priority Sliders with Distributed People

If you have a good digital whiteboard tool, and all stakeholders needed in the priority sliders are connected via a video conferencing tool, running the practice is relatively easy. We have provided a useful template that you can download from the book's GitHub repository and use in your tool of choice such as Miro, Mural, PowerPoint, Google Slides, etc.

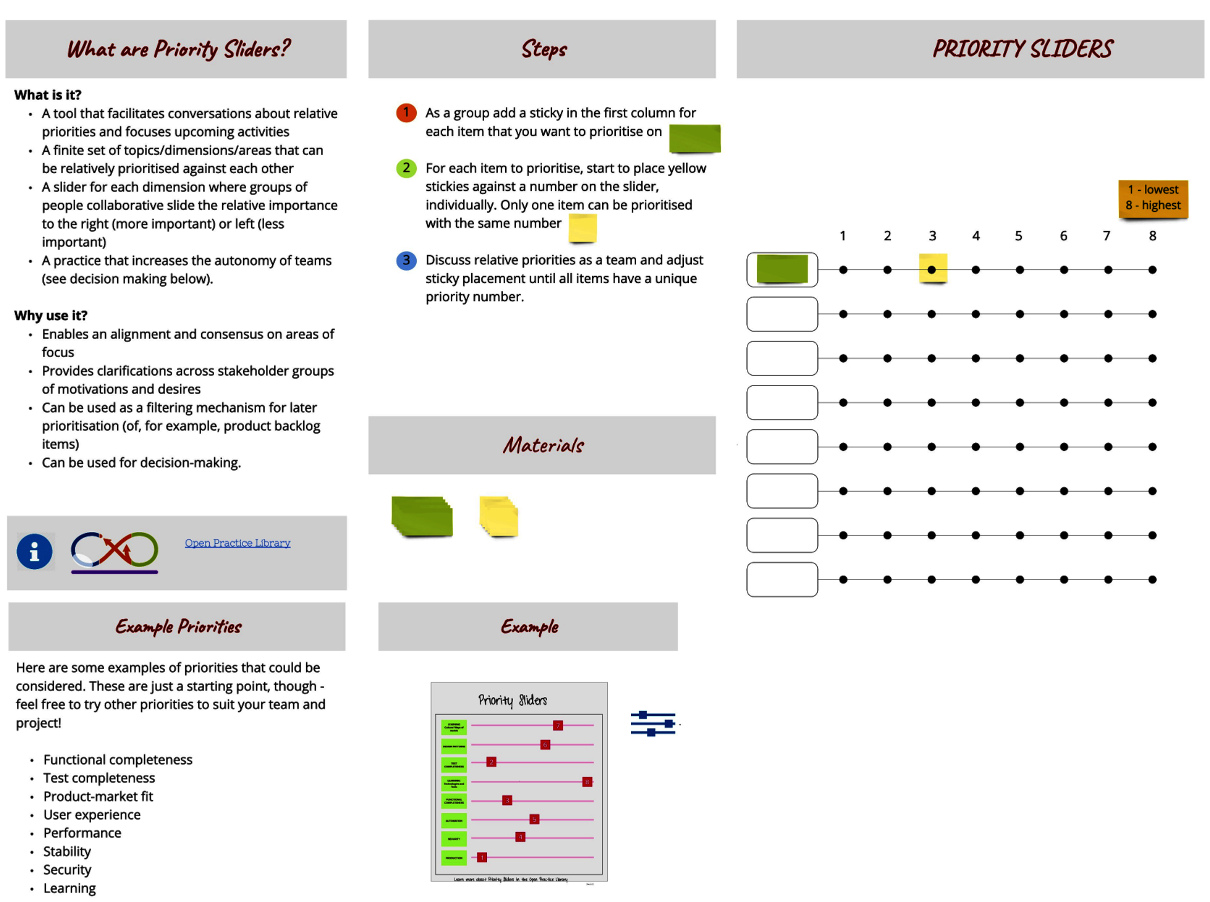


Figure 5.8: Digital priority slider canvas for use by distributed people

When there are sliders on the wall and everyone's on their feet and having to add their dot, the environment is so well set up for conversation and collaboration, people can't help themselves! On a video call, it's too easy for some folks to disappear and not get involved. So, really strong facilitation is needed. Adopting a liberating structure such as **1-2-4-all** will help get engagement from the outset. 1-2-4-all is a simple facilitation technique whereby individuals are first asked to independently and privately provide their input or opinion. They then pair up with one other person to discuss each of their ideas and merge them. Then two pairs group together to converge each of their ideas before the full group gathers together to bring all the input together.

You can learn more and collaborate on the priority sliders practice by going to the Open Practice Library page at openpracticelibrary.com/practice/priority-sliders.

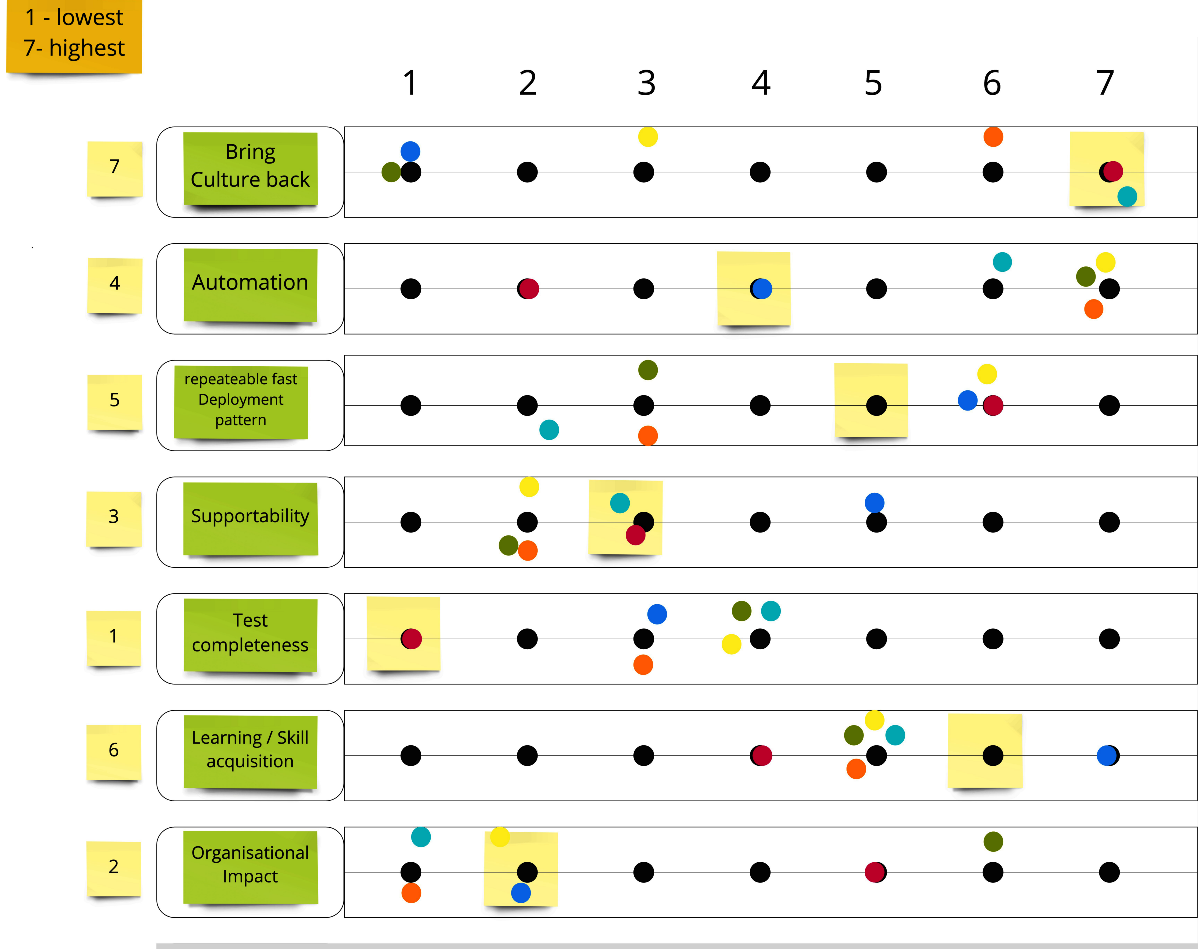


Figure 5.9: Digital priority slider example used distributed people

Regardless of whether we are in a physical or virtual working space, the practices introduced in this book so far (including priority sliders) all need a good space to be organized for the team to collaborate.

The Space

One of the big focus areas that can really help leaders establish a strong foundation for their team is in finding a suitable space for team members to work.

In the previous chapter, we explored the importance of motivation and autonomy, mastery, and purpose. The physical working space is a great enabler. Great teams work in great spaces.

When we run our Open Innovation Labs residencies, we do them in one of our own physical labs, which was purpose-built for this type of working. It is often the number one concern our customers' residents have – they say their own buildings are not set up for this type of work and there will be many blockers put up by Facilities, Security, Health and Safety, and other departments. We aim to use the time in the labs to *show* leaders and stakeholders how much engagement, energy, and value is delivered from this way of working and how much the physical space enables it.

We have some space recommendations based on a collection of best practices and lessons learned from real facilities built out of global, permanent Open Innovation Labs facilities, as well as experiences building temporary, *pop-up* team spaces. *Val Yonchev, Open Innovation Labs EMEA Leader*, and *Mike Walker, Global Senior Director of Open Innovation Labs*, provided much of this thought leadership in their own contribution to the Open Practice Library, which you can read at openpracticelibrary.com/practice/team-spaces/.

Let's look at a few examples. First, a lab space such as that in the next picture is very open. There are no barriers or wall dividers. It is highly configurable with all the tables, chairs, monitors, and even plants on wheels!

Figure 5.10: Open workspace example

Open workspaces are well lit, ideally with some natural sunlight, with tables and chairs well distributed around the room. There is a lot of wall space for all the information radiators.

Figure 5.11: Open workspace with lots of wall space

Information radiators should be easily accessible and in the line of sight of team members. There's little value in having all this information if the team is not going to be seeing it regularly.

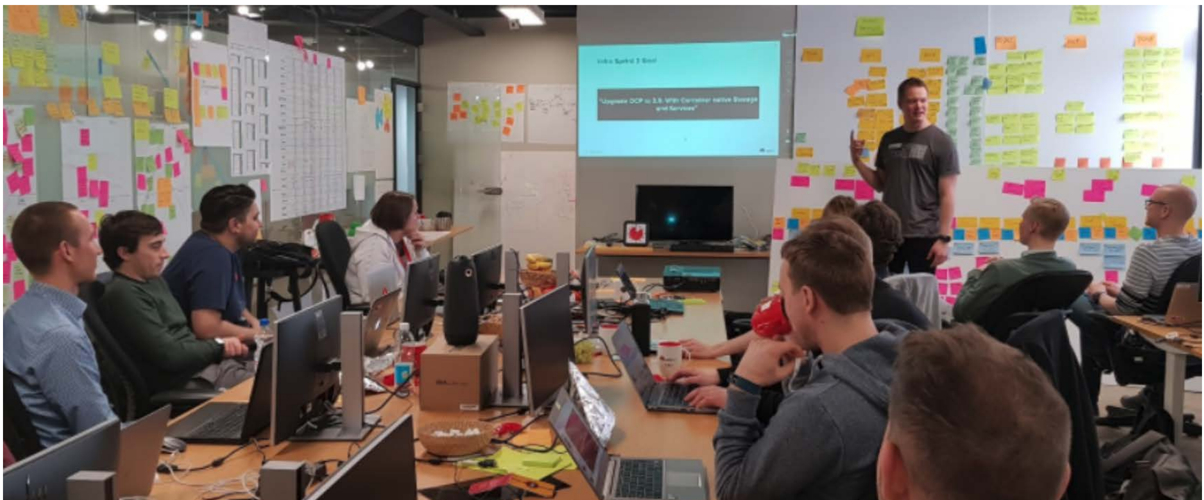


Figure 5.12: Examples of the visualization of work

Most importantly, the space should enable conversation and collaboration to happen as much as possible.

Figure 5.13: Configurable workspace set up for pairing and mobbing collaboration

Just how much do we need to invest in the space and what is most important?

The Minimal Viable Space

A question we often get is do we need to invest tens of thousands of dollars in a space and have all the things outlined in, for example, the Open Practice Library article in place before we can get our teams working in them? The answer is no. As much as we'd love all teams to have the perfect space, the reality is there are lots of barriers and challenges to getting there. Taking an agile approach to incrementally building out team spaces and applying continuous improvement can work just as well and can involve the teams in their construction.

The minimal viable space for a team is some open space without divides and some wall space. We've even got around wall space concerns with some of our customers by buying boxes of 25 foam boards (4ft by 8ft and 5mm thick). These are very portable and can stand against furniture, walls, windows, hang from ceilings on magnets, and so on. We like to call them portable walls!

How have our customers reacted to the idea of bringing in huge foam boards? Let's look at a recent example.

"We See What You Want To Do and Why and We'll Help You Get There" in Just 4 Weeks

I led a lab residency in a pop-up space in the offices of our customer, who was an international oil and gas customer. Understandably, security and health and safety were of utmost importance, and we realized getting custom furniture and movable boards was not going to be quickly approved.

So, we took the approach of only having portable equipment. We brought in the foam boards mentioned above, several rolls of static magic whiteboard, boxes of sticky notes (which would only be stuck to our boards or magic whiteboard), and a projector.

This meant we turned the space from this:

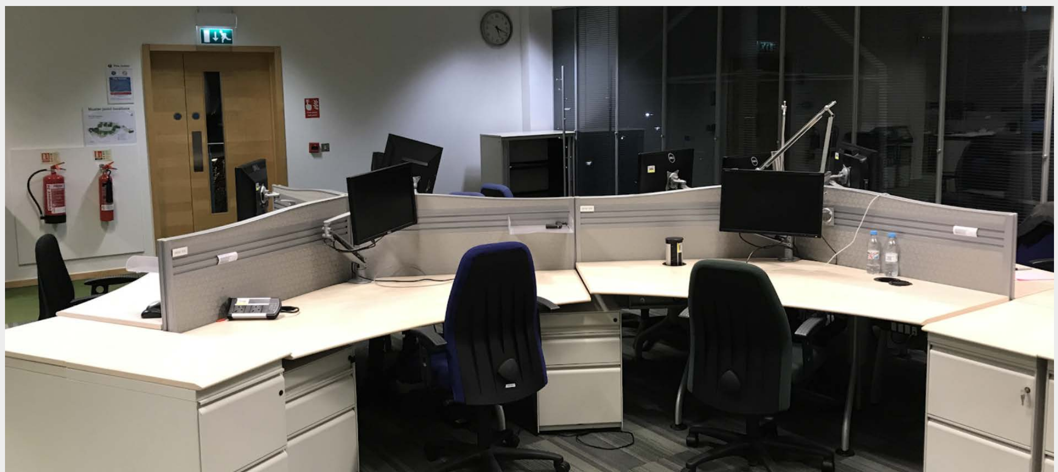


Figure 5.14: The traditional office space

To this:

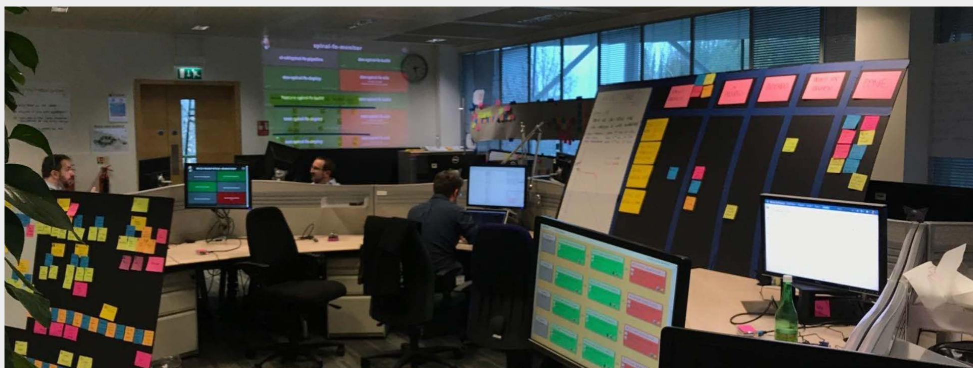


Figure 5.15: Existing space quickly transformed

Without touching any fixtures, fittings, moving any furniture, or fixing anything to walls or windows.

It gave us just enough of the right team workspace environment to enable some happy, empowered teams:

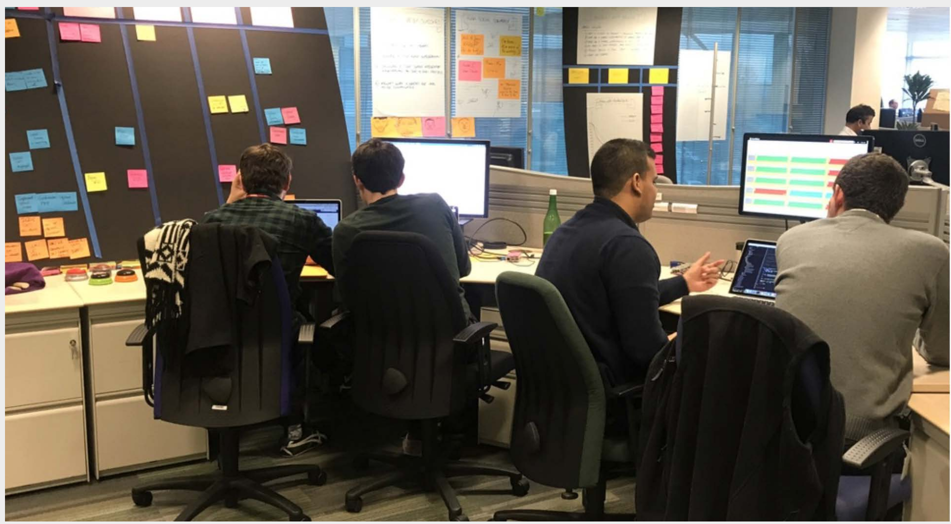


Figure 5.16: Collaboration facilitated by make-shift space

After the four-week engagement, I got some feedback from the now CIO. He said that Security watched the whole engagement with interest and moved from a mindset of saying No, *you can't do anything* to *we see what you want to do and why and we'll help you get there.*

You may be looking at the images and examples above and thinking it is impossible to have so many people in the same space at the same time. We often continue to probe such statements as we believe so strongly in the value of co-location and energetic workspaces full of visualization and information radiation. Of course, with our recent experience of the COVID-19 pandemic, it really did become physically impossible for most of us and we've all pivoted to using virtual spaces.

Virtual Spaces

In this chapter and the previous chapter, we've looked at individual practices and given some thoughts and guidance on how to run them with people distributed. With virtual engagement and a distributed team, there usually is no physical space to consider. We are immediately challenged and have some risks in not creating the autonomy and psychologically safe space to facilitate great collaboration.

There are two spaces we still need to consider and invest in:

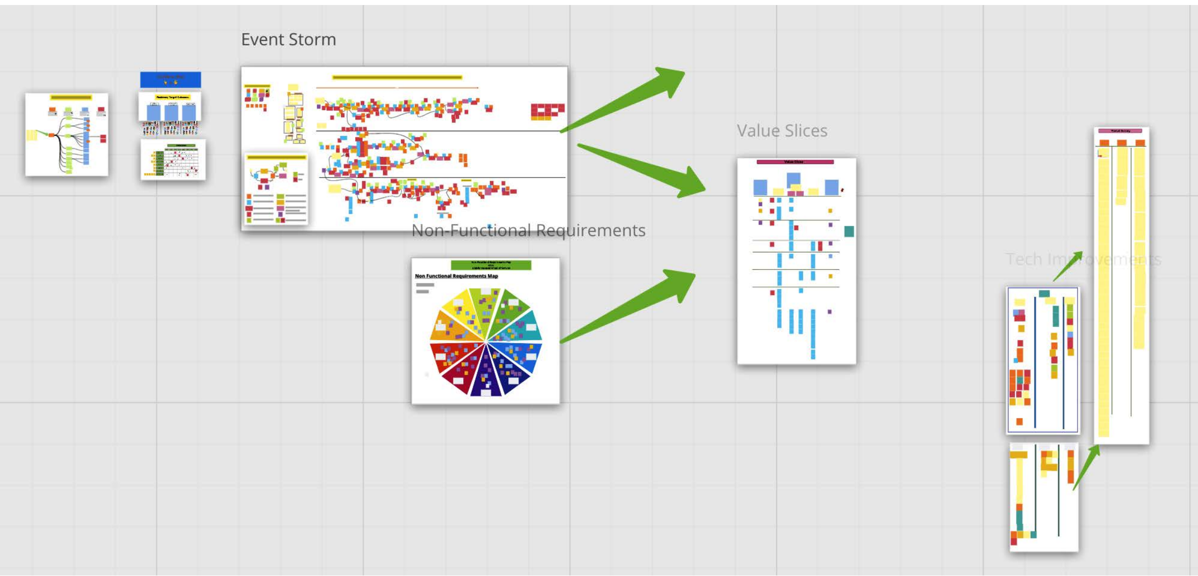
- Firstly, each individual's space. In the previous chapter, we explained how we should *radiate everything* and, even in virtual settings, individuals need good spaces, multiple monitors, tablets they can use as build monitors, their own wall space, and so on. Investing time and money upfront to suitably equip each team member will set the overall team up for success.
- Secondly, there is the digital space. That tends to mean investment in solid tooling, of which there are a growing set of vendors who offer support. Our guidance is to think about all the practices and ways of working that work so well in the physical space and find a tool or set of tools that will allow the emulation of that exact same practice. So, if you use the burndown chart practice, find a way to do that digitally (such as JIRA[5]). If you have a product backlog of index cards that everyone works from, find a tool that can manage that (such as Trello). If you regularly collaborate with people moving sticky notes around walls and annotating them, find a tool that enables that (such as Miro[6] or Mural[7]). Don't settle on one tool. Don't decide on one upfront. Take an agile approach and encourage the experimentation and empowerment of the people actually using them.

5 https://www.atlassian.com/software/jira

6 https://miro.com/

7 https://www.mural.co/

Suddenly the space looks like this:



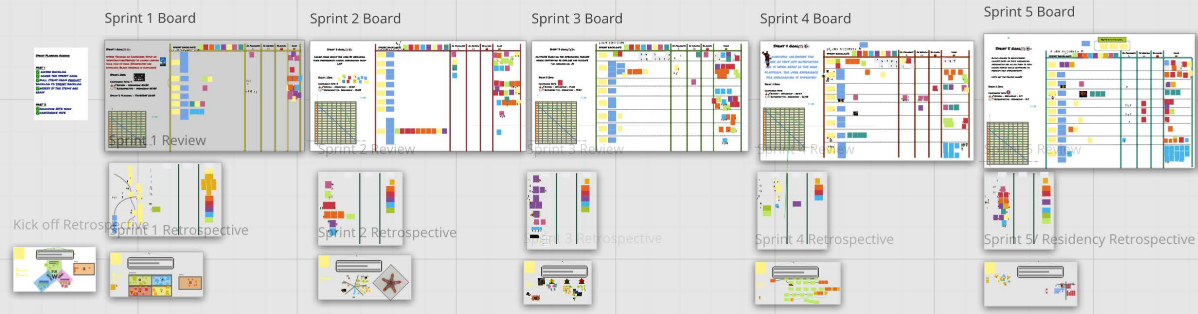


Figure 5.17: Digital walls

You can see the differences between physical space and digital space and, if you have worked in both environments, you'll know the difference in feeling and energy. They are very different, and we've learned there are benefits to both of them. Whilst the virtual workspace does not provide the same culture, energy, and multi-dimensional visualization you can have in the 360 degrees around you, it does provide you with infinite space. You will not run out of wall space. There are environmental benefits to not using thousands of sticky notes. Inviting subject matter experts from the other side of the world to review and comment on your team's collaboration activity is suddenly a lot easier and cheaper. Your security and audit folks may also feel more assured by using these digital tools. It will be interesting to see, in coming years, whether the virtual and physical space co-exist, if one will become the norm, or if they will merge together into some kind of hybrid.

Conclusion

In this chapter, we explored further what it means to be open with a particular focus on introducing open leadership and open spaces. The open culture practices introduced in *Chapter 4*, *Open Culture*, help teams become autonomous and self-managing. Leadership has a role to play in creating an environment that facilitates and enables this way of working.

We emphasized the importance of leadership setting intent when establishing a solid foundation for a team to base its product discovery and continuous delivery on. This is crucial for a DevOps culture to be successful. As we get further into this book to explore technical practices and how platforms such as OpenShift can deliver continuous business value, we need our teams to have a strong cultural foundation enabled and supported by an open leadership culture driving open organization behaviors from the top down.

We looked at some examples of strong, open leadership – just-in-time decision-making, collaborative priority sliders, and their role in helping find the right workspace for the team, either physical or virtual.

We explored some of our top recommendations to be considered in designing team spaces and also the minimal needs to start a team off with working with information radiation and in an open, collaborative space.

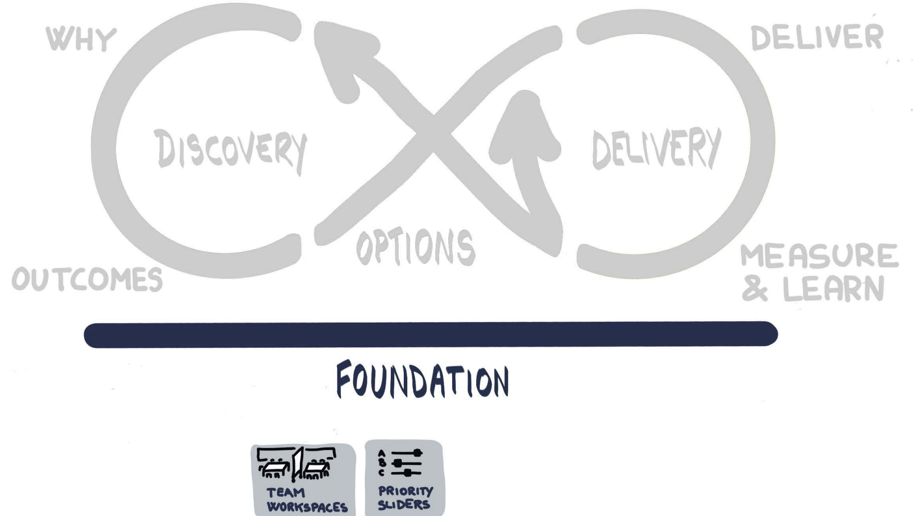


Figure 5.18: Adding more open culture practices to the foundation

With the mindset and practices adopted in this chapter and the previous chapter, we have a strong foundation of culture. In the next chapter, we will look at the other aspect of the foundation – technical practices and technical landscapes and what we do to establish a strong technical foundation prior to starting product development.

6

Open Technical Practices – Beginnings, Starting Right

So far, we have talked about the foundational culture and collaboration practices that support our discovery, options, and delivery Mobius loop. The next two chapters establish the technical practices that teams should implement to make the foundation even stronger.

Think of the Mobius loop as an engine turning from discovery through options generation and into delivery. This cycle continues by doing more delivery until we need to revisit the outcomes we've targeted. Delivery is where we take the concept and make it real. As we deliver, we will learn a lot and garner feedback from our stakeholders and our team. At some point in time, we will need to revisit the discovery side of the loop, either to adjust what we know or to realign what we deliver next.

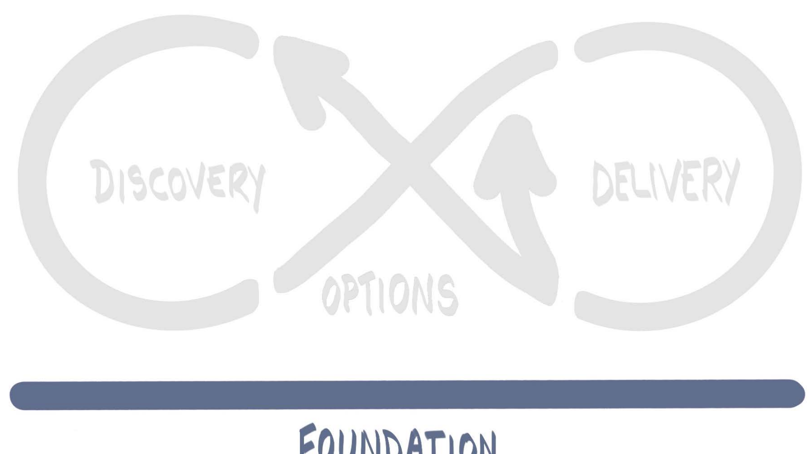


Figure 6.1: The Mobius loop

Imagine we've done one iteration and built some new features for our product, such as a leaderboard for our PetBattle app – it's very likely we'd be OK to demo and release that functionality without investing time or effort in building out a solid technical foundation. But as our iterations continue and the application's complexity grows, we may find ourselves beginning to drown in manual testing or technical debt. As we hit iteration five or six, things that were working will most likely start to break and our ability to predict how much we can do will start to fall apart. This could demotivate the team and have people lose confidence in our product or ability to execute. Breaking trust at this stage is a difficult thing to come back from.

To successfully build software incrementally, we need to ensure we can operate smoothly and sustainably. Constant rewrites and fixes will hinder this.

For these reasons, it's important we support our delivery capability with a set of foundational technical practices, much like we did with the collaboration and culture. Practices such as **configuration-as-code**, **infrastructure-as-code**, and even **everything-as-code** can help ensure a team's work is repeatable. Identifying how a team will do testing and automating that testing can lead to higher-quality output and simplify defect management. Picking the right development workflow and tools will accelerate the team's ability to deliver the software and not spend all their time managing their tools instead.

Adding practices to the foundation is not a one-time activity. As the applications grow in number and complexity, it's important to bolster the foundation with new and more comprehensive use of practices.

In the next two chapters, we will share the technical practices implemented on the foundational level that have enabled us to achieve the best success. They are part of the foundation because they are not time-boxed practices; rather, they are continually

carried out as part of our daily work. Later in the book, we'll explore how bolstering these practices with great use of the platform can enable sustainability and scalability of continuous delivery.

In this chapter, we want to start off right by covering the following:

- Green from go!
- Pairing and mobbing
- The container-native approach
- Pipelines: CI-CD²
- Everything as code
- Developer workflows

Green from Go!

Green from go! really just means setting yourself up the correct way when kicking off a new piece of work. For us, that means having all our foundational pieces of software up and running before writing any application software. It is another mantra of ours. Much like **show, not tell**, this one is all about getting things working with a light touch into a usable state.

For example, take choosing the tool we want to use to automate the building of our code, such as Jenkins or Tekton, or choosing how we manage secrets or access to private information. The ambition with **green from go!** should be obvious – clear the pathway to empower developers to get on and do what they do best – writing amazing, high-quality software that delights end users.

When we engage in any new piece of software delivery, we always ensure the tools we need to do our jobs are in place. We will have picked a few that will help us automate taking our code, compiling it, and delivering it to an environment where it can be tested. This means all the tools need to be re-creatable from scripts or another mechanism so we can easily kick-start any engagement to the same level with consistency and the learning from each run can be brought to the next one.

Green from go! will also include any reference apps or pipelines using the tools. This could be a simple AngularJS application scaffold with an end-to-end build process running against it, proving that we can take code and deliver it to users. The level of complexity of this process should be low enough to allow the teams to evolve it to their specific needs. For example, a team might want to do more non-functional testing in their pipeline, or perhaps another team wants to try out a new cool testing framework. The priority here is to have enough of a foundation to not slow the team down in doing these repeatable tasks when kicking off new work but to get them focused on writing new functionality.

It's also important to not have too much in the kit bag – it is our experience that when bringing new teams on board using an accelerator, the technical burden it can have can cause some team members to not engage with it. In order for teams to take ownership and maintain something that accelerates their delivery, they often need to feel like they helped to build it. If something is too complex to engage with, it becomes "that thing that Brian knows all about, so I don't need to know it." This kind of behavior is dangerous in a team as it can lead to silos and gaps in the team's collective ownership and responsibility. Often, when the build system then fails or breaks in some way, that person becomes the single point of failure when trying to recover it.

In the next section, let's look at two practices that help build collective ownership and understanding.

Pair Programming and Mob Programming

Pair programming and mob programming help us deal with a phenomenon that people term **Unicorn Developers**. It has various names across different regions and companies, such as **the Hero Developer** or **the Rockstar Developer**. But we all can identify who they are when we see them.

For those who don't know; the Unicorn Developer is the one who has all the knowledge and keeps it to themselves. They're the person who writes the most magnificent code, and the code that is usually the least understood. They have all the keys and secrets in their head, including all the ideas and knowledge. They are often the one producing so much new work that they don't have time to document it, meaning no one else can continue on the work in their absence. At this point, you can probably identify if your team has a Unicorn; it may even be you!

Figure 6.2: The Unicorn

So why do we have a problem with the Unicorn?

The Unicorn is a bottleneck and an icon of failed projects. They are the single point of failure in the system. When the Unicorn goes on holiday, projects grind to a halt. When things go wrong, the Unicorn has to step in to fix things, meaning new work cannot be completed while they are preoccupied.

Organizations want to create high-performing teams around their products – they want entire teams of **Rockstars**. A great philosophy in achieving this is to "mob to learn, pair to build":

- Pair programming involves two engineers working together with one computer on one problem at a time.
- Mob programming[1] involves an entire team working together with one machine to solve one problem.

Pairs and mobs of engineers allow for more knowledge transfer and create a shared understanding of the problem and the journey to get to the solution.

Mob to Learn, Pair to Build

To understand why pairing is different, let's use an analogy. Imagine you're a carpenter and you look at a magnificent rocking chair. What do you learn from seeing the completed piece? Probably not a lot; you might see how one or two pieces connect, but not all of them. Now imagine you worked with the carpenter assembling and crafting the pieces. You'd get to experience the hidden parts, the joinery that was used, how it was created, and how it's all connected. You'd identify the problems faced while fitting the parts together and understand why they're assembled in a given order. You could take a step back and review the furniture as it's being built, giving you a better perspective on the creation process. The same applies when writing and engineering software. Pairing makes better programmers.

I can hear the skeptics out there thinking to themselves, hmmm sounds like two developers doing one person's job. Doesn't sound very cost effective to me…. Well, there are a number of interesting advantages to pair programming and mobbing:

- **Mentoring of team members**: New team members can be brought up to speed quickly when they work alongside others, tackling the same problem as someone who knows the code better. Sharing tips and tricks or shortcuts can widen both pairs' skill depth. This sharing can also bring junior associates up to speed very rapidly.
- **Half the code**: When you ask an organization to spend two developers on one problem, the usual math kicks in of "won't that mean half the code gets written? In truth, hopefully even less code gets written! Two minds working on the same problem makes for more efficiently written code, so less spaghetti.

1 A relatively new practice championed by Woody Zuill – https://woodyzuill.com/

- **No pull requests**: Pairing means you have to share your thought process with your pair. This synchronization means code is being reviewed as it's written. Often, people reviewing pull requests are too busy writing their own code and they can only give very superficial reviews. When you pair, you review as you go and therefore write leaner, better-understood code. You won't cut corners when pairing as someone is watching.
- **Team bonding**: Humans are social creatures; we share and interact all the time. Pairing and mobbing facilitates this interaction. Instead of sitting in a room with headphones in, ignoring the world around you, developers in pairs look happier. A room with mobbing and pairing going on is louder. Happy coders lead to better code.
- **The knowledge stays in the team**: With more than one mind solving the same problem, the understanding and logic stays with them. As pairs naturally shuffle from task to task, the depth of knowledge stays with the team and not with an individual. This means when holidays or even flu season take over, the team can still continue to work at pace knowing the Unicorn is not leaving with key information.

Figure 6.3: Mob programming in action

When tackling new problems (whether that's a new framework, a new language, or even a particularly hard problem), we will all group together. Huddled around one computer with a very large screen, we can explore the problem we're trying to solve. We mob around the problem until the cross-functional team is satisfied that they have enough knowledge or a rough scaffold of how to complete their tasks. The team then breaks away into groups of two to pull items from the backlog and begin implementation.

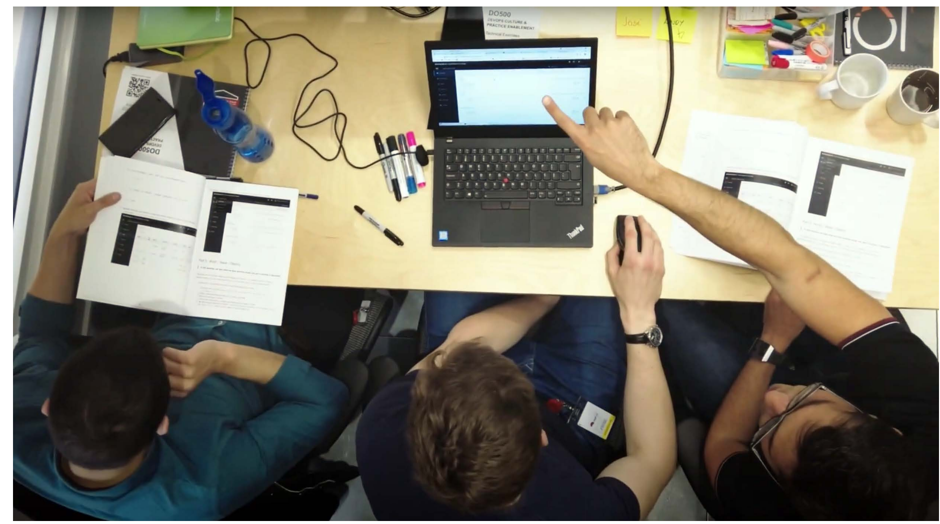


Figure 6.4: Pair programming in action

Mobbing and pairing allows teams to cross-skill. Sharing experience and expertise leads to better teams. Well-oiled teams working like this can continue to build their product sustainably and at pace, driving toward outcomes, not outputs.

You can learn more and collaborate about these practices by going to the Open Practice Library site at https://openpracticelibrary.com/practice/pair-programming/ and https://openpracticelibrary.com/practice/mob-programming/.

Containers and Being Container-Native

Before we can define exactly what containers are (hint: they are Linux processes!) and what container-native means, we need to look back in time to see what led to containers.

Container History

If you are over a certain age (over 30!), it is very likely your first computer program involved compiling source code and statically linking it with libraries from the operating system. Computer scientists then invented dynamic linking – which is great: you could patch one library and all of the programs you had written would pick up that change once restarted. This of course created a different problem – managing all of the dependencies. Packaging technologies such as RPM and YUM were created to help solve the dependency problem when distributing and managing Linux operating systems. Operating system distributions are one mechanism for collaboratively sharing and managing lots of different software packages at scale, and ultimately it is the software communities that evolve around these different software packages that solve real-world problems.

Of course, running your application on one physical machine is fine, but running lots of applications across lots of machines becomes a standard requirement as you scale. Virtualization solved how to run many operating systems on one machine in an isolated fashion. Indeed, the prominent form of cloud computing has been running virtual machines on someone else's hardware.

Virtualized infrastructure solved many problems of running applications at scale. However, configuring all of the pieces required to manage a fleet of **virtual machines** (**VMs**) leading to an explosion of tooling and technology related to configuration management. There was also the problem of "VM sprawl" – lots of VMs everywhere using too many resources that were difficult to patch and manage as a whole. Every application was managed independently, possibly by different teams. It was seen as desirable to reduce the interdependency between each application and so each application was also deployed in its own VM. To help control this spread of VMs, they were managed centrally by an infrastructure and operations team. Silos between teams were built! Many tools were created to help configure VMs. Each VM has overhead for running system processes and daemons, and so a lot of effort has been spent building tools that help avoid over-allocating machine resources to help save money.

For developers, the user interface in a VM deployed within an organization was not particularly self-service. Requesting a VM to be provisioned takes time. Workflow, ticketing, and provisioning systems were automated to try and help speed up this service request process. This was made radically better by public cloud services with an API-driven infrastructure, where provisioning a VM takes minutes and there is real self-service for developers. The control and VM sprawl issues still exist, though.

The application stack that developers used is still dependent on the operating system and libraries packaged into the VM that came with the kernel (for example, **libc**, **libssl**). And developers were usually not allowed to change the VM configuration, either because of perceived security or stability concerns. This was an infrastructure or operations team responsibility. Often, VMs were not easy to update, patch, and manage. It was not clear to the infrastructure or operations team what the effect of updating and rebooting a machine would have on the applications they supported.

How Containers Work

It is often said that the journey of containers is one of **process isolation**. The containers concept likely started with chroot in 1979, then graduated to BSD Linux jails in the early 2000s where Solaris Containers picked them up in 2004.[2] Solaris zones were a form of technology that isolated and combined system resource controls and boundary separation. From the outside, they looked like VMs, but they were not.

2 https://www.section.io/engineering-education/history-of-container-technology/

Technology companies that run a large number of workloads are always looking at ways to save resources and ultimately be more efficient. Roll forward to 2006 and a number of technology enhancements were made within the core Linux kernel that was related to the isolation of Linux processes. Google introduced the technology initially called process containers that was later renamed cgroups. It was designed for limiting, accounting, and isolating resource usage (CPU, memory, disk I/O, and network) of a collection of processes.

A novel solution to the dependency problem for containers was introduced by Docker in 2013. Packaging applications and their dependencies into container images lead to an explosion in popularity for containers. Container images were made freely available and distributed online via container registries such as dockerhub.io and quay.io.

A running container is really just Linux processes with extra protections and data structures supporting the process in the kernel. Running containers on a single machine was easy; running thousands of containers across a compute farm was a much harder problem to solve. Enter into the scene container orchestration engines of which the Kubernetes project is by far the most widely used today. The OpenShift Container Platform is a product that brings together Linux, Kubernetes, and container technologies to allow enterprises to run containers safely and securely at scale in the enterprise.

Of course, to get to real business value, it is not enough to package your applications as containers and deploy a Kubernetes platform such as OpenShift. Just because you build it does not mean that all users will immediately flock to the platform! Modern application delivery using trusted supply chains forces new tools and ways of working onto your teams. New behaviors are required.

With containers, the developer's user experience has been radically changed. Developers can now self-service their applications without having to go through the provisioning of a VM. Of course, someone still had to provision the platform! Provisioning and starting of containers took seconds and minutes, and today with serverless-focused technology stacks, milliseconds.

Developers can control the packaging, running, and upgrading of their applications easily using container images. The application is no longer tied to the version of libraries packaged in the kernel. It is possible to pull out all of an application's code and dependencies into a container image. You can run multiple versions of the same application together without being dependent on the same version of libraries in the kernel.

The immutable nature of a container image also improved the overall service quality of applications. Teams could ensure that exactly the same container image would be run in different environments, such as development and production. To be able to run this immutable container image in different environments, developers started to learn that by externalizing their application configuration they could easily run the same container anywhere. The application configuration management was now built in as part of the container deployment process and the platform. This led to clearer boundaries between what the **developers** controlled (their applications and configuration) and what **ITOps** controlled (the platform itself).

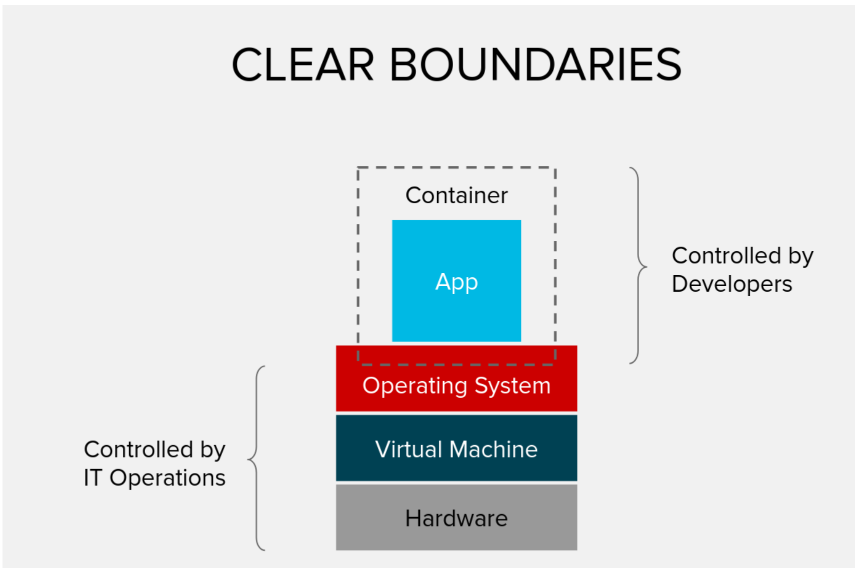


Figure 6.5: Containerization providing clear boundaries

In a multi-tenant environment, different groups of users can isolate via projects so as to increase utilization of the underlying infrastructure. In OpenShift there are built-in mechanisms for controlling network ingress and egress, role-based access control, and security, as well as out-of-the-box metrics, monitoring, and alerting capabilities. The platform supports the idea of mounting persistent data storage into your containers. The platform supports these stateful applications so that when a container is stopped/restarted or moved to another compute node, so too is the persistent volume.

The demarcation of team roles within a container ecosystem is different compared to virtualized infrastructure. **InfraOps** teams can manage the OpenShift platform and supporting infrastructure, while development teams can self-service provision and run application services on the platform. It is a "set up and get out of the way" mentality. Of course, there are still complexities that need to be discussed and agreed upon before you can reach this goal. When to run cluster-wide services and operators, how to perform rolling platform upgrades while managing business application service

levels under change, security, storage, high availability, and load balancing/networking concerns usually require everyone's involvement. It is the coming together of these teams and the DevOps conversations between them that form the backbone of modern DevOps practices today.

You can learn more and collaborate about the containers practice by going to the Open Practice Library page at openpracticelibrary.com/practice/containers.

Pipelines — CI or CD or CD²?

"*The job of a pipeline is to prove your code is not releasable.*" – Jez Humble

OK – let's set the scene and get some basics out of the way. How do we take our code from individual lines of text on a laptop to being an application running in a container in production? Well, there are lots of ways with lots of kooky-sounding names! Teams call the journey our software goes through a pipeline, but there are numerous ways to implement one.

Let's pause for a minute and think about what a software pipeline really is with the help of our friend Derek, the DevOps Dinosaur!

Derek the DevOps Dinosaur

Before I joined the Red Hat Open Innovation Labs team, I was a developer working for a large system integrator. While there, someone asked me to explain what a pipeline is – referring to some build automation I had created. The person asking me was an executive partner and had very limited technical knowledge. He wanted to know what a pipeline is in simple language that he could understand and relate to when talking to his customers. His questions were fundamental, such as what does one look like and what should it do?

While thinking of ways to describe a pipeline in a simplified, relatable way, I kept thinking about whether I could explain it in a way that a three-year-old would understand – I could probably explain it to him. And so, Derek the DevOps Dinosaur was born.

Let's Forget about Software for a Minute...

Imagine for a moment that we're not building software. We're not configuring Jenkins, dealing with shells, Ansible, or any other automation tool. Let's imagine we're building dinosaurs! Big, scary, tough, monstrous, and ferocious ones with lots of teeth! Close your eyes and imagine the scary dinosaur for yourself. Maybe you're imagining some kind of hybrid Jurassic Park dinosaur. Think about the parts of the dinosaur you'd want to build – how many teeth does it have? How many arms and legs? When I think of my scary dinosaur, I think of Derek.

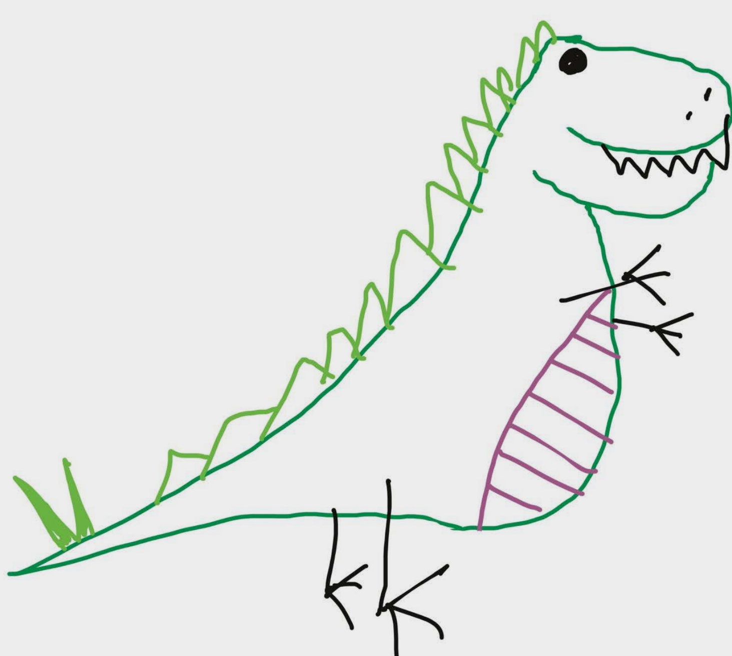


Figure 6.6: Introducing Derek

So, how do we know Derek is as big and scary as I need him to be? Let's start with his parts. First, we might analyze each of the individual parts of our dinosaur. Give them a quick check-over and ensure they meet the standard we set. For example, do I have two arms and two legs for my dinosaur? Has he got enough teeth? If it all looks good, we can then pop the parts in the Dino-Constructor 5000™.

With the Dino-Constructor 5000™ complete, we should hopefully produce our dinosaur, Derek.

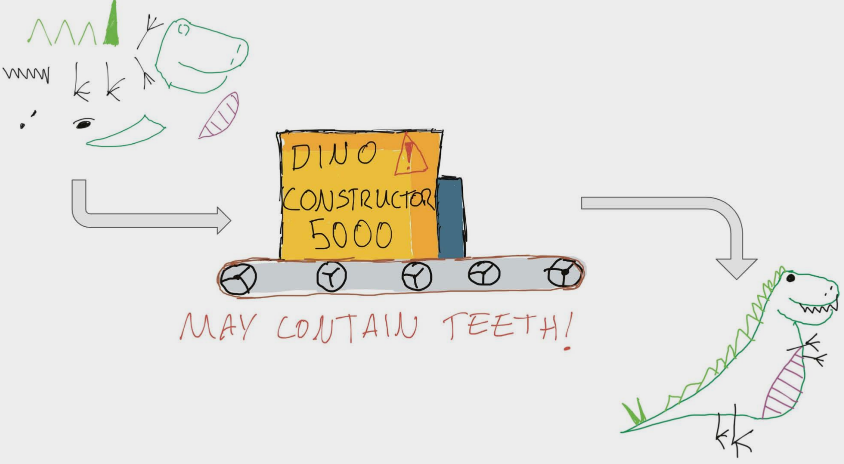


Figure 6.7: Introducing the Dino-Constructor 5000™

How Do I Know My Dinosaur Is Fierce Enough?

So, we've got a Dinosaur. But remember, we're here to build ferocious scary dinosaurs that are tough and fit. How do we know Derek is tough enough? Well, we could put him through a series of obstacles. Let's build an obstacle course for Derek.

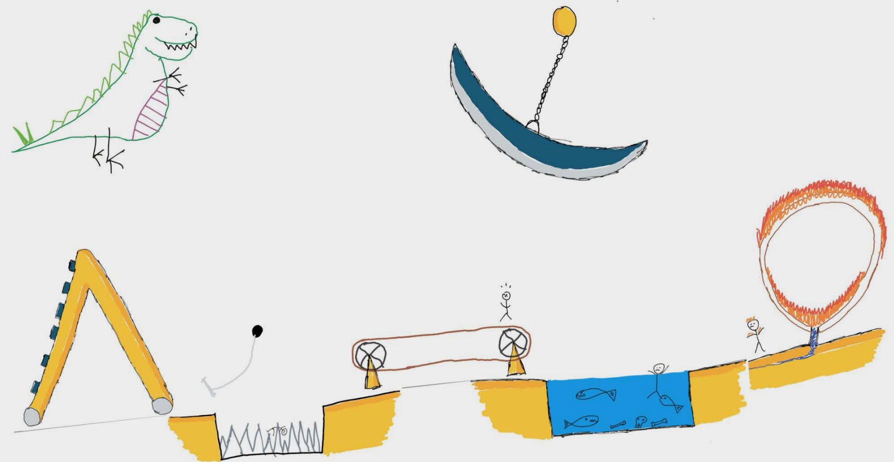


Figure 6.8: The dinosaur obstacle course

We'll start him on a climbing wall, much like the one you'd see recruits on in an army movie. Then if he's lucky enough to get over that hurdle, he's on to the next obstacle where he must jump over some spikes, Indiana Jones style! Next, we check how fit our dinosaur is; if he's able to run fast on the treadmill, he gets to pass on to the next step. Here he must try swimming past some fish that are trying to nibble on him. Once through that, perhaps he has to jump through a ring of fire. If Derek is capable and makes it through the obstacles, he can then run toward his pen – however, if Derek is not careful, he may be stopped by the swinging blade that was menacingly looming over him the whole time, like something from a Mario level. At any time, the blade could drop and stop Derek dead in his tracks. Let's for a moment assume Derek was careful and has made it into the pen where the other dinosaurs are.

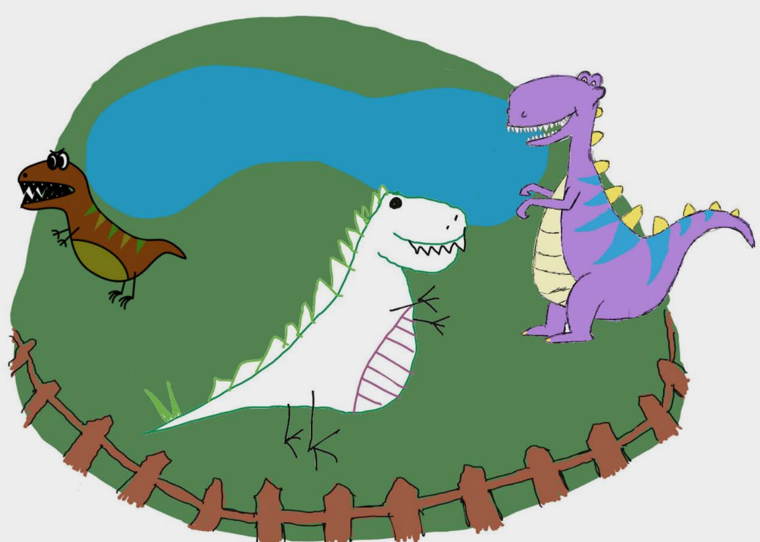


Figure 6.9: The dinosaur pen

Derek can now live out his days with the other dinosaurs in the Dino Petting Zoo, Danny and Debbie. However, unfortunately for Derek, Debbie the dinosaur is quite mean to him. She keeps stealing all of the precious grass and water that Derek likes to eat (Derek is vegetarian in this metaphor!). So, in order to give Derek the isolation and correct amount of things he needs to be strong and healthy, the zookeeper comes along and moves him to a new pen.

Derek, as it turns out, is quite a popular dinosaur at the zoo, so the zookeeper decides to make clones of him and puts them all in a pen with Derek. He is happy here and has enough of all the things he needs to survive.

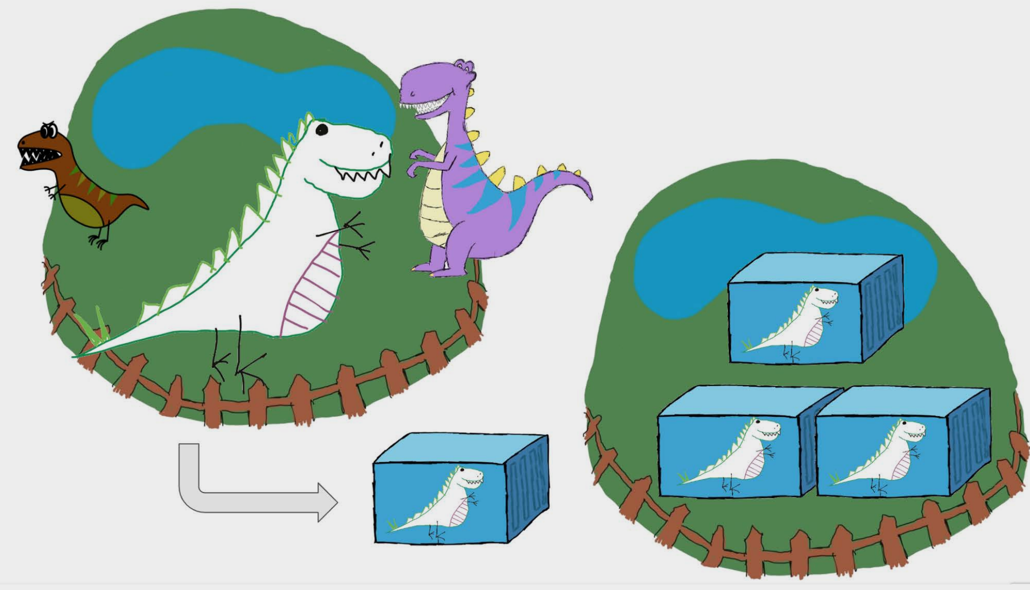


Figure 6.10: The zookeeper moves Derek to a new pen

But Wait – We're Building Software, Not Dinosaurs!

Sorry to shatter the illusion, but yes, we're (sadly) not in the business of making dinosaurs. We are here to build software applications. What we have just done to our dinosaur is the same thing we do to our code base on every commit. We build our code, run it through a series of obstacles, and then deploy it for our users to consume it. This is a pipeline; it's quite simple really!

Let's look at our dinosaur pipeline in more detail. In the first step, we assess the individual pieces that make up our dinosaur – its arms, legs, teeth, and so on. We ask questions such as are there enough parts? Does each hand have three fingers? I often think of this step as the static code analysis part of a pipeline. In the JavaScript world, this could be as simple as linting the code base or perhaps even running something more complex such as SonarQube to inspect the code quality. The Dino-Constructor 5000™ represents the compile step of any language.

The obstacle course we built for Derek represents the steps we should carry out to further assess our code quality. The initial hurdle Derek must get over could represent some unit testing. It is important that these obstacles are tough enough of a challenge while also not being so easy that they provide no value. For example, if Derek can make it over the climbing wall with ease, then it's probably not testing all the parts of him. Imagine for a moment that we decided to add another arm to Derek. We now have a terrifying three-armed dinosaur! If we were to ask him to climb the wall again, he would find it much simpler than before. In this regard it is important to increase the difficulty of the climb, perhaps widening the gaps or making it steeper so it presents more of a challenge. Thinking back to code, the logic still holds. When we introduce new features to our applications, we need to improve the testing coverage to include this. Writing tests is not a one-time thing; it must continue to evolve alongside our application development.

The other obstacles represent additional testing types. The small piranha pool Derek must swim through in order to get to safety could represent some early integration tests. The treadmill he must run on may be a kind of performance testing. The final obstacle Derek must pass unscathed is the giant blade hanging above him. Constantly looming, this testing type is, in my eyes, often the one that gets forgotten about. Derek may think he is free and run toward the pen only for the blade to drop on him and mean he can go no further – this is an example of security testing. Often forgotten about until the last minute, it can be a showstopper for final deployment in a lot of cases.

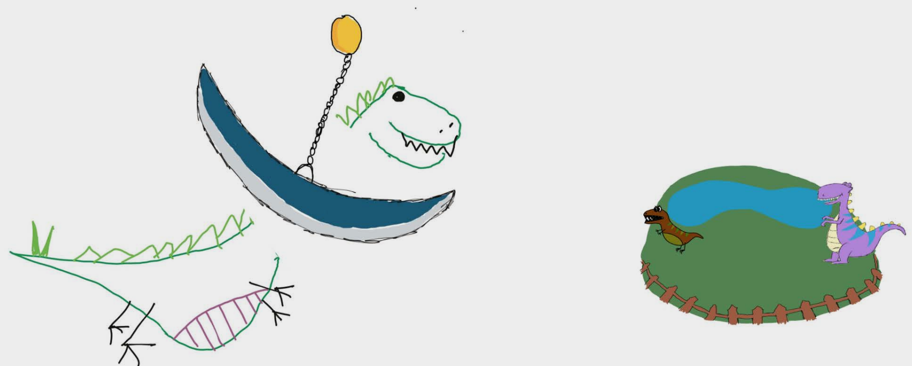


Figure 6.11: Failing to make the cut when moving to a new pen

Once Derek arrives at the dinosaur pen, he has to share the space with some other dinosaurs. Perhaps, at this point, the code has been deployed to a public cloud or a VM with competition for shared resources. Hopefully, by now, the operations team has noticed the application is running out of memory or there is a lack of compute. To combat this problem, the team might automate the containerization of the application. Once the code is in a container, it becomes shippable. We can move the container between cloud providers or even just between environments. At this stage, the code is packaged up with all of the dependencies it requires to run. This ability to move code without the need to rebuild and test can be safely achieved by building immutable container images. Versioning the application configuration separately from the built software means we can also horizontally scale our software easily by running more instances based on user demand.

A Final Thought on Building Dinosaurs

All of these testing types can, and should, be automated as part of a software pipeline. On each automated process that should execute building, testing, and deploying, the code should check if each proceeding step is successful. Through this process, teams can deliver new features faster. Teams can introduce new code without fear of regression. Container platforms such as Red Hat OpenShift and Kubernetes can ensure an application always exists in the desired state. These platforms can also be used to run our software pipelines, using build tools such as Jenkins to run the stages. Dynamic provisioning of test tools such as Zalenium to execute our browser tests as well as using Jenkins to build makes creating pipelines repeatable and reusable.

By automating all steps in a pipeline like this, we can ultimately get the dev and ops teams' awesome output into the hands of users quicker.

Thanks to Derek, we now know that a pipeline is a series of steps we use to build, package, test, and deploy our software. Now, let's look at some of the terminology people use to describe a software delivery pipeline.

Continuous Integration

Continuous Integration (**CI**) is a software development practice that was popularized by the authors of Extreme Programming. There have been countless books written about it but the shortest definitions are sometimes the simplest! The three-word definition of CI is to "integrate code continuously." That is to say, developers and teams should regularly commit and push their code into the repository and have some automated process to compile, package, and test that code. This process should happen frequently – many times throughout the day for maximum effect.

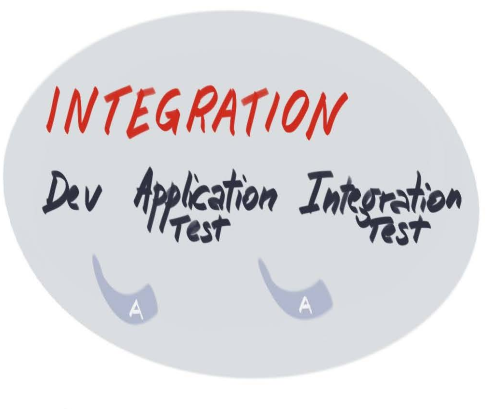


Figure 6.12: Continuous Integration

More teams fall down on this CI hurdle than you may think. Often, teams think they are practicing CI when in fact they are not.

Integrate Continuously

I worked on a Labs residency for a security company a few years ago. The team was fairly junior with several team members who'd just graduated. The team decided to create feature branches when writing their code so as to not break the workflows of others. Unfortunately, this led us to having these branches that lived for the duration of the sprint. We had lots of automation that was triggered when code was merged but we weren't merging frequently enough.

For two sprints, we had the same mad dash to merge all our features at the end of the sprint just before the weekly demo – and it was chaotic, to say the least! This resulted in bi-weekly "mini-integrations." We had lots of automation set up to validate our code but we were not using it frequently enough. As you can imagine, there is nothing continuous about this process – we were not integrating continuously!

To remedy this, we talked about it over our retrospective. If the tool you're using, in our case Jenkins, can give you data about the frequency of builds or its usage stats, these can be great things to print out or bring to a sprint retrospective. A brilliant Scrum Master I once worked with always did this and it helped the team focus during the retro on actionable things that we could do to make things faster. In our case on this residency, we were operating in one-week iterations. This meant only four days of actual development time! Through the retrospective, we identified a few actions from looking at the data being supplied to the team:

1. **Integrate continuously** – This was a big change for us, to try as often as possible to merge features together and get that validation we needed to avoid the merge hell we were encountering during demos.
2. **Smaller features** – The team realized that work was being broken down into too-large chunks. Each chunk was taking most of the sprint to complete. A smaller task size for each feature meant we could validate faster in smaller chunks whether things would work or not.

You can learn more and collaborate about the CI practice by going to the Open Practice Library page at openpracticelibrary.com/practice/continuous-integration.

Continuous Delivery

Continuous Delivery (**CD**) is a development process where on every code change, teams build, test, and package their code such that it can go all the way to production. It is delivered to the doorway of production in an automated way but not let in. Lots of teams get to this state, and it is a great place to get to, but are held back from releasing all the way to production usually due to organizational release cadences or additional approvals being required. The important thing here is that they could release to production if needs be.

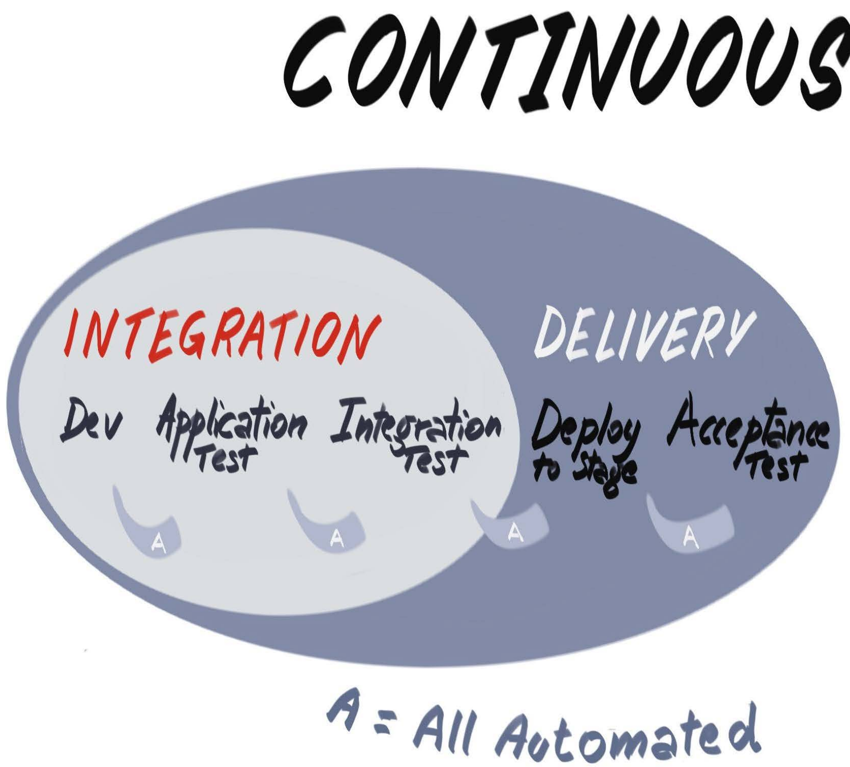


Figure 6.13: Continuous Delivery

Building Confidence in the Quality of the Software Delivery Pipeline

Early in my career when the concepts of automated testing and CD were new to me but still at the bleeding edge for some industries, I was working for a large retailer in the UK. They operated a very traditional approach to software deployments with a maximum of one release per quarter.

Deployments were a scary thing to them – they would involve a team of specialists who would come in during the dark hours of a Sunday morning to begin their manual task. They would take down the website, put up a holding page, and begin working through the script they were asked to run. Mostly this was a success, but on some occasions when things went wrong, they may have been left with outages for days!

This engagement was to build a mobile channel for the retailer to reach their customers. My role was to write some of the integration services between the mobile app and the commerce platform as well as to write a suite of automated integration tests. The retailer I was working for was very traditional and so they had in their project plan a three-week block prior to going live in which all the testing would occur. The retailer thought we were wasting our time writing automated tests and radiating the scores on a wall for all to see – they were confident the three-week window would be enough!

Our team was not willing to wait until the end to find out all the issues; they wanted feedback as they proceeded. We created a series of automated jobs in Jenkins to build the apps and APIs and deploy them to the user acceptance test environment. This meant that for months before the testing team was even engaged, we were delivering application revisions for them to test. Our automated tests emulated user behavior from the mobile app and tested for the happy path and all known error or sad paths through calling APIs with different input parameters. We also got hold of the user acceptance testing team's regression test scripts that would be manually executed and codified them as a set of tests for doing the same API calls. This excited the business as they began to see the app evolve. Features were getting added and issues were being fixed as they showed it off internally.

It was a new experience for them, as they were only used to seeing the whole thing at the end.

Fast forward to the end of the project and the business had started to see the value of the tests we'd written. On every change, we had automated the building of the mobile app, deploying it to the app store, and we ran a huge suite of integration tests. They continued to do their manual testing phase at the end, which did throw up a few bugs (which we then wrote automated tests for and fixed). However, when they compared the number of issues found during this phase against other similar projects, there were far fewer.

On the day of go live, the team was all set to push the app to the app stores and do the final deployment of the APIs. The retailer had marketing campaigns and other events aligned with this go live date, so the pressure was on! The teams were making minor app fixes right up to this point. Every change required the business to sign off the release, which meant involving the manual test team. Due to the pressure of the release window, the business decided to only do a quick smoke test of the app to see if the issue being fixed was resolved on a specific release candidate. This smoke test passed, so they were ready to roll – however, our automated tests threw up two failures in a service delivering product reviews within the application. There had been a minor change to the data format in the system of record further down the architecture that meant some data transformation functions were not working. This was not caught by the manual test team as they were not smoke testing this functionality. We flagged it up that our tests had spotted a regression, and the release was paused while this issue was resolved.

It may seem like a trivial example, but this marked a big turning point for the retailer. They'd witnessed first-hand the speed, reliability, and effectiveness of our automated test suite as well as the speed at which we could build, validate, and deliver a production-ready application. The act of writing and running automated tests built huge trust within the wider organization, prompting them to change their ways radically in favor of more automation and more test automation.

You can learn more and collaborate about the CD practice by going to the Open Practice Library page at openpracticelibrary.com/practice/continuous-delivery.

Continuous Deployment (CD²)

Continuous Deployment (**CD²**) takes the process of CD but goes one step further and delivers applications into production and therefore into the hands of our end users. I think of CD as a big train – one that operates on a very reliable timetable. It bundles up all the changes, taking everything in our repositories and compiling, packaging, testing, and promoting the application through all environments, verifying it at each stage.

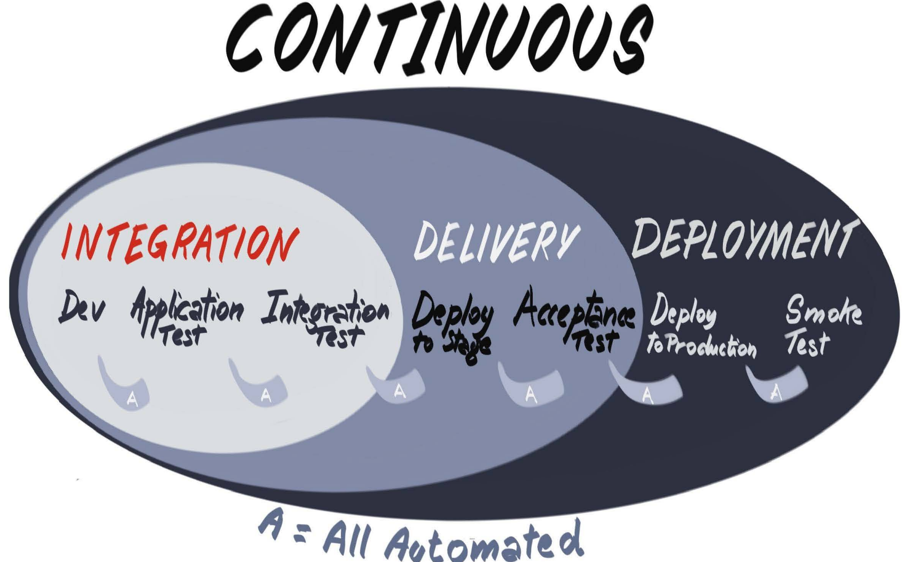
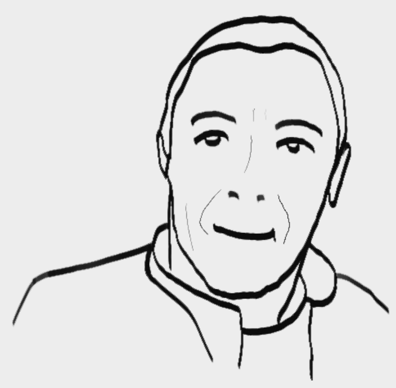


Figure 6.14: CI, CD, and CD²

By continuously delivering to production, you speed up the delivery of features and fixes to end users compared to holding back for big bang releases. Delivering faster leads to business agility – the ability to react to changing customer and market demands and generate feedback from features sooner. Developers will not have to wait weeks or months from when their code is written for an end user to try it. A quick feedback loop is vital and time and money should be spent considering the best tooling to enable this speedy delivery.

When the Work Is Done, Ship It!



Thinking about the ability to deliver at speed, where every change could be deployed to production, it's important to set the technology up to allow changes to flow freely with confidence. This requires strong buy-in from the people around the team, such as leadership and product owners, who can often block such efforts because too much change is considered harmful to quality or end user experience. These conceptions are often formed from previous bad experiences around failed deliveries. So, it is a two-way street – trust is built in that the team can execute with excellence.

One of the best product owners we've worked with was at a European car manufacturer. They were replacing a knowledge base application used by dealers and mechanics to diagnose problems and order parts. Historically, changes to this application were farmed out to suppliers with each one patching on their changes. They would hire a systems integrator to add some new functionality and in doing so would often introduce new bugs or issues. This outsourcing of development meant that architectural design decisions were made outside of the customers' product team, which led to technical debt and an unsustainable solution in the long run. The team decided to wipe the slate clean and rebuild the application by bringing the development in-house. We were engaged to help kick start this team the right way, using a residency, and help them build a product team connected to their end users.

A number of sprints into the engagement but still early in the development, the team was creating the authentication flow for users. I was pair programming with one of the engineers and we'd written the logout functionality. We had test-written and demonstrated the feature to the product owner running in our test environment. The Definition of Done the team agreed meant we had to show the feature to someone from the product team so they could accept it. So, as far as the engineering effort was concerned, we were done. The product owner did a quick test and it looked good in the test environment, so at the end of the sprint when we promoted all our changes up to production, our feature was released.

The UX folks were doing some usability testing with the latest increment of the application when they noticed some buggy behavior with logout not working from one of the screens. This was reported to the engineer and me, who worked on it initially, and we could spot the issue immediately. This was a small fix, so we wrote another test and made the change.

We demonstrated the process to the product owner – writing a test that failed, writing code that would make the test pass, iterating, and then delivering that fixed logout code all the way to production. The ability to deliver small incremental improvements into the hands of our end users when they were ready to be shipped paved the way to deliver continuously.

The lesson here is that Scrum may start to impede a team's ability to continuously deliver small incremental changes safely to production because Scrum delivers at the end of a sprint. "When the work is done, just ship it to production."

We have learned about the role of software pipelines, which codify the steps required to build, package, test, and deploy our application code into various environments up to but not necessarily including production – the practice of CD. We then looked at an approach to continuously deploying small incremental changes all the way to production.

You can learn more about CD² by going to the Open Practice Library page at openpracticelibrary.com/practice/continuous-deployment.

Everything-as-Code

You may have heard about this one before: [insert software term here]-as-code.

Examples include infrastructure-as-code, config-as-code, tests-as-code, and now everything-as-code. This practice has been around for a long time but some organizations have been slow to adopt it.

Here's the problem – historically, organizations have had to get expensive specialists to deploy complex environments. They would spend hours going through pages of instructions, line by line, eventually getting the deployment to work. A number of weeks would pass and the organization would like to create another environment, exactly like this one, for further testing. What do they do now? Call the specialist and ask them to come back at a great cost! This is fine, if you like hiring expensive specialists a lot.

So, what's the solution? The everything-as-code practice is simple: you treat every part of a system as you would any other line of code. You write it down and store it in a version control system, such as Git. Do we really mean to automate every part of the system? Yes.

We start by automating the infrastructure layer, the lowest level, from the bare metal servers to the operating systems, networks, application configuration, and on up through to application deployments.

This automation effort sounds like an awful lot of work, and could be expensive in terms of people's time – why should you invest in doing it? Here's why:

- **Traceability**: Having your environment descriptions and structure stored in a version control system allows us to audit changes made to the system, tracked to the individual who made them.
- **Repeatable**: Moving from one cloud provider to another should be a simple task. Picking a deployment target should be like shopping around for the best price that week. By storing all things as code, systems can be re-created in moments in various providers.
- **GitOps**: A single source of the truth means no more tribal knowledge or experts needed to set up cables or attach hard drives.
- **Phoenix server**: No more fears of configuration drift. If a server needs to be patched or randomly dies, that's OK. Just create it again from scratch using the stored configuration.

- **Cross-functional teams**: Writing all things as code improves collaboration between silos in an organization. The development team is able to contribute to the environment creation or can recreate their own like-for-like environments in a sandbox.
- **De-risking**: Changes can be applied to environments or application deployments and reverted to previous states quickly, thus de-risking big upgrades of any kind.

There are plenty of approaches to implementing everything-as-code:

- **Networks and infrastructure**: Ansible can be used to declaratively define the system you're implementing, and Istio can help with managing network traffic between apps and services.
- **Application environments**: Containerization provides a proven, repeatable way to package applications and their dependencies in a way that both developers and operators love.
- **Developer workflows/build automation**: Use Jenkins' Pipeline as Code or Tekton to describe how your application is taken from source, compiled, tested, and turned into something runnable.
- **Configuration drift**: ArgoCD is a tool that implements the GitOps pattern for your application and support tooling.
- **Testing as code**: Selenium tests written as acceptance criteria in the behavior-driven development form can bring business analysts and developers one step closer together.
- **Security and compliance**: Open Policy Agent and Advanced Cluster Manager are tools that enforce policies across the whole stack.

Teams who treat the whole system as code are stronger, faster, and better for it. We should no longer think about just infrastructure-as-code but automating the whole system – everything from application properties to networks and security policies. Then we codify it!

Can You Build a Second One of Those for Me, Please?

Spending time on automating the creation of test environments? "Sounds costly and a waste of my time" – I can hear some people reading this saying to themselves.

While working for a customer in the UK, I was building mobile apps and a bunch of JavaScript services to supply data to the apps in a consumable way optimized for the mobile. The services layer of adapters was deployed on IBM's MobileFirst (then Worklight), a big Java app that required a specialist to configure and install. We had several environments, from dev to system integration test environments to user acceptance test environments and production. All the common environments you'd imagine in a very traditional ecosystem.

The specialist spent two weeks configuring and installing the user acceptance test servers. Two of them were made available to allow us to have more than one thing under test at any given time. You wanted a third? Well, that meant bringing back that expensive specialist to build the third one and another week of their time. In production we had eight servers, each manually configured and deployed!

When I look back on this engagement and think about the pressure we faced to get the servers configured and deployed along with the time taken for each one, it seems like madness. The consultant would rock up, spend the day messing around on the terminal making manual changes here and there and manually testing the results. None of the config files back then were stored in Git or even turned into scripts that she could execute to make spinning up the next one faster. Every piece of information was tribal and in her head. We wanted a third server? We had to hire her to come back and do it all again!

> Some years later on another engagement for a public sector client, I saw similar behavior. I thought maybe creating servers in this way was a localized instance but on the government contract, there were teams spinning up servers for the developers to use that were not using any scripting or automation. If you wanted a server, you raised a ticket and waited a week. If you wanted an exact copy of that one, you raised another ticket and sometimes received one that was identical. In this case, the team was manually executing shell commands inside each VM and more often than not forgot to run a command or two!
>
> These examples may feel a bit old now – but the reality is that I still see organizations with a traditional approach to infrastructure, automation, and repeatability. Not being able to test changes on representative hardware can be a challenge for teams trying to go fast. Teams need to have the power to spin up and spin down application stacks on demand. Modern approaches to how we package applications, such as containers, can really help to bring down this wall. No longer does a developer need to stub out test cases with database calls, because they can just spin up a real database in a container and test against it.

You can learn more and collaborate about the everything-as-code practice by going to the Open Practice Library page at openpracticelibrary.com/practice/everything-as-code.

So, what approach did the PetBattle team take while practicing everything-as-code?

Establishing the Technical Foundation for PetBattle

This section will cover the beginning of our journey of PetBattle as the development team tries to set up a technical foundation with tools we will cover in later chapters. Any section in a box such as this one is going to lean in a bit more on the technical side.

PetBattle began life as a hobby for some engineers – a pet project, if you will. This project provides the team with a real-world application where they can try out new frameworks and technology. In order to wrap some modern software practices around PetBattle, they enhance the application with some build and test automation. As the demand for PetBattle increases, we will look at autoscaling and how we can apply practices from the Open Practice Library to identify how we should build things.

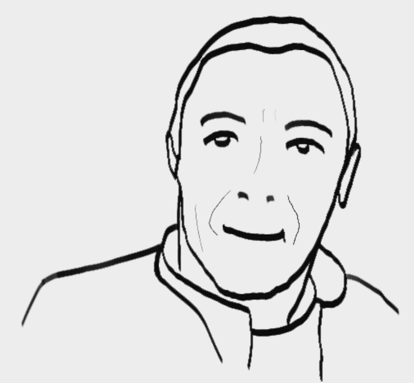
For PetBattle, we embrace modern software development paradigms – we monitor and respond to configuration drift so the team can implement GitOps to monitor this drift. Our environments should be like a phoenix, able to rise from the ashes! In other words, we can destroy them with confidence as we can recreate them from code.

Let's look at PetBattle's first piece of software they want to deploy, Jenkins. This section will explore how to deploy and manage Jenkins on OpenShift using Jenkins.

The PetBattle team is using OpenShift to deploy their applications. They have chosen to use Jenkins to get started with automating some of their tasks for building and deploying their software automatically. Jenkins is an open source automation server that can run many tasks and is supported on OpenShift. Jenkins also has a strong helpful community surrounding it and there is a large plugin ecosystem too, making automating almost any task you can think of a cinch!

Now that we have established PetBattle's technical foundation, let's explore Jenkins a little more and the role it can play in strengthening foundations.

Jenkins – Our Best Friend!

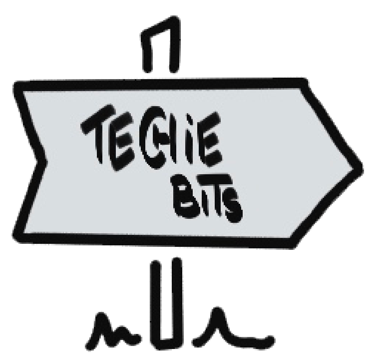


We like to think of Jenkins as our friend. We remember the days when teams would have someone build the app on their local machine and send it to the ops team via email. To do deployments, it would be a specialized team that came in, usually overnight, and did the deployment so as to minimize interruptions.

Deployments were seen as a risky, scary thing. One time, a team we worked with went out the night before a big deployment. When they stumbled into work in the wee hours of the morning they were not quite in the sharpest of mindsets. As you'd imagine when running an upgrade, they skipped a step and broke things. The reason we think of Jenkins as our friend is that he doesn't do things like that. He does not go out the night before and arrive at work tired (unless, of course, you forget to feed him lots of RAM and CPU). Jenkins also won't forget a line to execute in a script; he's pretty good in that way. But he's also pretty dumb in other ways; Jenkins is only as clever as the instructions you feed him. Jenkins in his vanilla form is fairly basic, so we give him additional superpowers to be able to run builds for specific technology using agents and to report test scores in a machine-readable way using plugins. But once he's got it once, he will do it over and over again without failing – especially if you configure him as code.

Helm Overview

This next section is going to get a bit more detailed on the technical side of things. Prepare for some code snippets and whatnot! If this is not your thing, feel free to skip over it to the next section all about Git and developer workflows. We'll mark any section that's going to have code snippets and be a bit lower level with this handy sign!

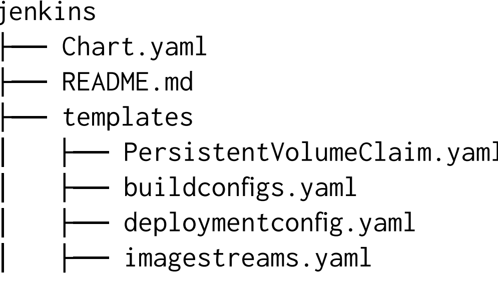


Jenkins comes with OpenShift, and there are several ways for the team to install and configure it. Any member of the cross-functional team could go to the OpenShift console and install it from the catalog. It's as simple as clicking a few buttons in the UI and choosing to add a persistent hard disk or not. This is a great way for the team to get moving fast but also would not honor our technical foundation practice of everything-as-code!

PetBattle now has two choices for how they could create an instance of Jenkins while honoring our everything-as-code practice. They could use OpenShift or Helm templates containing all of the Kubernetes and OpenShift objects that would be required to deploy a working Jenkins. For the purposes of this book, we will focus exclusively on Helm as our Kubernetes package manager.

Helm is an application package manager for Kubernetes that allows both developers and operators to easily package the resources and configuration that make up an application into a release. Helm is used for application life cycle management for installing, upgrading, and rolling back application deployments, thus simplifying the installation of an application on an OpenShift cluster. In Helm, applications are packaged up and distributed as Helm charts. A Helm chart is made up of several YAML files and templates. These Helm templates should output Kubernetes YAML once processed. Let's take a look at an example Helm chart.

From our experience using Jenkins on OpenShift with customers, we have written a chart to deploy the Red Hat instance of Jenkins and give it a few superpowers. We'll look at those afterward. Let's first explore the anatomy of a chart:



```
jenkins
├── Chart.yaml
├── README.md
├── templates
│   ├── PersistentVolumeClaim.yaml
│   ├── buildconfigs.yaml
│   ├── deploymentconfig.yaml
│   ├── imagestreams.yaml
```

```
│     ├── rolebinding.yaml
│     ├── route.yaml
│     ├── secret.yaml
│     ├── serviceaccount.yaml
│     └── services.yaml
└── values.yaml
```

The Jenkins chart, like all Helm charts, is made up of a number YAML files:

- `Chart.yaml`: This is the manifest of our Jenkins chart. It contains metadata such as the name, description, and maintainer information. The manifest also contains the application version and the version of the chart. If the chart has any dependencies on another chart or charts, they would also be listed here.
- `README.md`: Instructions for the chart, how to install it, and how to customize it.
- `templates/*`: This folder contains all the resources that need to be deployed to install and configure a running Jenkins instance, such as deployments, services, routes, and pvc.
- `values.yaml`: These are the sensible (default) values that the chart can be run with so a user can just install the chart and get up and running quickly. Customizations to these values can be supplied on the command line or by supplying your own values.yaml file when installing a chart.

Red Hat **Communities of Practice** (**CoP**) is an organization that creates reusable software based on experiences and learnings from working with customers. This software is then open sourced and shared. We can add the CoP Helm Charts repository, which contains a Jenkins Helm chart for us to use.

To start, we need the **helm** command-line tool. From your laptop, follow the instructions on the `helm.sh` (https://helm.sh/docs/intro/install/) website to install the **helm** tool. Then add the Red Hat CoP **helm** repository as follows:

```
helm repo add redhat-cop \
     https://redhat-cop.github.io/helm-charts
```

We can search this helm repository for Jenkins chart versions we can use:

```
donal — donal@dspring-mac — ~ — -zsh — Solarized Dark ansi — 110×5
#( 10/01/20@ 2:27pm )( donal@dspring-mac ):~
   helm search repo jenkins
NAME                 CHART VERSION   APP VERSION    DESCRIPTION
redhat-cop/jenkins   0.0.21          v2.222.1       A Helm chart for deploying Jenkins on OpenShift...
```

Figure 6.15: Searching the helm repository for the Jenkins chart

Installing Jenkins Using Helm

The quickest way to get access to your very own OpenShift cluster is to install CodeReady Containers on your laptop. Linux, Windows, and macOS are supported. You will need to log in and follow the instructions located here: https://developers.redhat.com/products/codeready-containers/overview, and you should see a two-step process similar to *Figure 6.16*:

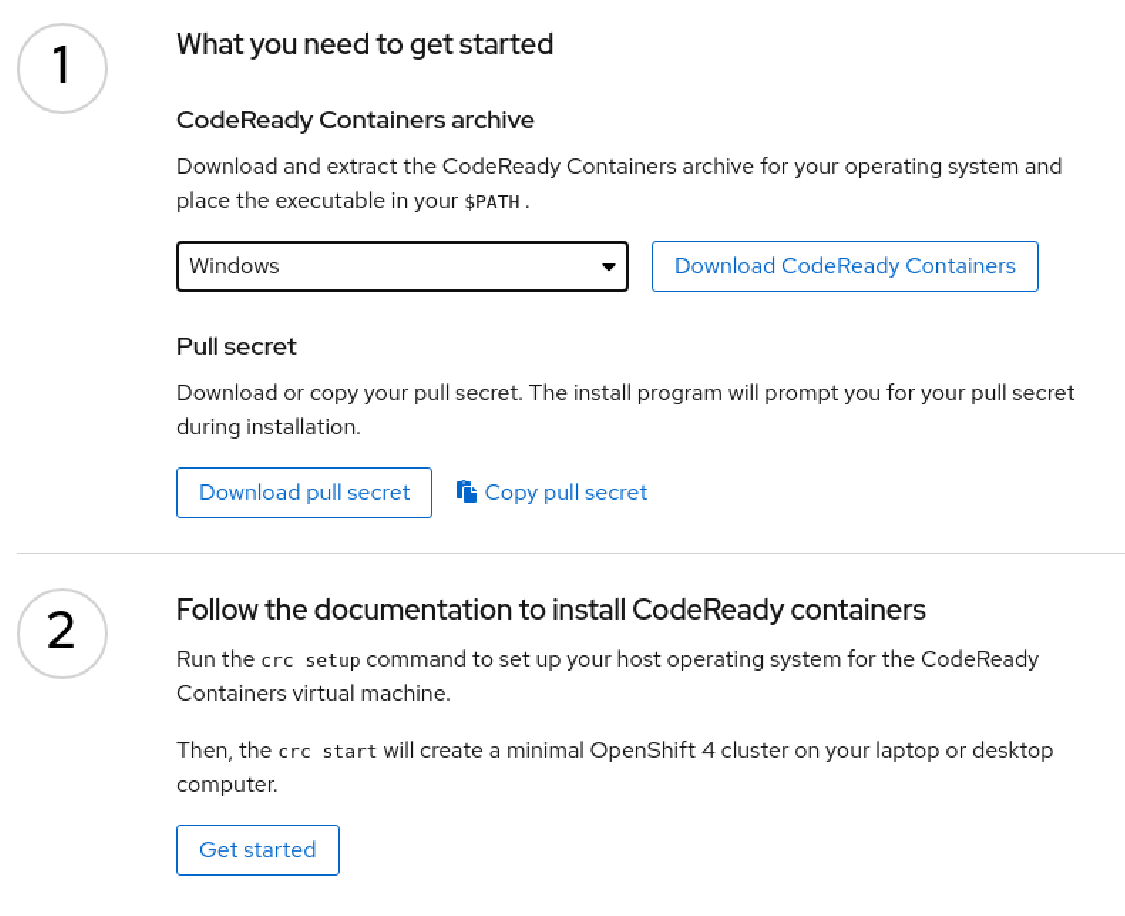


Figure 6.16: Installing CodeReady Containers

Other OpenShift clusters you have access to may also work, as long as you have sufficient resources and privileges. The CodeReady Containers install gives you cluster administrator privilege (the highest level of privilege) and is limited by how much RAM, CPU, and disk space your laptop has. We recommend 8 G RAM, 4 vCPUs, and 31 GB of disk space as a minimum, which would correspond to starting CRC on linux with:

```
crc start -c 4 -m 12288
```

There are more detailed OpenShift sizing instructions in the Appendix.

To install the Jenkins chart, we will log in to OpenShift, create a new project, and install the Helm chart. If you're missing any of the tools needed to run these commands, have no fear, as they can be downloaded and installed to match your OpenShift cluster version directly from the OpenShift console. Click on the **?** icon and then **Command Line Tools** to find the latest instructions.

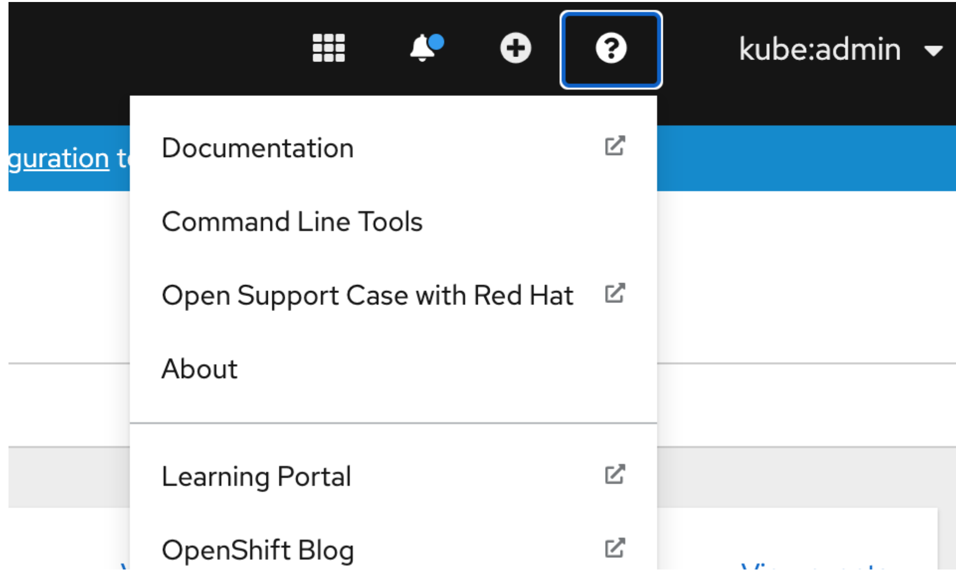


Figure 6.17: Downloading Command Line Tools from OpenShift

The string after installation, **my-jenkins**, is the release name that is used by the Helm template engine:

```
oc login <cluster_api> -u <name> -p <password>
oc new-project example
helm install my-jenkins redhat-cop/jenkins
```

It allows us to create multiple releases in the one namespace, which is useful for testing purposes:

```
donal — donal@dspring-mac — ~ — -zsh — Solarized Dark ansi — 105×10

#( 10/01/20@ 2:16pm )( donal@dspring-mac ):~
   helm install my-jenkins redhat-cop/jenkins

NAME: my-jenkins
LAST DEPLOYED: Thu Oct  1 14:16:48 2020
NAMESPACE: example
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Figure 6.18: Creating multiple releases in a single namespace

Helm charts can be installed in a number of ways. You can also run helm template against a local copy of the chart. If you are interested in doing this, fetch the chart and run this:

```
helm fetch redhat-cop/jenkins --version 0.0.23
helm template test jenkins-0.0.23.tgz
```

This can be useful if you want to play around and see what the output is before applying it to an OpenShift cluster or if you want to validate things while debugging or testing a chart's configuration. You can also supply `--dry-run` to the helm install command to verify the chart before installing it.

Let me just pause a minute and say that this is not a book about Helm! There are great books out there written specifically for it, such as Learn Helm (https://www.packtpub.com/product/learn-helm/9781839214295) by *Andy Block* and *Austin Dewey*. Our aim is just to scratch the surface to show how easy it is to get going in a reusable and repeatable way with Helm and OpenShift.

Using **helm install** as demonstrated previously is great as it will create a life cycle managed by the Helm CLI to run upgrades and roll back releases if needed. These revisions are integrated into OpenShift and can be viewed in the UI or on the command line. Every time a new revision is deployed to the cluster, a new secret will be created, making rollback very simple:

```
oc get secrets -n example | grep helm
```

To see all the pods being spun up by the Jenkins chart, you can run this:

```
oc get pods --watch -o wide -n example
```

You should see a large volume of pods being created – this is because this Helm chart contains lots of additional configuration-as-code for Jenkins. Write once and deploy many times:

```
Every 2.0s: oc get pods

NAME                            READY   STATUS      RESTARTS   AGE
jenkins-1-build                 1/1     Running     0          36m
jenkins-agent-argocd-1-build    0/1     Completed   0          36m
jenkins-agent-graalvm-1-build   0/1     Completed   0          36m
jenkins-agent-helm-1-build      0/1     Completed   0          36m
jenkins-agent-npm-1-build       0/1     Completed   0          36m
```

Figure 6.19: Pods being created

You may notice a bunch of agent build pods in the output. Jenkins by itself is a bit useless. One of Jenkins' superpowers is his ability to be extended using what are called plugins – small bits of code that provide new functions and features. To install these plugins, we could wait until Jenkins is deployed and configure the plugins manually through the UI – but this is the everything-as-code world, so we don't want to do that!

The Jenkins Helm chart is configured to pre-install a bunch of useful Jenkins agent plugins. These agents know how to build container images using various language-specific stacks. The configuration for the agent plugins is defined in the Helm chart's `values.yaml` file, which you can see by using this:

```
helm show values redhat-cop/jenkins

buildconfigs:
# Jenkins agents for running builds etc
  - name: "jenkins-agent-ansible"
    source_context_dir: "jenkins-agents/jenkins-agent-ansible"
    source_repo: *jarepo
    source_repo_ref: "master"
...
```

The Helm chart is defining a list of build configurations to build each agent image. The Jenkins agent images use an OpenShift project called **Source-to-Image** (**S2I**) to do their language-specific build of your applications. S2I is a toolkit and workflow for building reproducible container images from source code; you can read about it here: https://github.com/openshift/source-to-image. You basically feed S2I your source code via a Git repository URL and it takes care of the rest.

Using language-specific agents makes Jenkins easier to extend. We do not have to install tools into the base Jenkins image; rather, we define an agent and add it to the Helm chart agent plugins values list. OpenShift makes it very easy to create agents for Jenkins. We can extend the base image with any binary we want to use in our pipelines and apply the label `role=jenkins-slave` to make it discoverable in Jenkins. This gives us a near "serverless" ability for Jenkins to dynamically provision an agent when it's required. In this case, a pod gets launched and Jenkins will connect to it, execute its tasks, and destroy it when it's done. This means no agents lying idle waiting to be executed and a clean slate every time we run a build.

There are a bunch of Jenkins agents available in the CoP; you can use them or create your own: https://github.com/redhat-cop/containers-quickstarts/tree/master/jenkins-agents.

Apart from the agent plugins, the Jenkins image is extensible from the base image in a number of different ways. You can specify a list of plugins to install when you build the Jenkins image. We use S2I to build our Jenkins image and add our list of `plugins.txt` from this Git repository: https://github.com/rht-labs/s2i-config-jenkins.

Once the Jenkins build has completed, a Jenkins deployment and running container instance will be available.

```
donal — donal@dspring-mac — ~ — -zsh — Solarized Dark ansi — 105×24
#( 11/08/20@ 2:39pm )( donal@dspring-mac ):~
   oc get pods -n example
NAME                               READY   STATUS      RESTARTS   AGE
jenkins-1-build                    0/1     Completed   0          13m
jenkins-1-deploy                   0/1     Completed   0          3m56s
jenkins-1-xq8cd                    1/1     Running     0          3m52s
jenkins-agent-ansible-1-build      0/1     Completed   0          13m
jenkins-agent-arachni-1-build      0/1     Completed   0          13m
jenkins-agent-argocd-1-build       0/1     Completed   0          13m
jenkins-agent-conftest-1-build     0/1     Completed   0          13m
jenkins-agent-erlang-1-build       0/1     Completed   0          13m
jenkins-agent-golang-1-build       0/1     Completed   0          13m
jenkins-agent-graalvm-1-build      0/1     Completed   0          13m
jenkins-agent-gradle-1-build       0/1     Completed   0          13m
jenkins-agent-helm-1-build         0/1     Completed   0          13m
jenkins-agent-image-mgmt-1-build   0/1     Completed   0          13m
jenkins-agent-mongodb-1-build      0/1     Completed   0          13m
jenkins-agent-mvn-1-build          0/1     Completed   0          13m
jenkins-agent-npm-1-build          0/1     Completed   0          13m
jenkins-agent-python-1-build       0/1     Completed   0          13m
jenkins-agent-ruby-1-build         0/1     Completed   0          13m
jenkins-agent-rust-1-build         0/1     Completed   0          13m
#( 11/08/20@ 2:39pm )( donal@dspring-mac ):~
```

Figure 6.20: Available Jenkins deployment and a running container instance

All of the S2I plugins and agents are configured. You can log in to Jenkins using its route, which is available in the OpenShift web console, or by running this:

```
oc get route jenkins
```

By running this single `helm install` command, we get a sensible starting point to be able to do lots of things with our build server, Jenkins. By codifying the Jenkins configuration, we can repeatedly deploy Jenkins into many environments without ever having to touch the Jenkins UI.

Now that we have our build server, before starting development we should familiarize ourselves with the types of code workflows developers use. If you are an experienced developer, you will already be pretty familiar with the next section's content.

Developer Workflows

Git is a **version control system** (**VCS**) created by Linus Torvalds (author of the Linux kernel) to track changes in source code and easily manage these changes across many file types and developers. Git differs from other VCS in that it is decentralized. This means that unlike, for example, **Subversion** (**svn**), each developer retains a complete copy of the source code locally when they check it out. Locally, each developer has a copy of all the history and can rewind or fast forward to different versions as they need to. An engineer makes their changes and applies those changes as a delta on top of another's work. This is known as a commit. Git can be conceptualized as a tree, with a trunk of these changes or commits on top of each other. Branches can spring out from the trunk as independent pieces of functionality, or work that is not ready can be merged back to the trunk. Once something is committed to Git, it is forever in the history and can always be found – so be careful not to add something secret, such as a password, by mistake!

Git is the underlying technology behind some big companies such as GitHub and GitLab. They have taken the Git product and added some social features and issue-tracking capabilities to help manage a code base.

There are many workflows for Git that development teams can use when writing code, and choosing the correct one can seem like a daunting task. Some are designed to give teams a sense of safety and security, especially in large complex projects, while others promote speed and trust within the teams. The most popular source code management workflows for Git are Trunk, GitFlow, and GitHub Flow. Let's explore each in detail and see how we could use them to help us promote CD.

GitFlow

GitFlow was first published about 10 years ago by Vincent Driessen. The workflow was built from his experience using Git, a relatively new tool at the time. As teams moved to Git from a non-branching-based code repository, some new concepts and core practices had to be defined. GitFlow tried to answer this by adding a well-thought-out structure to branch names and their conventions.

A well-defined branching strategy is at the heart of GitFlow. Changes are committed to different named branches depending on the type of change. New features are developed on branches that are called `feature-*` branches. `hotfixes-*` branches are created for patching changes to bugs in production and a release branch. GitFlow describes two reserved and long-living branches:

- **Master**: This is the branch that contains our releases or our production-ready code. Sometimes this branch is referred to as the main branch.
- **Develop**: This branch is our integration branch. It is usually the most turbulent and very likely to contain bugs or other issues as it is the place where teams first bring their code together.

The naming and usage conventions defined in GitFlow make it easier for a new developer to discover what each branch is doing. The developer can bring additional changes made by other team members into their feature branch, when they choose to, by merging in any new changes. Branching in this way avoids breaking things for other engineers by ensuring that the feature functionality is complete before asking to merge their code from the feature into the develop branch. When a set of features is ready to be promoted to the mainline master branch, the developers merge their code to the master via the release branch.

You may be reading this and thinking, this sounds complex! And in some ways, it is. But in a large project with a single code base, this can be exactly what is required to ensure developers are free to work on their code without having to manage their code.

GitHub Flow

GitHub Flow is similar to GitFlow in that it shares some of the same words in its name. Branching is a core pillar of Git, and GitHub Flow uses this by keeping one long-lived branch, that is, the main or master branch. Developers then work in branches off main, where they can commit changes and experiment without affecting the main branch.

These could be feature branches like in GitFlow, but there is no naming convention to be followed. It is important to name the branch sensibly using a descriptive name, such as sign-up-form or refactor-auth-service. No branches called another-new-feature-branch, please!

At any point in time, the developer can raise a pull request, where other engineers can discuss the code and its approach, and design by providing feedback for the changes that are still in progress. The original author can then incorporate this discussion into the software. When the team is happy and the code is reviewed, the changes can be approved and merged to the main branch.

GitHub Flow is great at promoting the peer review of work and promoting transparency in how a decision was made. Git by its nature is searchable, and the discussion on a merge request provides valuable insight and traceability into how architectural and coding decisions were made.

Trunk-Based Development

Both GitHub Flow and GitFlow use branching strategies and merging patterns to bring independent development activities together. Branching in Git is made extremely easy. However, when merging all of the branches together, conflicts can still occur that require human intervention.

Depending on the frequency of this branching, some teams end up in merge hell, where everyone tries to bring their changes in at once, leading to the complex and often frustrating event of trying to unpick all the changes that were made while maintaining a working code base!

Trunk-based development (https://trunkbaseddevelopment.com/) takes a somewhat different approach to this particular problem by saying **no** to branches!

Figure 6.21: Merge hell

In trunk-based development, developers collaborate on one single main branch referred to as the trunk. Devs work on their changes and apply them directly to the trunk. In a perfect world, the commits are small in nature and frequent throughout the development process. The golden rules here are never break the build and always be release-ready. In this regard, a developer must always ensure this is the case. Development could be automated using some CI process, but the key is that the trust within the team must be there.

In a large-scale enterprise, this constant merging to master sounds like it could create headaches. How, for example, do you do a peer review of the code? For large-scale application development with many engineers and many teams, it is suggested that very short-lived feature branches can be a great help. They provide decision logs during the review process, but the key here is short. Short-lived feature branches should only be alive for a day or two at most (definitely no longer than a sprint) and are deleted once the code is merged to prevent them from becoming feature release branches.

Too Many Choices — Tell Me What to Do

Each of these Git workflows has been tried and tested with teams for years. Some teams choose one as a standard, whereas others adopt one or more depending on their own context.

The original author of GitFlow recently revised his views to suggest GitFlow does not work well for "applications that are delivered continuously" such as web apps. Branches can create distance between production code and work in progress and GitFlow sees code moving between several branches before it's released. If we think of a developer working on their feature, they get it merged into the develop branch and then they move on to a new feature branch.

The new feature sits waiting (potentially along with some other features that have been completed) until the end of the development cycle. At this point, it's bundled up and moved across to master via the release branch, possibly two weeks after the work was completed. All of these extra steps mean a developer is not getting the feedback they need from users in the field for a long time after the development is complete. In terms of a feedback loop, if the rework is required on the item or a bug arises, it could take weeks before it is rectified. Add to that the context switch for the developer, who has to go back over what they previously did, which could end up having an impact on the team's velocity.

In CI, does having feature branches slow us down? It's very easy for a developer to effectively hide out on their branch while development is happening. We have worked with teams in the past who have claimed to be doing CI, but their build system remains idle until the day of review. At this point, all the developers rush to integrate their features in a last-minute mini-integration session that often reveals misunderstanding in the designs or broken software. Long-lived feature branches do not easily marry up with CI.

Short-lived feature branches are a great way to help with some of these concerns. Developers can work away on small chunks in an isolated way and still merge frequently. Short feedback loops are the king of improving software delivery metrics. If branches add time to this loop, how can we tighten it further? Peer reviews can often be a burden to teams by creating a dependency on one individual or breaking the focus of another engineer in order to complete a feature. By pairing engineers, you gain implicit peer review. Pushing changes as a pair straight to the trunk is a great way to achieve speed. In a container ecosystem, you only want to build once and verify your application is working before deploying it to many places. Trunk-based development underpins this by encouraging frequent small changes pushed straight to the head, where CI and CD can then take over.

From our experience in kickstarting product teams with varying skill sets, choosing the right one should be seen as more of a pathway, a sliding scale from immature to mature teams. Teams that are new to Git may find the use of feature branches a comforting way to not step on the toes of other developers. The book *Accelerate*[3] measured the software delivery performance of many teams and concluded that high-performing teams use trunk-based development.

No matter what you choose as your approach to managing code, the key here is the frequency of delivery. How long will it take you to get software into the hands of your end users? Does having feature branches slow you down? Or do those branches provide you with a safe place for your team to start? As the team matures and becomes more familiar with each other and the tools, your software output can increase.

The big call to action here is to let the teams choose the way that works best for them and build the automation around the workflow and tools. This allows the developers to focus on the hard stuff – writing code not managing the code. Initially, this will just be a guess. Teams should use retrospectives to assess whether things are working or not and evolve accordingly. It's important to not set out one dogma to fit all development activities across all teams because every team is going to be different. One shoe size is not going to fit everyone!

3 https://itrevolution.com/book/accelerate/

Conclusion

In this chapter, we learned that we could get off to a great start by being green from go! By automating the deployment of our application build and packaging tools, Jenkins and Helm can establish a technical foundation that will allow our teams to integrate continuously (CI) and continuously deploy (CD) our code to production.

We learned that we can align our developer code workflow across our team and begin to iterate on our CI/CD pipelines to help us deliver applications faster. We can increase code quality and understanding by pairing developers together to help shorten the code review feedback loop.

As a team, we learned all of these new skills and techniques together by trying mob programming and, in the process, said goodbye to our love for Unicorn developers.

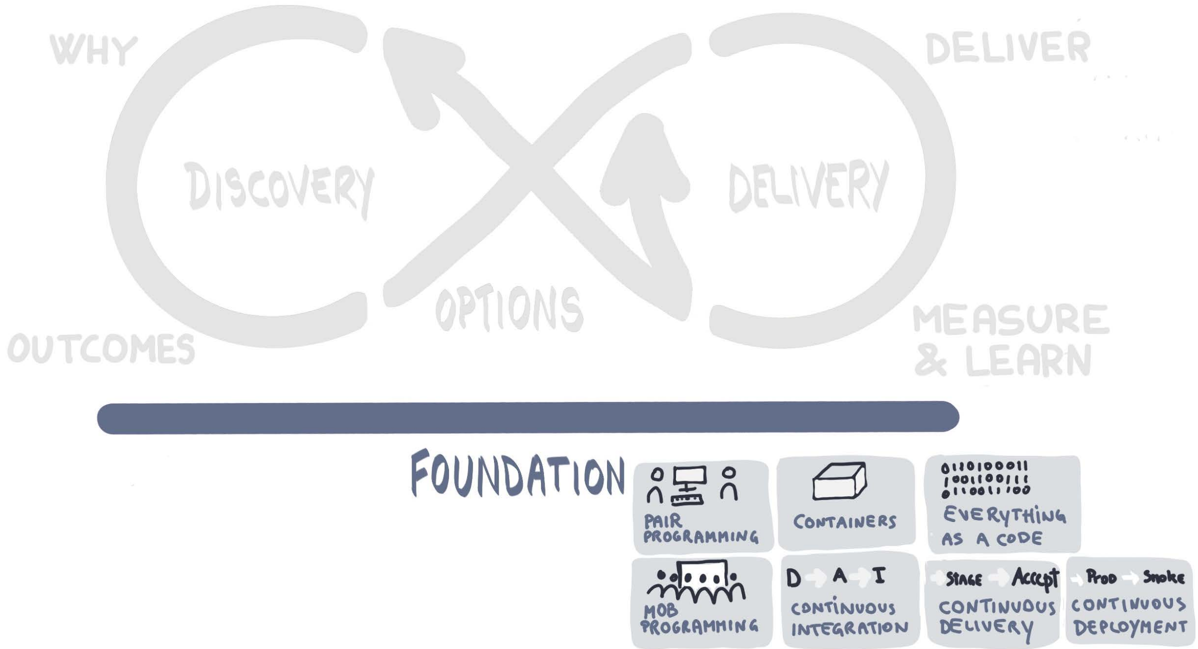
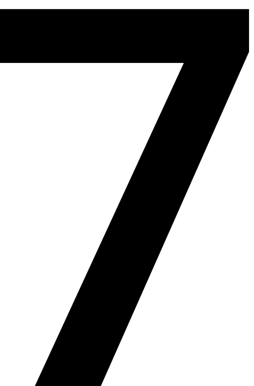


Figure 6.22: Adding technical practices to the foundation

In the second half of *Open Technology Practices*, we will learn about the bigger picture, discover what GitOps is all about, vastly improve our code quality through testing, and finish off with some lessons about our emerging architecture.



Open Technical Practices — The Midpoint

In this chapter, we are going to build on the foundational technical practices that we started in the previous chapter. We will acquire a shared understanding of our software delivery pipeline using the Big Picture practice. Even the less technical team members will be able to follow what happens to our software as it is being written and delivered.

We will then explain a technique that allows DevOps teams to deliver software changes using Git as the driving tool. The practice of GitOps leads to greater visibility of changes within our system, allowing the team to debug and resolve issues faster. We will explore how to improve our code quality through test automation and conclude this chapter by asking the question *How do we know if our architecture is good?*

This chapter will cover the following topics:

- The Big Picture
- GitOps
- Testing
- Emerging architecture

The Big Picture

An Open Technical practice that costs little to produce but is great in creating a shared understanding of part of a system is the Big Picture workshop. It is a simple practice used to visualize all the steps that a software pipeline goes through in moving code from source (for example, Git), through compile and test, and then into the hands of our happy users. Building it collaboratively is a great activity for a team to do as it helps to bridge the gap between techies and business folks. It's great for articulating the importance and sometimes the complexity of continuous delivery.

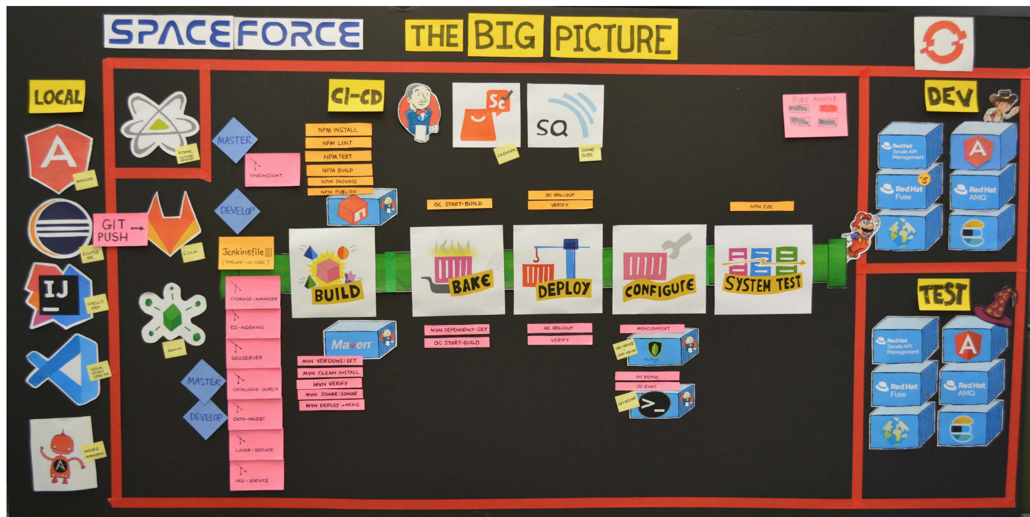


Figure 7.1: The Big Picture

A Big Picture can easily be created with just some stickies and a clear board or space. Of course, if you're feeling more artistic, it can also be doodled!

Figure 7.2: An example Big Picture

You may be reading this and thinking *Sounds fluffy to me – why should I bother to make one*? Here's why:

- **Shared understanding**: When the whole team collaborates around making the Big Picture, they get a shared sense of how their pipelines connect code to users.
- **Prototype quickly**: It's cheaper to write and draw before implementing a single line of code! Rapidly prototype with some markers and Post-Its, moving stages of your pen and paper pipeline.
- **Complexity simplified**: The Big Picture helps bring non-techies into the mix by showing them the components required to manage the software life cycle. Build it up one step at a time to demonstrate the complexity in a simple visual flow.
- **Information radiator**: Like all these practices, the Big Picture is an evolving artifact. As the complexity of a software delivery pipeline grows, the Big Picture should be updated to reflect this. It is a graphic that can be displayed to all and should not be hidden.

Figure 7.3: Collaborating to get a shared understanding of the Big Picture

Big Pictures can also be drawn using online collaboration tools. We used Miro to draw the following digital Big Picture online.

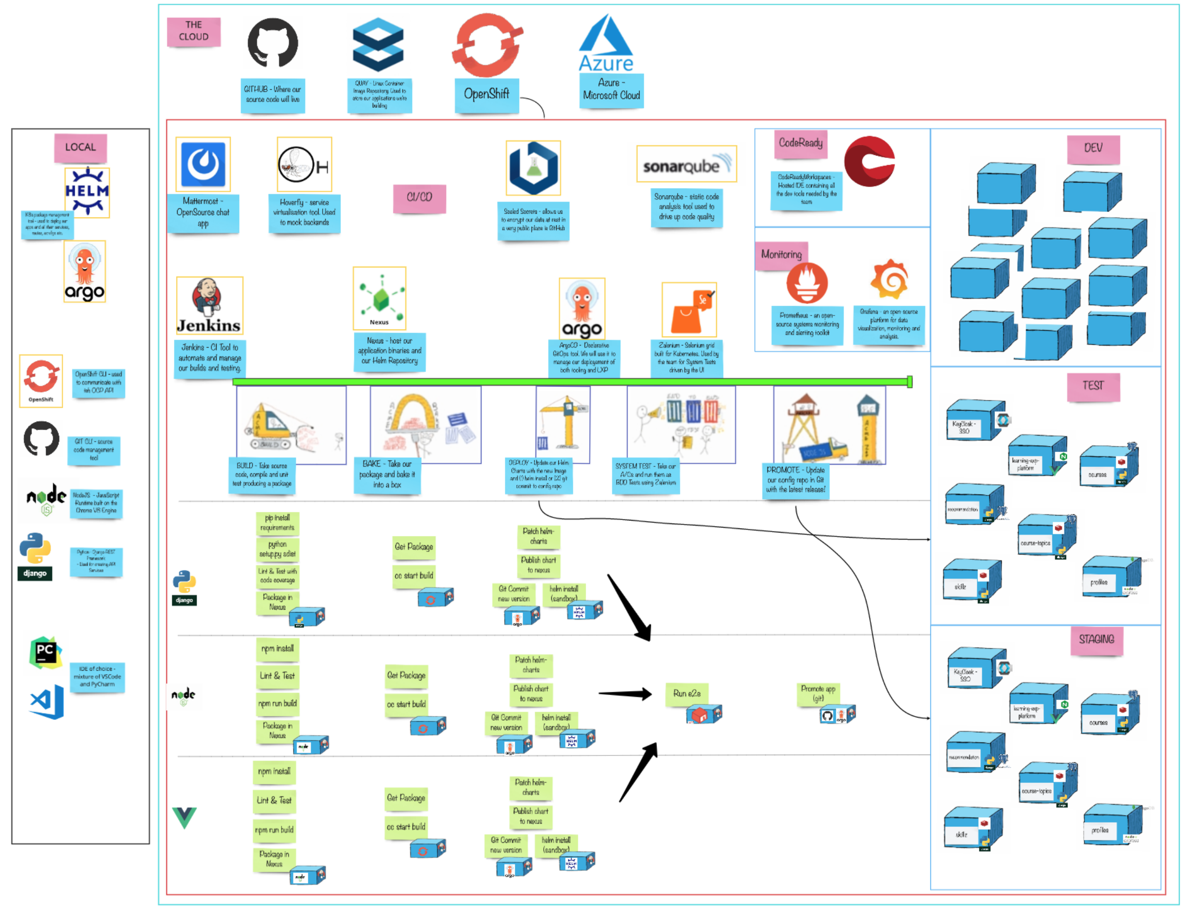


Figure 7.4: A digital Big Picture

The material required for this practice is fairly simple: some stickies, marker pens, painters' tape, and a big blank wall or canvas are all that's required, and these are fairly common things to have in our kit bags! There are a number of simple steps to follow in creating your Big Picture, but let's use our PetBattle example to show how a team might use this in practice.

PetBattle – Building a Big Picture

The PetBattle Techies decided to build a Big Picture to demonstrate their proposal for how the software should be built, tested, and deployed via some automation.

First, they invite all the others in the team to help explain some of the technology and complexity of the automation.

They use painters' tape to form a large box that represents the cloud and another box inside it to represent the OpenShift cluster they're going to use (deployed in the cloud). In this case, the metaphor is: OpenShift is just a big box where we can put some things running in the cloud. The box is so large that we can fill it with all the things we could possibly want, from sandboxes, to tooling, to production apps.

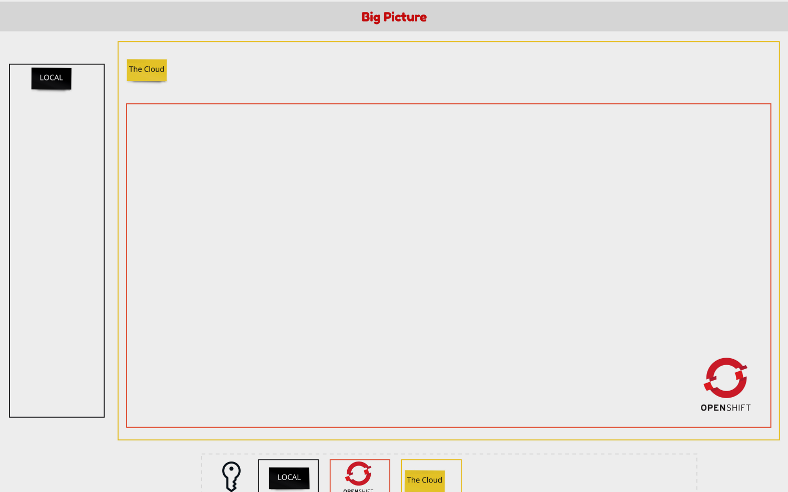


Figure 7.5: Starting the PetBattle Big Picture

They draw a box to the left side to represent their local development environment. This is their laptop for now, but it could also be a cloud-hosted IDE that the development team could write their code in that is deployed inside the cluster. One such product, CodeReadyWorkspaces, is a cloud-hosted IDE that runs in the OpenShift cluster that could be of great use to the team. Using an IDE like this allows us to further our everything-as-code practice by providing developers with their coding environment as a code artifact.

Next, they slice up the OpenShift cluster into smaller boxes. Each of these represents the OpenShift projects (or Kubernetes namespaces). We can think of these projects as rooms that separate one collection of applications from another. To keep things simple, the team decides on four namespaces initially:

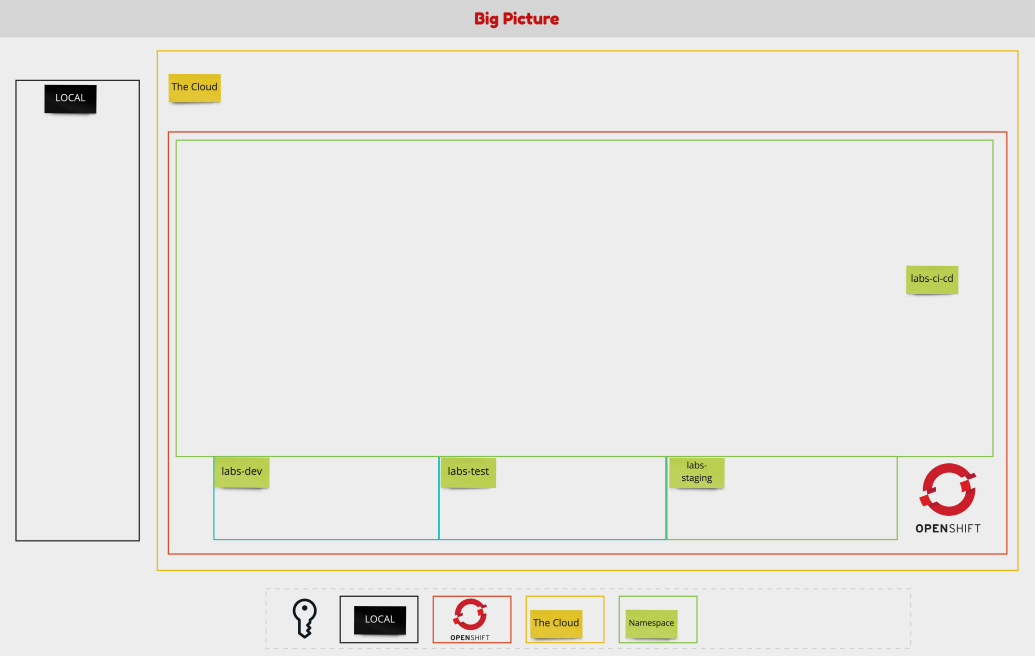


Figure 7.6: PetBattle Big Picture projects

- **Dev**: A sandbox project for the dev team to validate their app or get fast feedback from.
- **Test**: A project to deploy all our applications to and run our system tests against.
- **Production**: The project that PetBattle's customers will use to access the applications once they've cleared our tests.
- **CI-CD**: The project that houses all the tooling that supports **Continuous Integration** (**CI**) and **Continuous Delivery** (**CD**).

With the OpenShift cluster logically sliced up into the projects the teams will use, the team draws the tools they will use in each project.

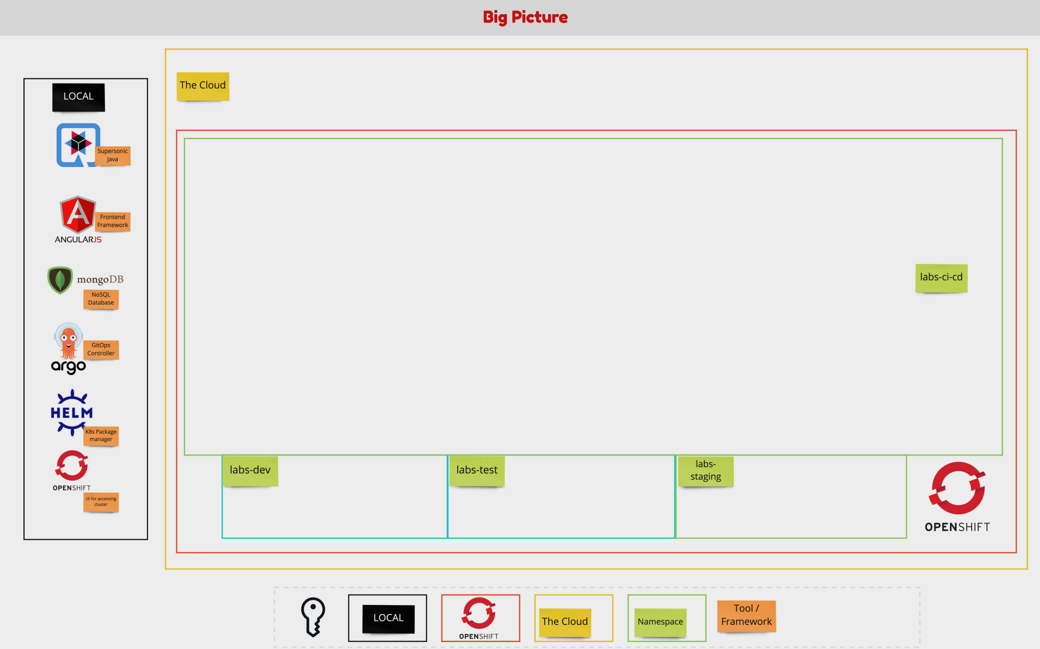


Figure 7.7: PetBattle Big Picture initial frameworks

Starting with their *local* development environment – in other words, their laptops or cloud-hosted workspace – the existing PetBattle is built using Angular (a JavaScript framework for building web apps) for the frontend. Quarkus (supersonic Java) is used for the API layer, and MongoDB for the persistence layer, so they add each of these tools to their workspace and write a one-line definition for how the tool or framework is being used by this team.

For PetBattle, we are going to use Helm to package up all the Kubernetes resources (Deployments, ConfigMaps, and so on) used to manage the application topology. We'll also use ArgoCD, a GitOps tool to manage our config-as-code.

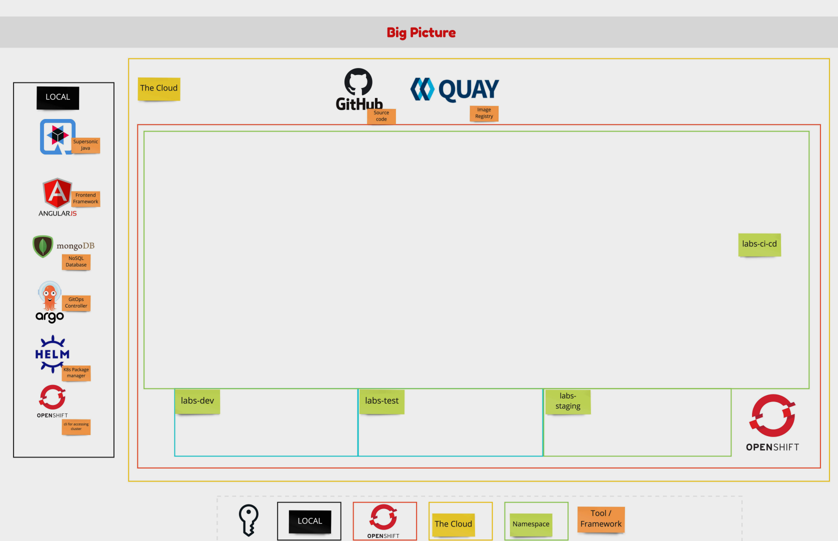


Figure 7.8: PetBattle Big Picture source code and registry

PetBattle will use GitHub to store its source code. When building images, it is likely the team will need to store the built image internally on the OpenShift cluster using the internal registry. The team also wants to make their images available externally and so have decided to also make use of Quay.io, an external registry hosted in the public cloud.

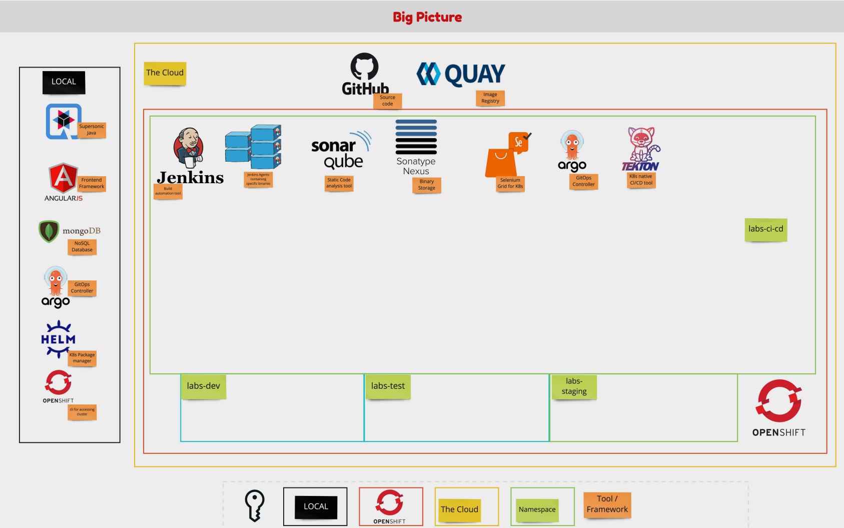


Figure 7.9: PetBattle Big Picture pipeline tools

The team then starts to add the tooling they will use to create their pipelines in their CI/CD namespace. They use more stickies to draw the tools and add a one-liner definition of what each tool is or how they will use it.

For example, the team is going to use Jenkins for their build and test automation. To store and cache application build dependencies and artifacts, the team opted to use the open-source artifact repository called Nexus. For Nexus, they add a simple one-liner to highlight the fact that it is used to house their software artifacts as well as their Helm repository. Shared understanding is key here, so it's important for the team to make sure everyone is aware what the purpose of each item is – this includes the product owner, designers, and all other interested parties. They don't need to be experts, but having an understanding of what the tools are used for can help them establish better empathy with the development team and see for themselves all the things needed to be able to ship code so quickly to users.

With some of the tools in place on the Big Picture, the PetBattle team can now start to implement the design they've put in place.

The Big Picture can be created in a physical room with lots of colorful sticky notes or with everyone distributed using a tool such as Mural, Miro, PowerPoint or Google Slides. We have provided a useful template with all the icons we use which should help you get started. You can download this from the book's GitHub repository.

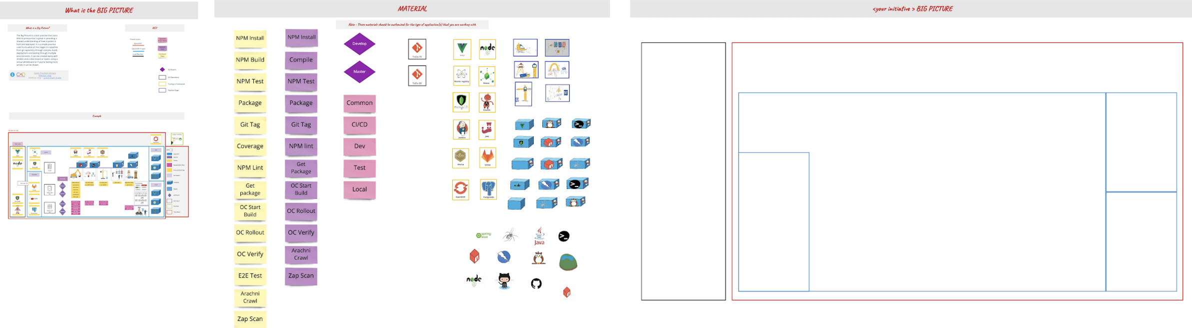


Figure 7.10: The Big Picture template

You can download this from the book's GitHub repository.

The Big Picture allows us to get a shared understanding and team alignment around the use of technical tools at a high level. Like all the practices we put in place in this book, the Big Picture is not a one-time thing. The Big Picture is a tool we will revisit and enhance as we add more complexity to our architecture and begin implementing our pipelines. We will continue to explore the Big Picture in *Section 6*, *Build It*, *Run It*, *Own It*.

You can learn more about, and collaborate on, Big Picture practices by going to the Open Practice Library page at https://openpracticelibrary.com/practice/the-big-picture/.

GitOps

Up to this point, we've talked about Git and the developer workflows available to our teams. We've spoken about everything-as-code, from infrastructure to tooling to applications all along the stack. Now, let's tie this all together with GitOps.

GitOps sounds a bit like a buzzword, as DevOps was when it was first coined. In fact, we heard someone describe it to us as DevOps for the year 2020. GitOps is a simple process of managing all of your systems, environments, and applications via Git. Git represents the single source of truth for all your applications, your tooling, and even your clusters. Changes to any of these things can be pull requested and discussed before an automated process applies them.

The difference between **infrastructure-as-code** (**IaC**) and GitOps is the approach to managing the configuration. IaC is agnostic to where you store the configuration; it could be on a flash drive in your drawer or it could be a shared drive in the cloud. GitOps, as the name suggests, means storing the full system specifications in Git.

The same principles hold true for IaC and GitOps – ideally, every action should be idempotent. Every action or operation can be applied multiple times, producing the exact same result. This is a very useful property in many situations, as it means that an operation can be repeated or retried as often as necessary without causing unintended effects. Configuration should be created declaratively. That is to say, you write the configuration to describe the desired state of an application or set of apps.

GitOps can be seen as a developer-centric approach to Ops. It teaches developers good practices around taking ownership of code once it leaves their machines and the approach to deploying and monitoring this code once it's running.

As developers, we hate repeating ourselves, so much so that we even have an acronym for it – DRY = don't repeat yourself! When encountering something that needs to be done more than once, our first instinct should be to try to automate it. Once something is automated or repeatable, the next step is simple. Check it into Git so that it can be audited, shared, and managed.

For example, whenever we want to deploy a new application to OpenShift, we could run some manual commands to spin up the application, create services and routes, and even bind a ConfigMap. But taking the time to create a Helm chart for this is reusable and repeatable. We can design the final state of the application in code and then check this into Git instead. This is a more cloud-native way of writing and managing our application code.

To implement a GitOps approach to our Helm chart example, all we need to do is connect a tool to the Git repository, which can be alerted or watch for changes coming through. When those changes arrive, this tool can assess the difference between what the current state is and what state is desired and apply the changes automatically for us. Enter ArgoCD.

ArgoCD

From ArgoCD's website, this is described as a tool that:

automates the deployment of the desired application states in the specified target environments. Application deployments can track updates to branches, tags, or be pinned to a specific version of manifests at a Git commit.[1]

When something is seen as not matching the required state in Git, an application becomes out of sync. Depending on how you have implemented your GitOps, ArgoCD can then resync the changes to apply whatever is in Git immediately or fire a warning to initiate some other workflow. In the world of Continuous Delivery as implemented by ArgoCD, Git is the single source of truth, so we should always apply the changes as seen there.

What types of things can ArgoCD apply? ArgoCD recognizes traditional Kubernetes YAML, Kustomize,[2] Helm, and all sorts of other things. Unlike Helm, which uses templating heavily, Kustomize allows you to take YAML files and emits text in a template-free declarative way. You can patch Kubernetes resources and use folder-based structures to apply what is termed an overlay or YAML override, which emits text, leaving the original YAML untouched. For our purposes, we will stick to Helm and a little bit of Kustomize where appropriate.

ArgoCD is another tool (and there are others like it, such as Flux) in the long list of tools that we need to be able to implement CI and CD. Unlike Jenkins, which we could also use to manage our application deployments, ArgoCD is specialized and very good at managing and maintaining just our deployments.

1 https://argo-cd.readthedocs.io/en/stable/

2 https://github.com/kubernetes-sigs/kustomize

Jenkins could apply our Helm charts in a done once and finish sort of way. It doesn't have the capability to keep watching our Kubernetes resources to ensure the desired state in Git stays that way in our clusters. If someone decides to change something in the cluster, for example, add a new environment variable to a running application, ArgoCD will detect that change and overwrite it. This means no more one-of-a-kind deployments or manual tweaks once they're deployed.

ArgoCD enables teams to enforce this golden rule – if it's not in Git, it's not real. This is perfect for audit tasks – all you have to do is check the Git log to see who committed and pushed the code.

If It's Not in Git, It's Not Real!

We worked on a virtual residency with the World Health Organization helping to manage the COVID-19 crisis. We were building a new platform to help educate people in the field and disseminate information faster. We decided to use GitOps to do Continuous Delivery and, in particular, used ArgoCD to manage our Helm charts.

To do cloud-native properly, namespaces and environments should be ephemeral. We should be able to recreate everything of use to us from their description in code. This includes namespaces, quotas, and role bindings, as well as applications and databases. To prove this in one sprint, we created a cleanup job that would delete the dev and test projects in OpenShift. Our configuration repository was linked to ArgoCD, which watched the cluster and, if anything changed, it was set to reapply the resources as described in Git. The time for this job to execute was on a Wednesday afternoon around lunchtime, about an hour before our sprint review. What could possibly go wrong?

The team got ready to do their demo as normal, but about 20 mins before showtime, one team member called out that the build was failing and his demo was broken and he could not figure out why. The team scrambled

to sort the issue, with everyone jumping on a call and mobbing around the problem. Rewinding what could have changed in the past hour, the only thing that had executed was the cleanup job we had written. We immediately thought we'd written something incorrectly with our job and so went to debug it, but it was fine. The next step was to look more closely at the build and the failure message.

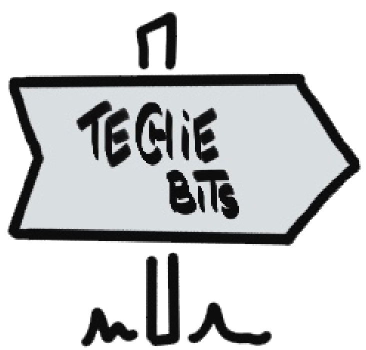
At this point, we discovered someone on the team had manually deployed a database to the dev environment. They were connecting to it for their demo AND using it as the test database in our Jenkins Pipelines. Essentially, someone on the team had created a Pet – a hand-reared server that was cared for and nurtured by one person and not known about by the rest of the team. In the world of ephemeral environments, what we really want is Cattle. Cattle are mass-produced, created using automation, and killed off when no longer required. Hence, when our job ran to clear out the project, all resources were destroyed.

The team learned a valuable lesson in this experience which they made very visible:

Figure 7.11: If it's not in Git, it's not real

It gave rise to a mantra we added to our social contract from *Chapter 4*, *Open Culture*.

Implementing GitOps



Let's build the big picture with some real working code! In this section, we are going to take a technical detour! Prepare for some code snippets. If this is not your thing, feel free to skip over it to the next section all about testing! We'll mark any section that's going to have code snippets with this handy sign.

Let's explore ArgoCD and create the components of our Big Picture from code. To do this, we will first explore a sample project that can be used as a starting point for development.

At Red Hat Open Innovation Labs, we have automated the bootstrap of Labs Residency CI-CD tooling to accelerate setup and onboarding. The code repository is called Ubiquitous Journey, so it makes sense for us to start here. We will explore this repository and set up our technical foundation using it. In later sections of the book, we will extend it with new technology and tools. This repo is available on the PetBattle GitHub organization – https://github.com/petbattle/ubiquitous-journey.

The first task we would normally perform on our OpenShift cluster when deploying Jenkins is to create a new project using the command line. We could follow this manual approach again, adding in role bindings and quotas for our project, and repeat these steps for each bit of our Big Picture. But let's do it in a way that honors our everything-as-code practice.

From your laptop, fork the sample project and open it up in your favorite code editor.

We are going to make changes to our project so maintaining your own copy of it is necessary for GitOps. From here on out, when we encounter a new repo, you'll probably find it easier to fork it so you can make changes to it. For the purposes of the book going forward, we will continue using PetBattle organization so feel free to equate this to your own organization or user.

```
git clone https://github.com/petbattle/ubiquitous-journey.git
```

The Ubiquitous Journey project is broken down into two main components (some of the files are removed from the breakdown below for simplicity), Bootstrap and Ubiquitous Journey. If you're wondering why we named the project Ubiquitous Journey... well, we didn't! We hit the **generate random name** button on GitHub and this is what it chose for us. As is the case with most things in software, naming things is hard! We did plan on renaming the repo at some stage, but now the name has kind of stuck and we like it!

```
$ tree ubiquitous-journey
ubiquitous-journey
├── argo-app-of-apps.yaml
```

```
├── bootstrap
│   ├── charts
│   ├── Chart.yaml
│   └── values-bootstrap.yaml
├── docs
├── ...
├── README.md
└── ubiquitous-journey
    ├── Chart.yaml
    ├── templates
    │   ├── argoapplicationdeploy.yaml
    │   └── _helpers.tpl
    ├── values-day2ops.yaml
    ├── values-extratooling.yaml
    └── values-tooling.yaml
```

The bootstrap folder contains a Helm chart definition with a values-bootstrap.yaml file and Chart.yaml manifest. There are no templates for this chart because it's actually just a wrapper for other Helm charts. If we look at the Chart.yaml manifest, we can see that it has a dependency of the ArgoCD chart, another called bootstrap, and a helper chart called sealed-secrets. The bootstrap folder Helm chart is acting as a wrapper chart, allowing us to control the variables we pass to these dependencies. In this case, our variables are stored in the `values-bootstrap.yaml` file:

```
bootstrap-project:
  enabled: true
  ci_cd_namespace: &ci_cd "labs-ci-cd"
  pm_namespace: &pm "labs-pm"
  ops_namespace: &ops "labs-cluster-ops"
  dev_namespace: &dev "labs-dev"
  test_namespace: &test "labs-test"
  staging_namespace: &stage "labs-staging"


  bindings: &binds
  # this labs-devs is the GROUP NAME in IDM
    - name: labs-devs
      kind: Group
      role: edit
  # this labs-admins is the GROUP NAME in IDM
    - name: labs-admins
      kind: Group
      role: admin
    - name: jenkins
```

```
        kind: ServiceAccount
        role: admin
        namespace: *ci_cd

    namespaces:
      - name: *ci_cd
        bindings: *binds
      - name: *pm
        bindings: *binds
      - name: *ops
        bindings: *binds
      - name: *dev
        bindings: *binds
      - name: *test
        bindings: *binds
      - name: *stage
        bindings: *binds
```

The bootstrap chart is responsible for creating the listed projects in our OpenShift cluster. In the example, these are labs-ci-cd, labs-dev, labs-test, labs-staging, labs-pm, and labs-cluster-ops. Dev, Test, Staging, and CI/CD will hopefully be self-explanatory; if not, take a look at the previous chapter, where we discussed CI/CD in depth. The labs-pm namespace is for deploying other project management tools (for example, collaboration tools such as etherpad). The labs-cluster-ops namespace is used for operational jobs and tasks.

Resources in OpenShift have **role-based access control** (**RBAC**) applied.[3] RBAC determines whether a user is allowed to perform a given action within a project. We bind the listed user groups to the service accounts within these projects. Don't worry if your cluster does not have the labs-dev and labs-admin groups set up right now. It is enough if you are logged in to your cluster with a user who has cluster admin privilege.

```
  argocd-operator:
    enabled: true
    name: argocd
    namespace: *ci_cd
   argocd_cr:
      applicationInstanceLabelKey: petbattle.app/uj

    # operator manages upgrades etc
    version: v1.8.6
```

3 https://docs.openshift.com/container-platform/4.6/authentication/using-rbac.html

```
    operator:
      version: argocd-operator.v0.0.14
      channel: alpha
      name: argocd-operator
```

The second part of this file overwrites some variables in the ArgoCD chart. This Helm chart installs the ArgoCD operator and configures it with sensible defaults. For a list of all the possible variables that could be passed to this chart, you can check out the Operator Docs for ArgoCD – https://argocd-operator.readthedocs.io/en/latest/. There is no point in recreating those docs in this book, but it's useful to have them saved if you want to do some exploring.

It is worth calling out the `applicationInstanceLabelKey` variable. This needs to be unique for your cluster. If you deploy more than one instance of ArgoCD to a cluster with the same instance label, the two ArgoCD instances will try to manage the same resources and then they'll fight over who actually owns them and get you into a world of pain, so make sure the `applicationInstanceLabelKey` is unique!

Let's deploy this setup and see what it gives us. If you wish to change the names of the projects that are created, you can edit the values file, but for now we'll use the defaults. In a terminal on your laptop, try the following command:

```
$ helm template bootstrap --dependency-update -f \
bootstrap/values-bootstrap.yaml bootstrap
```

Running a Helm template like this should bring down our chart dependencies and process our templates. This can be a handy way to validate that the YAML file looks as we expect. Let's install the bootstrap Helm chart into its own namespace.

```
$ helm upgrade --install bootstrap-journey \
  -f bootstrap/values-bootstrap.yaml \
  bootstrap --create-namespace --namespace labs-bootstrap
```

The output of this command should be a successful installation of the bootstrap Helm chart:

```
/bin/bash
/bin/bash
virt:~/git/pb-ci-cd/ubiquitous-journey ⎇ main#179a27d$ helm upgrade --install bootstrap -f bootstrap/values-bootstrap.yaml boots
trap --create-namespace --namespace labs-bootstrap
Release "bootstrap" does not exist. Installing it now.
NAME: bootstrap
LAST DEPLOYED: Tue Feb 16 09:56:52 2021
NAMESPACE: labs-bootstrap
STATUS: deployed
REVISION: 1
TEST SUITE: None
virt:~/git/pb-ci-cd/ubiquitous-journey ⎇ main#179a27d$
```

Figure 7.12: Bootstrap ArgoCD using Helm

You can check the pods coming up with:

```
oc get pods -n labs-ci-cd
```

You should start to see the ArgoCD server start to come alive after a minute or two:

```
ubiquitous-journey — donal@dspring-mac — ..itous-journey — -zsh — Solarized Dark ansi — 132×10
#( 11/10/20@ 4:19pm )( donal@dspring-mac ):/tmp/ubiquitous-journey@master✔
[   oc get pods -n labs-ci-cd                                                                      ]
NAME                                             READY   STATUS              RESTARTS   AGE
argocd-application-controller-85575b7ccb-5rk4q   0/1     ContainerCreating   0          15s
argocd-dex-server-5d5f797794-4bmtc               0/1     Init:0/1            0          15s
argocd-dex-server-7f86bd574f-fl5fz               0/1     Init:0/1            0          15s
argocd-operator-5974776b76-fnp6k                 1/1     Running             0          4m31s
argocd-redis-6f7cfddbcb-7fm5x                    0/1     ContainerCreating   0          15s
argocd-repo-server-69fdf5f445-z5zbf              0/1     ContainerCreating   0          15s
argocd-server-78454cc9db-4nrp8                   0/1     ContainerCreating   0          15s
```

Figure 7.13: Labs-ci-cd namespace pods starting up

Or, if you have a look in the UI, you should see the topology with all the components of ArgoCD:

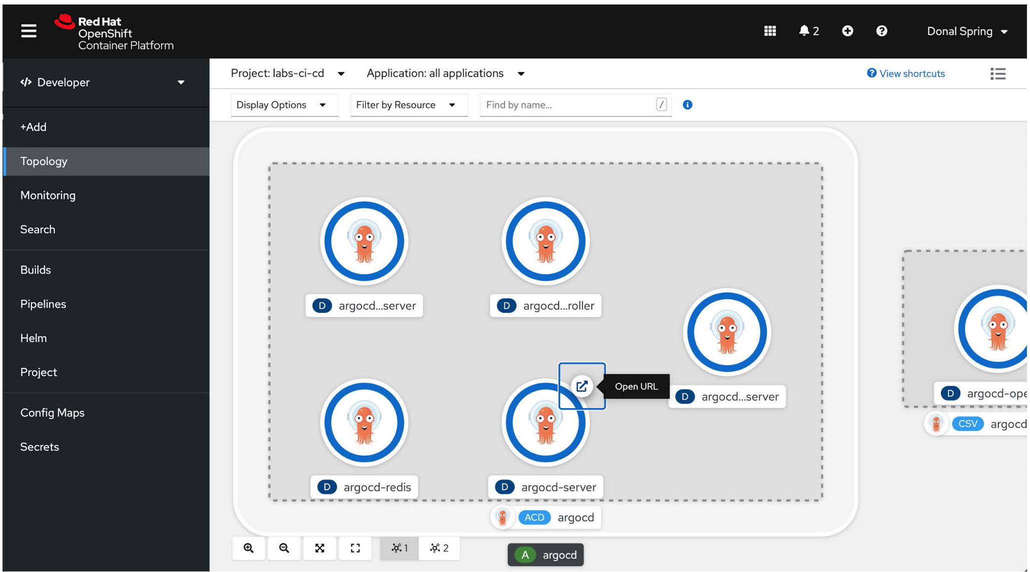


Figure 7.14: OpenShift developer topology view of the labs-ci-cd project

Let's take a look at ArgoCD by clicking the link in the UI, or you can get the URL from the command line using the following command:

```
oc get routes argocd-server -n labs-ci-cd
```

Log in with your OpenShift credentials. We should see an empty ArgoCD instance:

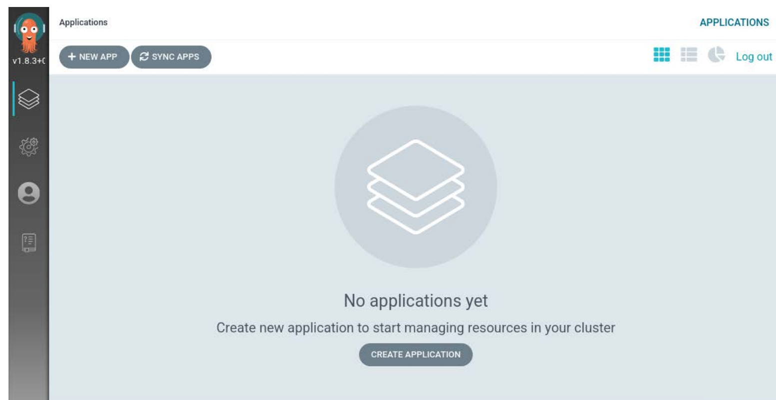


Figure 7.15: Empty ArgoCD instance from the web interface

At this point, we should ask ourselves the question *What happens when someone changes the bootstrap configuration values for our cluster*? for example, to add more projects or change the roles or groups? Can we do this in an automated and tracked way, in other words, using GitOps? Fear not, ArgoCD to the rescue! We can now point ArgoCD to the Git repository we've been working on.

We can create an ArgoCD application from the ArgoCD web interface by selecting **+New App** -> **Edit as YAML** and copying and pasting the following definition:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: bootstrap-journey
  namespace: labs-ci-cd
spec:
  destination:
    namespace: labs-bootstrap
    server: https://kubernetes.default.svc
  project: default
  source:
    helm:
      parameters:
      - name: argocd-operator.ignoreHelmHooks
        value: "true"
      valueFiles:
      - values-bootstrap.yaml
    path: bootstrap
    repoURL: https://github.com/[YOUR FORK]/ubiquitous-journey.git
    targetRevision: main
  syncPolicy:
    automated: {}
```

Hit **Save**, followed by **Create**. You should see the `bootstrap-journey` application synced:

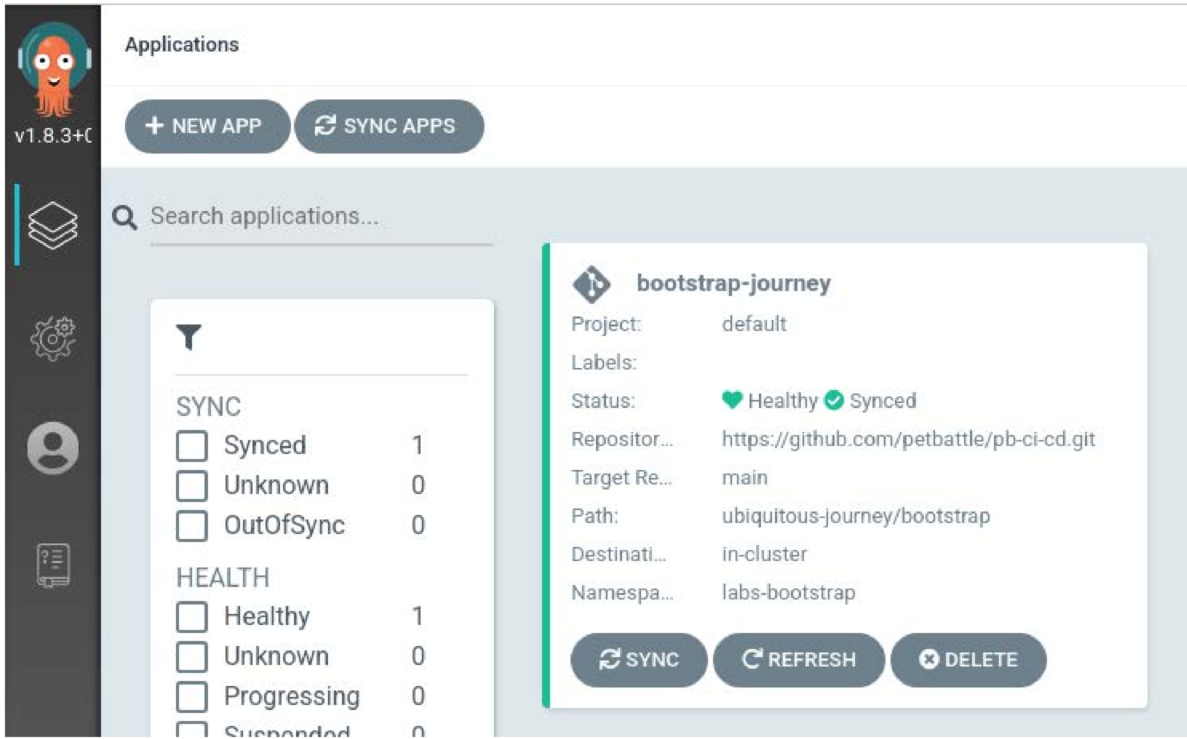


Figure 7.16: Bootstrap ArgoCD application from the web interface

We can also create the same application on the terminal. You can log in using single sign-on to OpenShift from the terminal using this one liner. It requires a terminal that is not headless, in other words, it can connect to your screen and browser:

```
$ argocd login $(oc get route argocd-server --template='{{ .spec.host }}' \
-n labs-ci-cd):443 \
--sso --insecure
```

Create the new app and sync the changes. With this in place, argocd will now actively track changes to our Git repository and roll them out for us! Simple!

```
argocd app create bootstrap-journey \
  --dest-namespace labs-bootstrap \
  --dest-server https://kubernetes.default.svc \
  --repo https://github.com/[YOUR FORK]/ubiquitous-journey.git \
  --revision main \
  --sync-policy automated \
  --path "bootstrap" \
  --helm-set argocd-operator.ignoreHelmHooks=true \
  --values "values-bootstrap.yaml"
```

You can select the application in the web interface to drill down into it:

Figure 7.17: Bootstrap application details from the web interface

Excellent – we are on our way to completing our Big Picture as code and laying down our technical foundation! We've created the projects and added the first tool, ArgoCD, to our kit bag. Now, let's take it a step further and fill our cluster with some of the applications we think would be initially useful for building out CI/CD pipelines. At the beginning of any project, this will usually be a best guess. As we start to build out the product, we must continuously evolve the toolset we use. This is not a one-time process; it's a set of tools that need to be extended when required or trashed if no longer useful. The important thing here is to ensure that things are deployed in a repeatable way.

Let's add some tooling. Open your editor on the ubiquitous-journey project. Inside ubiquitous-journey/values-tooling.yaml, we have some useful variables referencing Helm charts ready for us to pick from, including Jenkins, which we manually deployed previously!

```
##############
# 🛎 Argo App of Apps declaration
#############
# enabled true on an app is to tell helm to create an argo app cr for this
item
# Custom values override the default values in Helm Charts
applications:
  # Nexus
  - name: nexus
    enabled: true
    source: https://redhat-cop.github.io/helm-charts
    chart_name: sonatype-nexus
    source_path: ""
    source_ref: "0.0.11"
    sync_policy: *sync_policy_true
    destination: *ci_cd_ns
    ignore_differences:
      - group: route.openshift.io
        kind: Route
        jsonPointers:
          - /status/ingress
  # Jenkins
  - name: jenkins
...
  # Sonarqube
  - name: sonarqube
...
```

The layout of this file is simple. For each item in the applications array, it expects to find a Helm chart or a reference to a Git repository with some Kubernetes yaml (or Kustomize) at a particular version.

When using Helm, any overrides to the defaults supplied by the chart can be added here, but for the Nexus chart shown, we are using the default values, so there is no need for value overrides for Nexus. There are other fields for each application, and these are mostly related to the operation of ArgoCD. For example, you can configure the application synchronization policy – sync-policy – which tells ArgoCD to always keep your application synced when set to automatic. The destination namespace may be specified. With some Kubernetes and OpenShift API objects, ArgoCD needs to be asked to ignore differences it finds; this is particularly true when controllers and operators write back the status and other fields into the objects themselves. We have found over time that each release of ArgoCD lessens the need to specify these *ignores* as the generated differences are taken care of automatically.

The other important field for each application entry is the `enabled: true | false` – it's easy to run down the list and enable the tools we know we need straight away. For now, we are going to start with just four tools: Jenkins, Nexus, Tekton, and Code Ready Workspaces. These are the bare bones for scaffolding our application and pipelines. At this point, it is worth mentioning the other two `values` files, `extratooling` and `day2ops`:

```
└── ubiquitous-journey
    ├── Chart.yaml
    ├── ...
    ├── values-day2ops.yaml
    ├── values-extratooling.yaml
    └── values-tooling.yaml
```

Like our CI/CD application list in `values-tooling.yaml`, they contain references to useful Helm charts and YAML files for deploying in our cluster. The extra tooling contains project management and collaboration tools, while the `day2ops` contains useful prune jobs to keep our cluster tidy. For now, we will disable all of the extra tooling and day2ops apps. This gives us a minimal setup to get started with.

If you are running CRC, please check the Appendix for any details prior to deploying the tooling. Let's deploy these tools from the command line using Helm and oc:

```
$ helm template -f argo-app-of-apps.yaml ubiquitous-journey/ \
| oc -n labs-ci-cd apply -f-
```

If you check the ArgoCD web page, you should now see these applications begin to deploy and synchronize into your cluster. It will take some time for them all to synchronize completely. Jenkins, for example, builds all of the default agent images that we may need for running pipeline jobs.

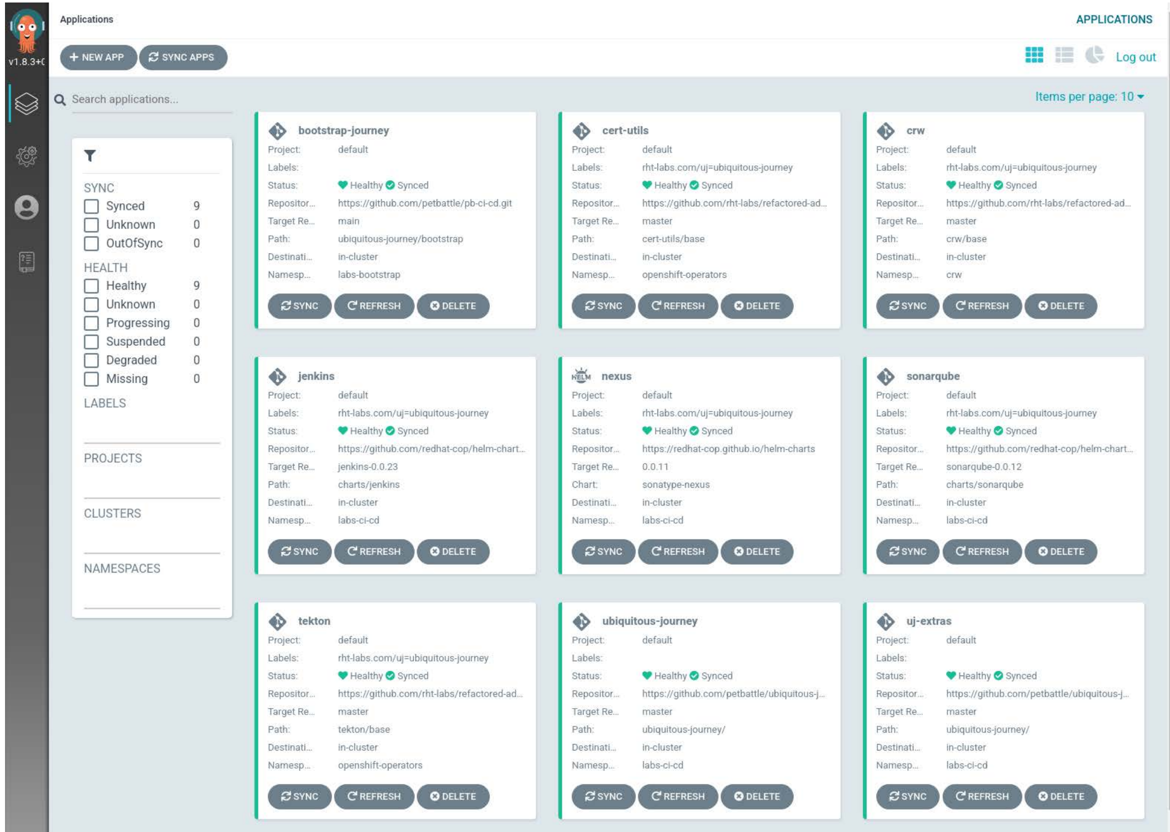


Figure 7.18: The complete picture with all our tools installed

We have now successfully bootstrapped our CI/CD tooling! We will revisit these configurations as we find we need to add and update the tools we need to develop, test, and deliver PetBattle. By practicing *everything-as-code*, we can easily redeploy these tools into any Kubernetes cluster, track changes we may make, and manage the life cycle of the tools (upgrade them as their versions and features change).

Testing Testing Testing!

Up to this point, we've spoken about some of the tools we can use to move application code from ideas through compilation and into deployment. But how do we know the stuff we've built is actually working as we expect it to? If we create a pipeline that just compiles code and moves it to production – is it done? No, there are testing quality steps and gates that we need to introduce into our software pipelines!

The Test Automation Pyramid

How do we know our feature works as we expect it to? We should test it and see! It is not always clear how we should test our feature, nor is it clear when we have done too much or not enough testing. Should we create test instructions and manually test the feature? Should we test the feature in isolation? Should we test all its constituent parts or just the whole thing? What is a definition of a unit test exactly?

Let's face it, testing is complicated. We are going to advocate for creating not just any tests, but automated tests! *The Test Automation Pyramid*, authored by *Michael Cohn*, is a good starting point for us moving through the world of automated testing. Let's take a simplified look at the *traditional* test automation pyramid by the original author:

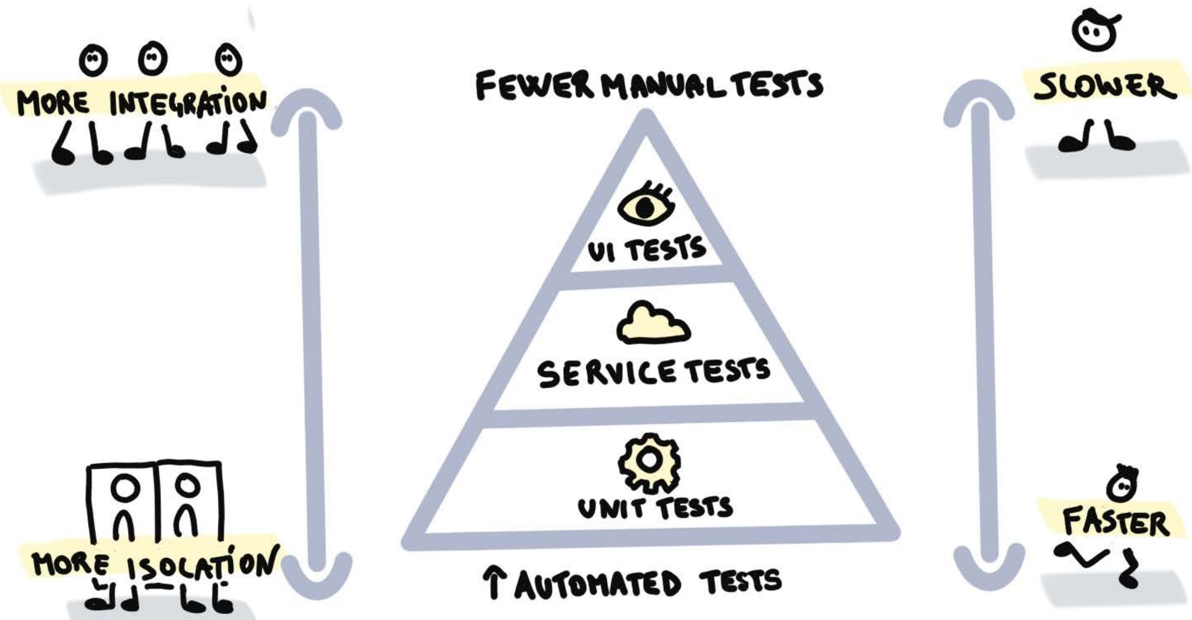


Figure 7.19: The testing triangle

In the standard three-tiered testing triangle, the things at the bottom (listed above as **UNIT TESTS**) are the things we should do more of. Unit tests are the smallest amount of code we can test within an application. These units should have little to no dependency on other items and so when we run them, they give us immediate and precise feedback. Unit tests should point us exactly to where the problem is in our code. Moreover, the thinking here is that unit tests are cheap to write, easy to maintain, and fast to execute. Therefore, we want more of them. This is why they sit at the base of the testing triangle.

Service tests are sometimes seen as integration tests and are the next level up the testing triangle. These are API tests that are validating the services within your application behave as expected. This may include single service calls, as well as chains of service calls, when one service calls another service. The width of the testing tier in the triangle relates to how many types of a particular test there should be in your code base. According to the pyramid, we should have fewer of these service tests than unit tests as they can be costly to execute.

The top tier of the testing triangle is reserved for **User Interface** (**UI**) tests, or end-to-end system tests. These are responsible for validating that the system, as the sum of its components and parts, is behaving as expected. Often, UI tests can be brittle in the face of change, break more often, and require maintenance to keep them relevant, so the rationale from the testing pyramid is that we should do fewer of these as they are difficult to perform and provide less feedback for us.

Testing in Practice

The Test Automation Pyramid is a great place to start when thinking about your own testing. As with all models and patterns, people have over-simplified some of its original meaning. In fact, if you do an image search for the testing pyramid, you'll see that most of the results are missing the most important keyword – automation! Often, organizations lose sight of this, and they think doing manual testing for these tiers is good enough.

Testing is important; in fact, it's critical to being able to deliver at speed! If you imagine not investing the time into writing automated tests, it may be possible to complete a sprint without breaking things. It's probable that we'd be able to do two sprints and not break things. However, once we hit that third or fourth sprint, that's when your software system starts to misbehave. Applications that were written in sprint one now have bugs appearing in them because their functional behavior does not work as intended. Functions and APIs that were thought to be working were, in fact, completely broken! Being able to release software at speed is one thing, but being able to release quality software at speed is the differentiator.

What is important when thinking about testing is to apply context. You don't have to blindly follow a model such as the testing pyramid. In fact, it's a good place to start from, but it's not a golden hammer to apply in all environments. For example, you might be building a web app with static content or third-party services, so UI testing is probably the most important thing.

What is important is to be sensible about the types of testing you're aiming to perform and the value they provide. You may find that it's more important to your product that covering the services layer is a better option. If you don't have access to the code, then writing black-box tests that assess the services with well-defined inputs and outputs is

more appropriate to your quality control. Likewise, measuring the number of tests, as suggested by the pyramid, tells us nothing about the quality of the tests. Good quality tests catch errors before your user does. When there is a failure in production, or a bug raised by a user, it is very likely that you need to write some more automated tests.

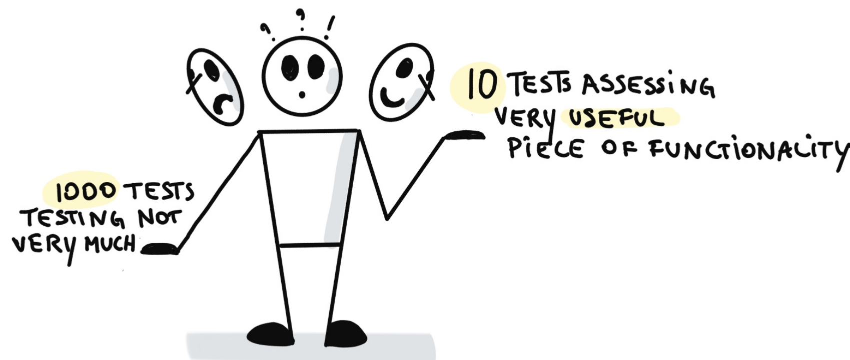


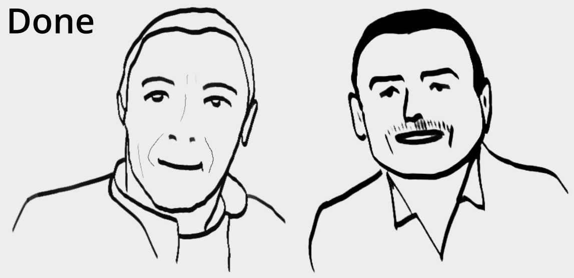
Figure 7.20: Measuring tests

The other way to look at this would be to calculate the risk of not testing a piece of functionality. Perhaps the application you're writing is a one-time throwaway or just a simple technical spike that does not require rigorous testing. However, if a piece of functionality within your product is used all the time and it has no automated tests written for it at all, this could be a good place to focus your automated testing efforts.

Create a culture within your product development team where testing is a continual practice. Testing should not be an afterthought of the development process. All too often, we see testing beginning once the development team throws a package over a wall to the test team for some testing to begin. For us, every item in a sprint will always have some level of testing. This isn't done by some third-party team, but by the engineers themselves. Developers will often favor unit and integration testing, but **quality assurance** (**QA**) teams will often favor automated UI testing to validate the application from a user's point of view. Sometimes, if the culture is not right and a team is being squeezed to turn out new features, testing quality can drop, leading to an almost inverted testing pyramid: a few unit tests on the bottom, followed by a few more service tests, and then a load of brittle UI tests sitting on top! This has an effect on the quality of the software delivery pipelines. The feedback loop from Dev to QA can be very long, with little to no value from unit tests and expensive UI tests that are not providing feedback quickly enough.

Decreasing the quality by inverting the testing pyramid during delivery can be very damaging to a team. If the volume of defects increases markedly, trust in the team will falter. If there is no trust in the team, then autonomy could be the next thing to break, leading to a heavy command-and-control-driven culture. Teams operating in this way will very quickly fail and top talent will leave.

Testing and the Definition of Done



While working on a residency recently with the World Health Organization, we started out with great ambition to write tests for each sprint item. We got off to a good start by including testing in our Definition of Done for each sprint item. We agreed that there must be some automated testing for each item being taken into a sprint.

The first sprint went by in a flash as we were working in one-week iterations. As we were a new team, everyone was motivated and keen to try new things. By sprint three, we were taking on more stories than we could get through with our team's capacity.

The work we were doing was becoming more complex and we missed out some of the automated tests. We claimed a feature or two were done. In the demo for that week, we admitted to the product owner that the piece of work was functionally done, but not done according to our own criteria.

We tried to be honest with ourselves, but found we'd slipped up again the following week. At this point, it became clear to us that when we did sprint planning, we were not thinking correctly about the capacity required for writing tests. The Definition of Done was in place, but we were still not being honest. We were a team that was keen to keep moving forward and picking up new items before previous ones were done.

In a retrospective session, we decided a good way forward would be for us to capture testing effort when writing our tasks. When taking an item from the backlog, we would add subtasks for all of the automated testing. This way, all of the work associated with test automation became visible to the team because these subtasks were on the sprint board. Having a task to write tests for your feature makes it pretty hard to move on to the next item when it's still in progress!

You can learn more and collaborate on CI practices by going to the Open Practice Library page at https://openpracticelibrary.com/practice/test-automation/.

TDD or BDD or DDT

There are a number of books written on testing and how to write great tests that are meaningful and provide value. Our ambition is not to rewrite these books, but to give you pointers to things you could research further if this topic really interests you. Some approaches to testing that teams find useful at the various levels of the triangle are things such as **Behavior-Driven Development** (**BDD**), **Test-Driven Development** (**TDD**), and **Developer-Driven Testing** (**DDT**).

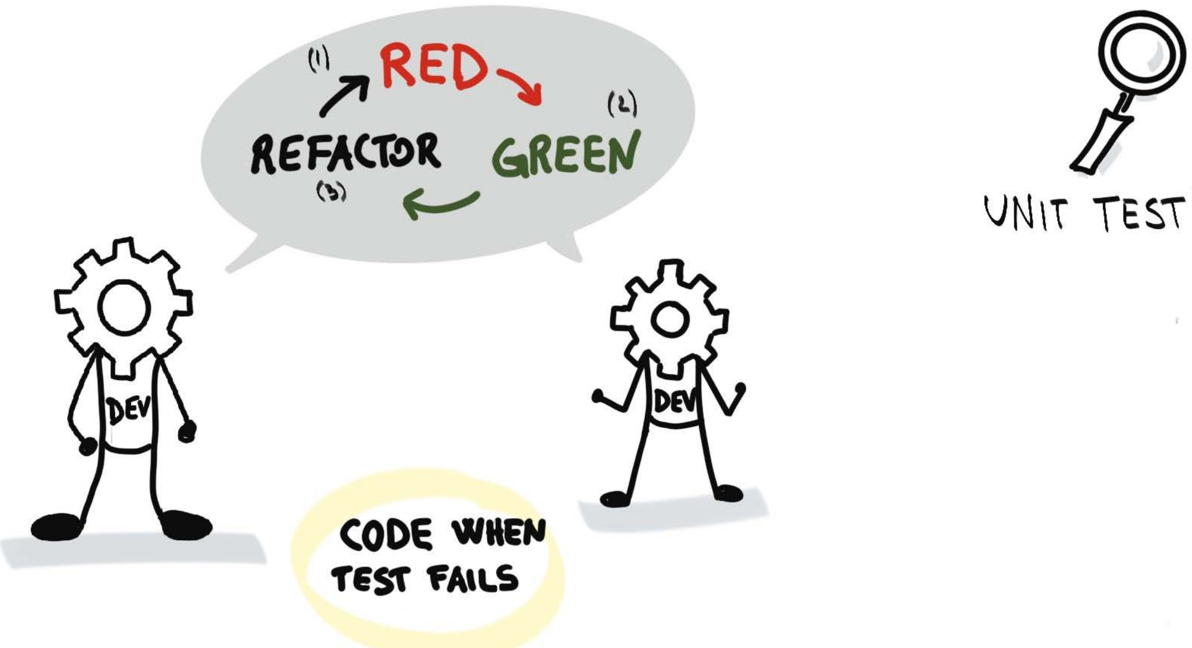


Figure 7.21: Test-Driven Development

TDD is a simple process, yet somewhat misunderstood by some teams. The process is fairly simple. Start off by writing some tests for the functionality you're building. At this point, they should fail (**RED**). If they don't fail, then your tests are not very well written OR the functionality already exists! A developer will then write the code to make the test pass (**GREEN**). With the tests now green, refactoring can take place or, as Kent Beck, an American software engineer and the creator of extreme programming, puts it, *refactor to remove duplication*. Remove duplicate code or make the code leaner and tidy it up while maintaining the green state of the tests. The process is simple: **Red > Green > Refactor**. Writing tests first is a hard practice to do and takes time and perseverance to get the skills right, but it can lead to less spaghetti code. Because the tests are written first, they lead the design and implementation of the code.

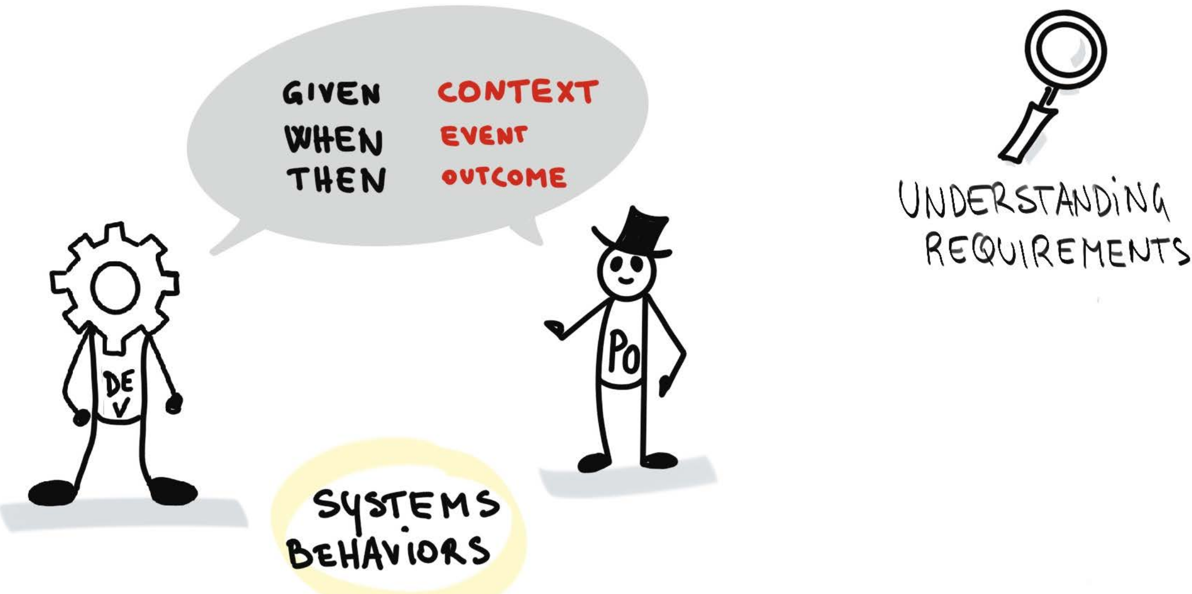


Figure 7.22: Executing Test-Driven Development

A great exercise to do with teams wanting to try TDD without code is to do the Lego TDD simulation on Gargoyle Software's website: http://gargoylesoftware.com/articles/lego_tdd.

You can learn more and collaborate on TDD by going to the Open Practice Library page at openpracticelibrary.com/practice/test-driven-development.

DDT is easy and probably the place to start if you're not writing any tests. The important point here is that some tests are being written! DDT focuses on the developers writing code as well as writing the tests. Simply put, the developer codes for a bit, writes some automated tests, and then goes back to coding and testing. This might sound a bit like TDD, but the key difference is the order. Code first and then test, resulting in the code influencing the tests as opposed to the tests leading the software design. The objective of DDT is that developers need to own their code and that everyone should be responsible for testing.

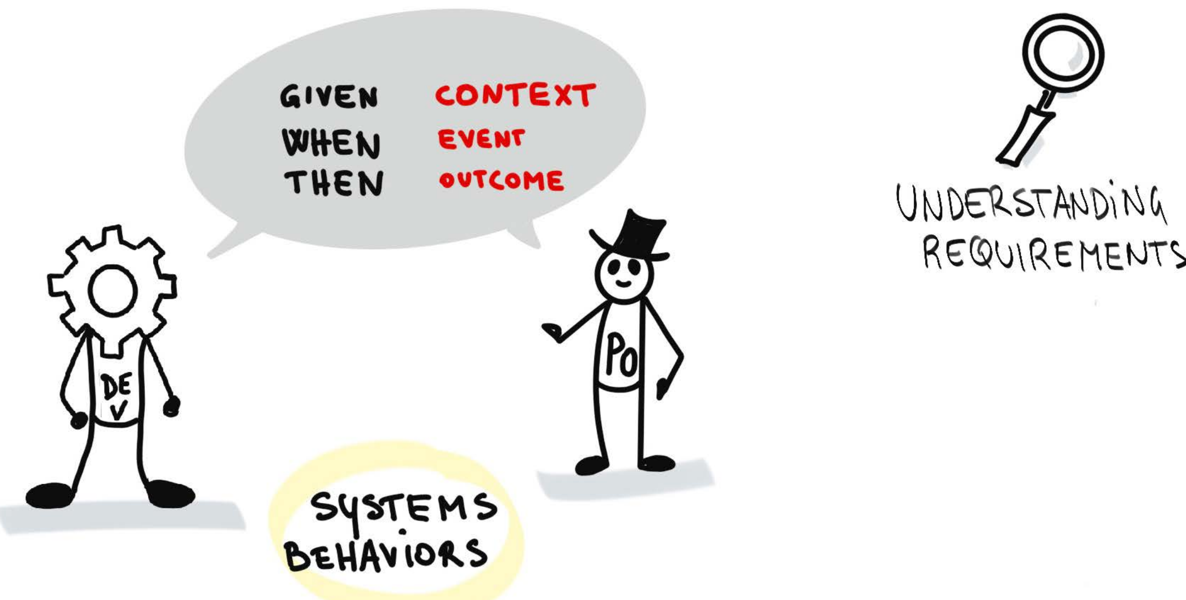


Figure 7.23: Developer-Driven Testing

BDD is a great tool to have in your toolbox as it brings people together in a shared understanding of the scope of a story or feature under development. It's less of an engineering tool and more of a method that focuses on the conversation to be had between business and developers when writing features. BDD is about using a shared language to write concrete examples of how an application should behave.

How the tests are implemented is then decided by the developers. But, more importantly, a common language can be used between developers and product owners to scope out a story without leading the design of the software. BDD can be a useful way to write acceptance criteria for a story together. There is a common syntax or approach to writing BDD tests based on work by *Dan North*, an agile coach and originator of BDD[4]:

```
Scenario  1: Title
Given [context]
And [some more context]...
When  [event]
Then  [outcome]
And [another outcome]...
```

4 https://dannorth.net/introducing-bdd/

For example:

```
Scenario  1: Buying an Ice Cream to cool off on a hot day
Given I have ten pounds in my pocket
When  I purchase a Choc Ice for two pounds
Then  I have only eight pounds left
And have a Choc Ice
```

For any feature being developed, there are probably a number of scenarios that could be tested. These scenarios are defined using the common syntax of Given, When, Then. Codifying the acceptance criteria using a common syntax can simplify the writing of tests and gaining a shared understanding of the scope of an activity. Dan North suggested this story-driven approach to BDD some years back and, since then, the syntax has been adopted by lots of the testing frameworks, such as Cucumber.

BDD for Our Ops Tooling Python Library

I worked on a public sector engagement a few years back. I was part of a team helping them automate some of their Ops capabilities. They had teams of people configuring VMs manually in a non-repeatable way. Part of that work involved me building a command-line interface for the team to help automate the creation and onboarding of team members (users) and their roles into a Free IPA server (Red Hat Identity Management). The following screenshot shows one of the BDD scenario templates for adding an existing user and deleting a user:

```
Feature: user

  @story-RF-1233
  Scenario: As a DevOps engineer I would like to be able to specify a job title when adding a user to LDAP so that the
  automember rules will work.

    Given The user has an optional parameter "title" set to "god"
    And All default add_user parameters
    When I create a user
    Then req param "title" is set to "god"
    And The user_add request is sent to freeipa with username "bob.builder"

  @story-RF-1233
  Scenario: Job title is optional

    Given All default add_user parameters
    When I create a user
    Then The user_add request is sent to freeipa with username "bob.builder"

  @story-RF-1233
  Scenario: User is returned without error where it already exists

    Given An existing user
    And All default add_user parameters
    When I create a user
    Then The user_add request is not sent
    And The existing user details are returned

  Scenario: User is deleted

    Given An existing user
    And All default add_user parameters
    When I delete a user
    Then The user_del request is sent to freeipa for group "bob.builder"

  Scenario: User is deleted becuase it doesn't exist

    Given A user "notexisting" does not exist
    And All default add_user parameters
    When I delete a user
    Then The result is None
```

Figure 7.24: A BDD scenario

The architect on the team was a strong believer in the BDD approach to writing stories. All of our acceptance criteria were written in this way, and it was a great way for us to understand the scope of what we were doing. When I was pairing with another engineer, we would use the acceptance criteria written in the BDD syntax as our starting point. We imported the syntax straight from Jira to scaffold out the test cases using Python Behave. For us as engineers, this made coding the features a breeze. We had been given the specifications, so we could easily implement our code to pass the tests.

BDD can help engineers understand the context of features better. It also helps bridge the gap of alignment with business experts and product owners:

Product Owners Seeing Their Thoughts in Code!

When we coach teams, we encourage them to use the sprint review or showcase events as an opportunity to show the world EVERYTHING they've worked on. That includes setting up and improving test automation.

One particular recurrence I've noticed from several teams I've worked with is when the product owner or business SMEs first see automation of BDD running. They think back to the sprint planning event a week or two earlier, when the teams were confirming acceptance criteria for stories they were going to accept into the sprint. Often, these criteria would be written using BDD syntax and it would be the PO or business experts providing the input.

When they see a test automation suite running in the sprint review or showcase, they will see the console showing the tests automated, the same thoughts, the same instructions, and the same business logic all codified.

BDD brings down the wall between technology and business people.

Example Mapping

Example Mapping, by Matt Wynne, CEO of Cucumber,[5] is another great tool to have in the toolbox. Once again, with a lot of these practices, it's just another really useful way to articulate and drive a conversation. In this case, Example Mapping is primarily used to drive shared understanding when writing stories and creating acceptance criteria. We believe it's great for helping teams write behavioral-driven tests. The process is simple and only involves four colored Post-Its:

- **Yellow**: For the story itself (as a header for the example map)
- **Blue**: For specific rules associated with the story
- **Green**: For examples of rules
- **Red**: For questions or unknowns that arise during the discussion

5 https://cucumber.io/blog/bdd/example-mapping-introduction/

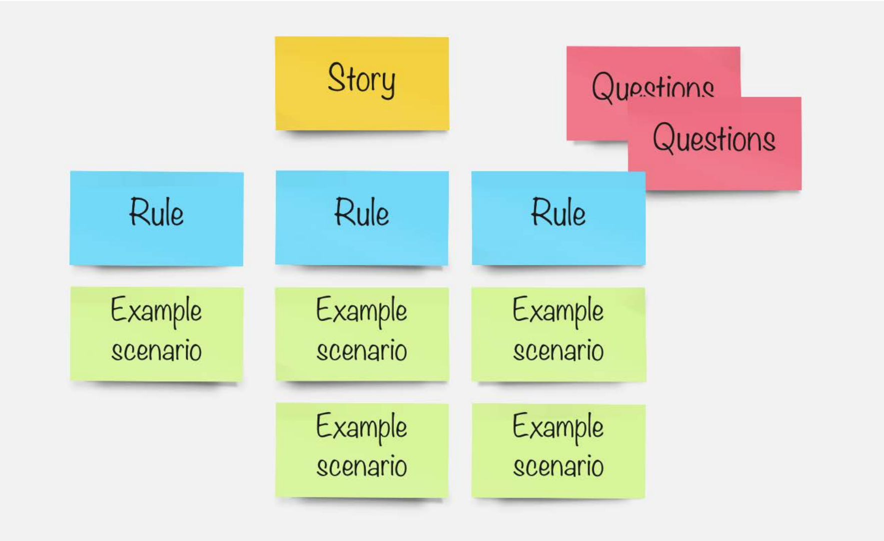


Figure 7.25: Example Mapping

Begin by selecting a story and write it on a yellow sticky note. Place it at the top of your example map as a header. In a horizontal row underneath that, begin writing business rules on blue sticky notes. Beneath the blue business rules, create columns of green sticky notes with individual examples of those business rules. These could be relatively unstructured Friends-notation *The one where...* examples, or full-blown Given, When, Then criteria.

As misunderstandings arise surrounding individual examples or entire business rules, add red stickies with questions written on them.

When there are enough examples that everyone is comfortable with, they can be rewritten as both automated tests and acceptance criteria.

Example Mapping in the Field

On the World Health Organization residency, I found this practice so simple to use but such a great tool to articulate the scope of a story and get alignment on the acceptance tests we'd write.

We were using Event Storming (more on this later) to model the onboarding process for a new user to their application. We had a command that read *Submit relevant topics of interest*, which was added to our backlog. We chose this command so we could learn more about things our users would be interested in, in order to better serve them recommendations.

We used Example Mapping to break this story down by first writing some rules. We were not super strict on following a ubiquitous language at this point as we knew the team would convert them into BDD-style syntax afterward.

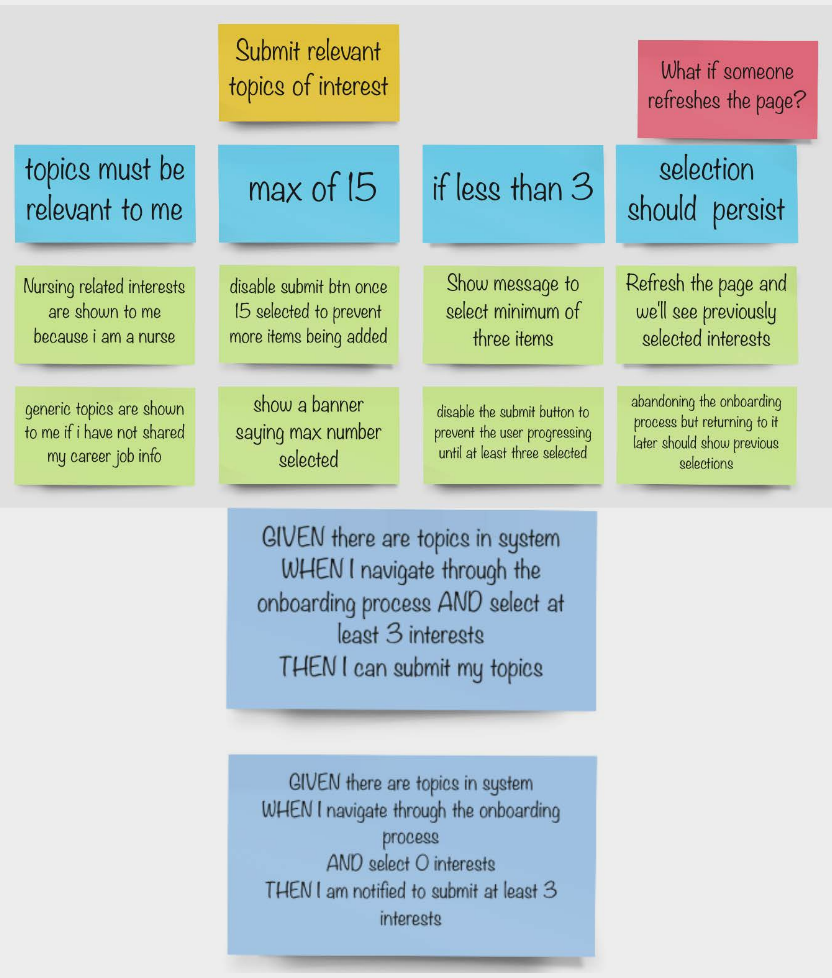


Figure 7.26: Example Mapping example

The conversion within the team brought out some misunderstandings regarding the scope of the activity. The developers wondered more about edge cases, such as, *what happens if the page is refreshed or returned to?* We were able to capture these questions as part of the Example Mapping and add a new rule and some examples. Subsequently, the team could convert the examples into the BDD syntax.

As with all these practices, the act of having this conversation with the correct people and capturing the examples meant we gained great team alignment and were able to convert them to acceptance tests and implement them as part of our development workflow.

You can learn more about, and collaborate on, the Example Mapping practice by going to the Open Practice Library page at openpracticelibrary.com/practice/example-mapping.

Non-functional Testing

While the importance of testing cannot be overstated, it's critical to keep an eye on other metrics that may give further insight into the quality of our code. For example, how do you know your tests have enough breadth to check all the code? What if my tests are passing, but the application response time is awful? Traditional unit and integration testing might not catch these things. There are tools we can use to identify causes and issues with our code base and, more importantly, fix them sooner rather than later.

Code coverage reporters are simple to implement and usually come bundled up with a lot of modern test frameworks. The idea is simple. While running our test cases, the code base is being watched. Once test execution is completed, a report is generated showing what lines of code have been hit and where there are gaps. These are useful reports to help the team identify where there is room for improvement but they should not be treated as the absolute truth. As with all these things, there are ways to trick the coverage reports, but good developers and peer review processes should catch these things. Often, teams will strive to increase the testing coverage if they have not started from a very good state. Bringing these reports to a retrospective can be good for teams to analyze and set higher targets. More aggressive teams may even fail their pipeline as unstable if the coverage is below a certain threshold!

Static code analysis is another tool that can provide insight into a code base not detected by unit testing, creating rules for how the code should look and execute. Consistency in an approach to how you write code is particularly important for non-compiled languages such as JavaScript. JavaScript also behaves differently in different browsers, so writing a set of rules such as using single quotes instead of double quotes for all strings can help ward off any unexpected behavior. If we have the rules codified, we may as well ensure that everyone adheres to them, so add them to our pipeline! Coding standards are very important in multi-team setups too. If the code base conforms to a standard structure and design, it can also make maintenance and updates to it very simple.

Performance Testing Sam's Code

Around 2014, we worked for a retail organization building mobile backend services and some automation around it. This layer of services was responsible for aggregating data from different backend systems such as product listers and categories, and reviews. The services also performed some very basic data manipulation to make the payloads more mobile consumable. It was critical that the adapters responded in a very timely manner, as mobile latency was high compared to modern mobile networks and a fast API response time made all the difference.

Our team was always cognizant that we should keep track of the time taken for the adapters to respond. We knew the organization would perform a traditional load-testing initiative at the end of the program; however, we didn't want to wait until then to reveal any surprises. We figured there had to be a way to continuously validate changes we made to the adapter tier in order to highlight any performance degradation.

We created a nightly job in Jenkins (our automation tool) to check the performance of the adapters each evening. This was a fairly simple job that simulated 1,000s of parallel requests to the APIs. From this, we plotted the response time of the service each day and reported it through Jenkins. This allowed us to create a baseline for where a normal response should be and allow us to fail the job's execution if the value fell above or below an expected range!

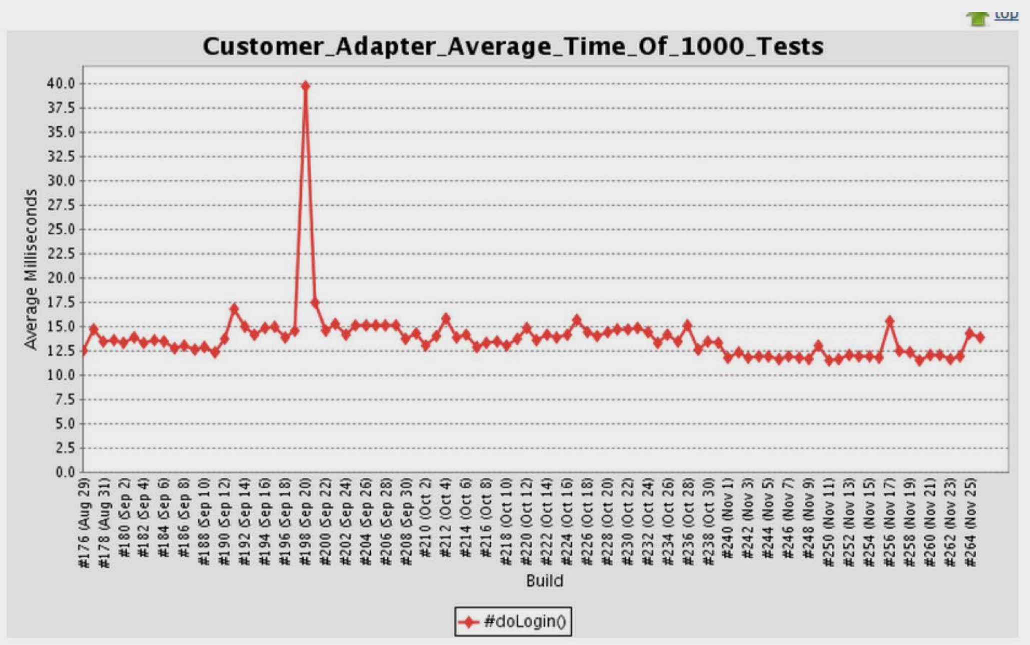


Figure 7.27: Automated discovery of performance bottlenecks

One day, we came into the office and our nightly job had turned red! Perfect, we thought, let's Stop the World and stop all the things we're doing while we inspect what's changed in the system since last night. A quick check of the changes that were made in the system revealed that Sam, one of the team members, had tried to check in some new logic for one of the data translation functions. Sam had introduced a big loop inside a loop inside another loop, which had caused the code execution time to spike. It was something that was not caught by our traditional unit testing, as the logic was working fine. It was just taking longer to compute.

We quickly responded and fixed the problem immediately. If we hadn't caught this performance bottleneck, it could have been weeks or more before we realized what was happening. We could have been building more functionality on top of this dodgy piece of code, making it much harder to unpick at a later date.

Being able to respond to feedback like this was critical. We're not saying that big load testing on the product was not necessary, but this one simple automated job provided us with a ton of value just by catching this one issue. It was cheap to write and maintain and caught this error potentially sooner than we would otherwise have noticed. Sam tried to write some more code a few weeks later and we got a similar failure.

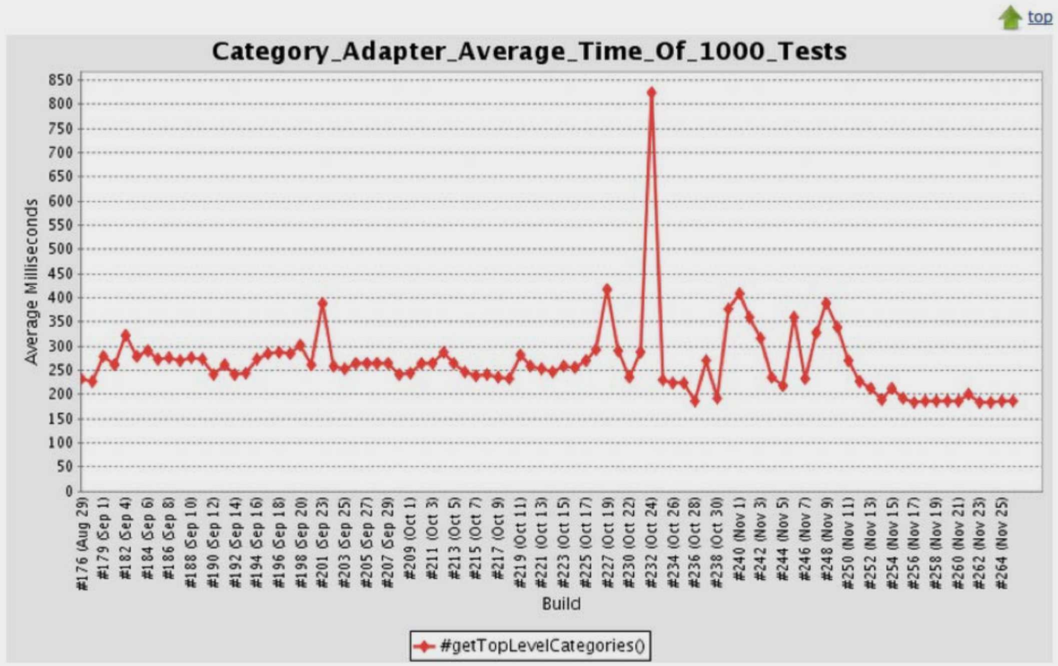


Figure 7.28: More light automated performance tests

Have no fear though – Sam, who's a good friend of ours, no longer writes code for a living as he's moved into a technical sales role. We don't have a blame culture within our workspaces and I'm sure if Sam was telling you this story, he'd say it was one of us that checked in that silly piece of code. I'll let you decide who it was.

There are lots of other types of testing and I won't list them all; we'd have to write another book to fit them all in. We go into more detail about the non-functional nature of our software in the next section, *Discover It*.

A Few Final Thoughts on Testing

We cannot understate the importance of testing in delivering features at speed, especially automated testing. Whether you follow the test pyramid or some other paradigm is up to you – just remember it's all about the conversation. If TDD is not the right thing for you, make sure you still have the conversation between business and technical teams to identify sensible tests using examples. The go-to for us is to use BDD as it allows us to bring together the world of business and technology.

Lastly, we're not saying there is no place in the world for separate QA teams. Not at all, it's about automating all the things and getting feedback early. If the QA is a separate function within your organization and is only engaged some weeks before going live, then this is a problem. Bring the skills of QA into the team and left-shift that capability into the team so that they can get early feedback more often.

Emerging Architecture

Hope is not a design method.[6]

How do we know our architecture is good? What does good mean? Is good architecture measurable? Have you ever had to operate, support, or fix a system that is poorly architected?

It may be easier to identify some characteristics of what *a poor architecture* looks like:

- An unstable and unreliable system that fails regularly in unknown and unexpected ways.
- The system is slow from a user's point of view.
- It does not scale well with increased users or loads.
- It is hard to upgrade because one small change requires everything to be re-deployed, which is slow and costly.
- It is dependent on clients or other systems and cannot be easily modified or changed without changing the other systems as well.
- It has a lot of complex business functions that are buried in the database, that may involve triggers, and cannot be easily changed due to a complex database schema with unknown side effects when modified.
- The system is hard to manage and operate.

6 Michael T. Nygard, Release It!: Design and Deploy Production-Ready Software

The list goes on and on.

In frontline software support and operations, there is nothing worse than getting called consistently at 3 in the morning to firefight a recurrent complex system crash, and after restoring the service, the root cause analysis points back to a complex failing architecture – where there is no easy fix other than re-designing or rewriting the software.

A lot of deep-seated issues with software arise from poorly judged architecture decisions. Many times, most of these decisions are made at the initial stages of product development when the big architecture is designed upfront and set in stone, concrete, or mud. Often, the system architects present their architectural masterpiece to the development teams and henceforth work to ensure the problem fits the architecture rather than the architecture fitting the problem.

The development process continues using this eighth architectural wonder of the world and all goes well initially; it even may run in production. But then, one day, the business asks for some feature that doesn't fit well into the architectural approach and then there's hell to pay to get the change done.

Technical decisions during development often have to be made based on the best intentions, but with incomplete or sparse information. The wisdom gained from the more experienced members of the team can often be invaluable during design discussions. Those with scars from previous projects with bad architecture are definitely worth listening to and learning from. There can, however, be a downside to this, as we'll see in The Hammer section.

As a product team, we must be willing and able to adapt and change the architecture when requirements substantially change or a system failure tells us that we're hitting the limits of our existing architecture.

> **Note**
>
> Generally speaking, it is better to make architectural and technical decisions as late as reasonably responsible to do so, so that the most information is available to those making the decisions.
>
> Emergent architecture is the practice of having *just enough of an architecture* so that the product developments keep moving forward, but is flexible enough that architecture changes can be made as more information becomes available.

There have been literally dozens of excellent books and articles written on what is considered *good architecture and patterns* over the years. Our personal choice is anything written by *Martin Fowler* (https://martinfowler.com/books/), *Chris Richardson* (https://microservices.io/), and *Sam Newman* (https://samnewman.io/books/), but there are many others.

Observations from the Field

In this section, our intention is to outline some of the recurring patterns/approaches, both good and bad, that we've come across. None of these are new, but we thought it useful to call them out here.

Patterns per Square Meter

The *count of software patterns applied* in a product is never a good quality metric. Software patterns are well-known, reusable templates for solving particular problems. Don't make the mistake of assuming that a system that includes a bunch of software patterns is superior to one with fewer of them.

Expect Failures and Deal with It

Catastrophic system failure is clearly undesirable. A system that never completely stops working in the face of subsystem failures is usually preferable. It can recover gracefully when subsystems fail and may support a subset of functions even if components are unavailable. We can apply architectural patterns wisely, for example, bulkheads to reduce the damage done by any individual failure. Absolute failures are often easier to deal with than capacity, latency, or seems kind of slow issues.

When reviewing an architecture and, in particular, a distributed one, one piece of invaluable advice that I received from a very experienced architect some time ago is this – always be asking this question: *What happens if this component fails or slows down*? If there is no good answer to the question, then there is likely more that needs designing to prevent failure scenarios.

The Hammer

One pattern or behavior that we've seen a lot through the years is the Golden Hammer pattern, in other words, if all you have is a hammer, everything looks like a nail. This is more formally known as the Law of the Instrument.

All developers have their favorite tools and architectural approaches. For example, the authors are fans of reactive, streaming architectures (Mike), asynchronous event-driven messaging (Noel), and anything with Node.js or Emojis (Donal). The risk here is that your own bias may lead you down an architectural path that is ultimately the wrong approach.

If you find yourself listening to the first 10-20 seconds of a discussion around a business problem and feel compelled to jump in saying Oh well, product, architecture, or tool X can help with this, let's face it, you may be reaching for your golden hammer.

Resumé-Driven Development

Many times, we've been involved in discussions around a technology being used in a customer solution that is either out of place or just doesn't quite fit the problem they're trying to address. We often discover that someone had introduced this technology as they were keen to learn it and somehow it went from a technical experiment or spike to a core component technology. There's absolutely nothing wrong with technology experimentation; it should be encouraged, but care should be applied to ensure that a chosen technology doesn't lead to a dead end or become a technology solution looking for a problem. Examples of technologies where we've seen this include Service Mesh and others as depicted in *Figure 7.29*:

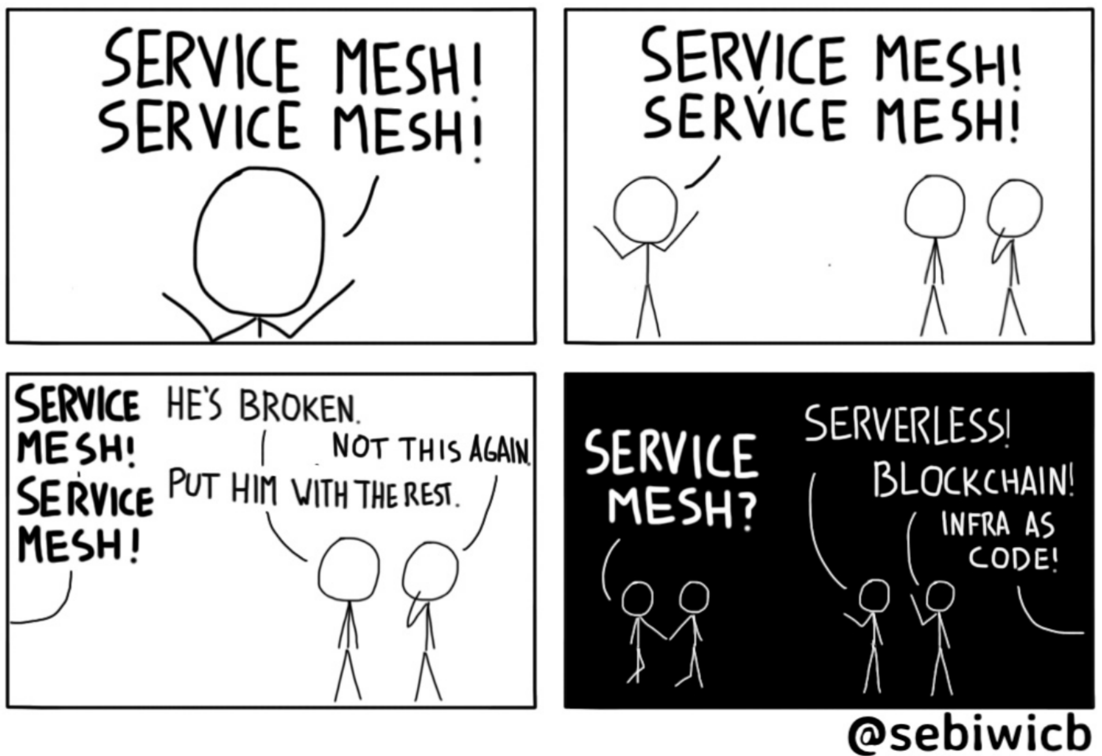


Figure 7.29: Adopting the coolest tech on the block

Wear Different Hats

Software architecture has to work well from multiple perspectives, not just from the design and build viewpoint. Different teams/personas will have different perspectives, for example, deployment, testing, and operational management. A *good* software architecture will try to address as many of these concerns as possible.

Social Media-Driven Development — Keeping Up with the Cool Kids

Companies such as Google, Amazon, and Microsoft produce and utilize some amazing technology and techniques. These are often focused on the size of the problems that these hyper-scalers face. Most of us never work with that scale of complexity or user demand, so care should be taken that you judge technology on your particular business needs rather than what is the tech *du jour* that the *cool* kids are using. One area where we observe this a lot is in the *monolith versus microservices* discussion. Both are very relevant and very valid approaches to software architecture. Both have their pros and cons, but the correct approach to take is to ask yourself what is best for the business and customers that adds value.

Good Service Design

Good service design can be hard to achieve. Ultimately, we should always lower the operational burden of our applications. We can do this by designing them so that we minimize the cost of change of any given application. Modern applications are normally broken down into different components or services that expose methods or functions. At the heart of good system architecture is service design. This is often based on practices such as **Domain-Driven Design** (**DDD**), which is fundamentally about understanding a business problem and communicating that understanding among the team in an unambiguous way. Services that are part of the same business domain are grouped together, like our Tournament Service in PetBattle V2, or our Cat Service in our hobbyist application. We can achieve good service design by following these two principles:

- **Loose coupling**: When a change to one service does not require a change to another service. By designing loosely coupled service APIs, we can deploy service changes easily. The interior design of the service may be changed completely without API consumers being affected.
- **High cohesion**: We want related system behavior to sit together and unrelated behavior to sit elsewhere. Our order management system is independent of our shipping and delivery system. This lowers the cognitive load for developers because related system functionality sits together. Often there is design tension here between defining related business system domains (using DDD, for example) and reusable technical functionality, such as libraries or APIs, that may span multiple systems.

Technical Design Group-Think

We have talked about some anti-patterns when designing systems in general. Another bit of advice is for technical leaders to set directions and not just descriptions of what to do or of what has been. One way to help achieve this is an exercise in parallel thinking, whereby everyone contributes their ideas collaboratively and at the same time, rather than just following the one way of thinking from the most senior in the team. The emphasis is on *what can be*, not *what is*, to help design a way forward. It is not about who is right and who is wrong.

Human Resources and Time Are Your Most Valuable Assets

In knowledge-based work, humans are usually the most expensive resource. So, it makes sense to strive to reduce toil or undifferentiated manual work. This is a never-ending trend to automate all the things, which allows much better quality and feedback for our products.

Information Leakage – Data Centricity Matters

In tech, we are flooded with data. But do we make the best use of all the data available in our applications and systems? When architecting, we have to consider carefully the quantity and quality of data available within our applications and infrastructure. Often, engineering decisions and trade-offs must be made that move data processing nearer edge devices just because sending all that data back to a central processing core is not physically possible because of bandwidth or latency restrictions, or it is just too costly to move all that data around (think cloud!). So, when designing systems, consider when:

- Data integrity is lost during data capture
- Data is not streamed or stored at all
- Data is not accessible for other uses
- Data is not analyzed at all
- Data is not communicated and remains hidden
- Data is not used in decision-making

We often forget to think about how much data is lost – the lost data can be a massive source of lost opportunity for our business. This happens in cloud, IoT, industrial, and even mobile web use cases with processing data on our mobile phones.

Some Final Musings on Architecture

Architecture is a critical concern when building software systems. Getting it right is a continuous balancing act of reviewing current and potential future requirements and assessing how the architecture fits those requirements.

A certain degree of upfront architecture and design work is always needed, and it should be accompanied by flexibility and honesty to ensure that the initial architecture can change as answers to uncertain questions are discovered and more information is added to the collective understanding of the problems at hand. The ability to constantly improve the architecture throughout the product life cycle is another important goal.

One quote that comes to mind when we talk about big upfront decisions is the following famous quote from a Prussian field marshal: *No plan survives contact with the enemy. – Helmuth von Moltke*. Or, in more modern terms: *Everyone has a plan till they get punched in the mouth. – Mike Tyson.*

Flexibility, adaptability, and the willingness to change are key characteristics required for success in dynamic environments. In summary, there are many architectural considerations that the team needs to consider as their applications scale and adapt to change. By experimenting and adapting the architecture as the business needs change, they will be better able to deliver the service SLAs that were promised and ultimately evolve the user experience to be optimal.

Conclusion

In this chapter, we continued our exploration of technical practices to create a solid foundation for us to be able to deliver at speed as one single cohesive unit. By using techniques such as the Big Picture to gain a shared understanding of our delivery pipelines, we further identified methods for testing and how we can connect the business to the acceptance tests in a way that's more developer- and business-friendly.

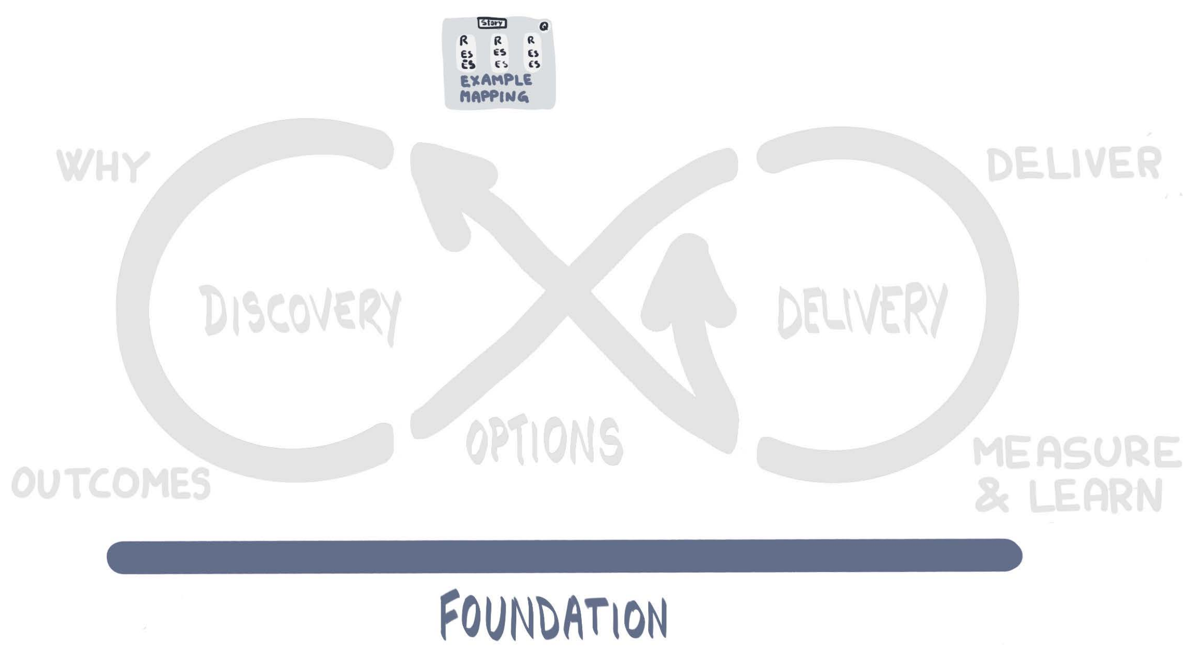


Figure 7.30: Adding more technical practices to the foundation

As we explored lessons that led us to an emerging architecture approach, we also learned that a lot of the magic is in having the conversation to begin with. Treating the whole of your IT organization as a satellite will not be effective; we must create an environment where we can succeed as a whole. Key to this is bringing together the people with the knowledge, and the people with authority and power.

In later chapters, we will go more in-depth into the technical implementation of PetBattle.

To close off this section, we have now built a solid foundation of culture, leadership, and technical excellence. We have put in place principles and practices, including:

- Autonomy, mastery, and purpose
- Psychological safety
- Social contracts, stop-the-world events, real-time retrospectives, team identity, and information radiation
- Leadership intent and team empowerment
- Priority sliders
- Team spaces
- Everything as code
- Containers
- Continuous integration, continuous delivery, and continuous deployment
- Test automation
- Emerging architecture

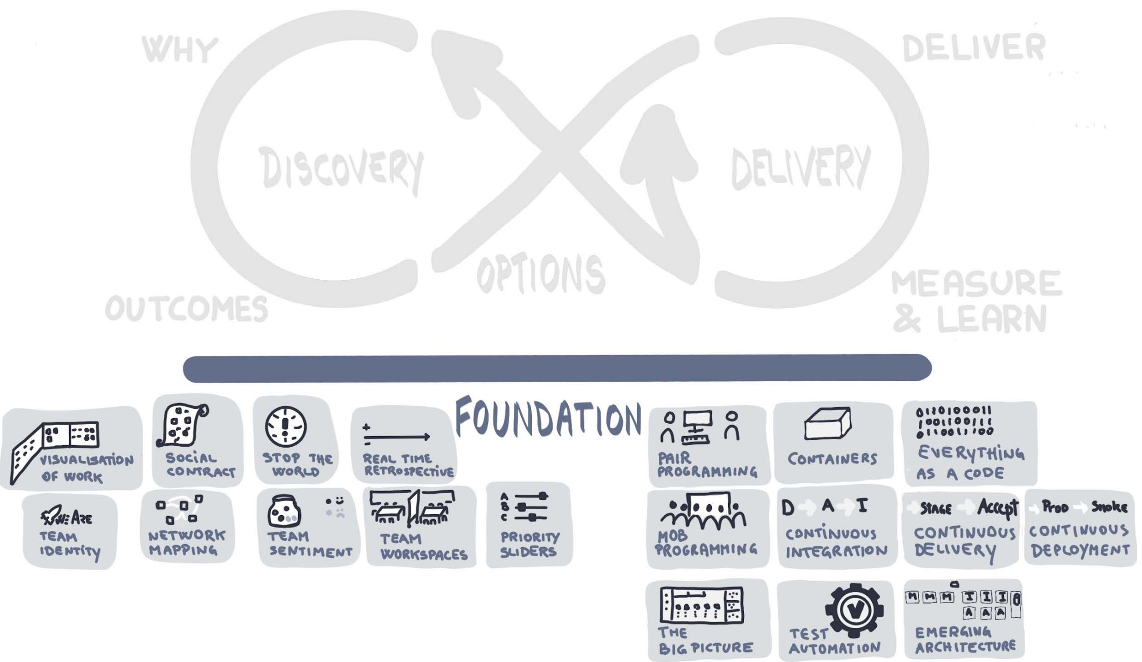


Figure 7.31: The foundation of culture and technical practices

Our foundation is strong. It will need continuous nurturing and bolstering as we build products on top of it. However, we're good to go with our first product teams. In the next chapter, we'll explore some practices we can use for continuous product discovery.

Section 3: Discover It

In *Section 2, Establishing the Foundation*, we built up the initial foundation for our first teams to work on. It consisted of a culture that enabled psychological safety and promoted autonomy, mastery, and purpose for team members. It also consisted of the technical foundation that we bootstrap to enable teams to confidently start their application development, giving them the best chance of maintaining steady continuous delivery and not getting trapped by early technical debt.

We're now going to work our way around the Mobius Loop, starting with Discovery:

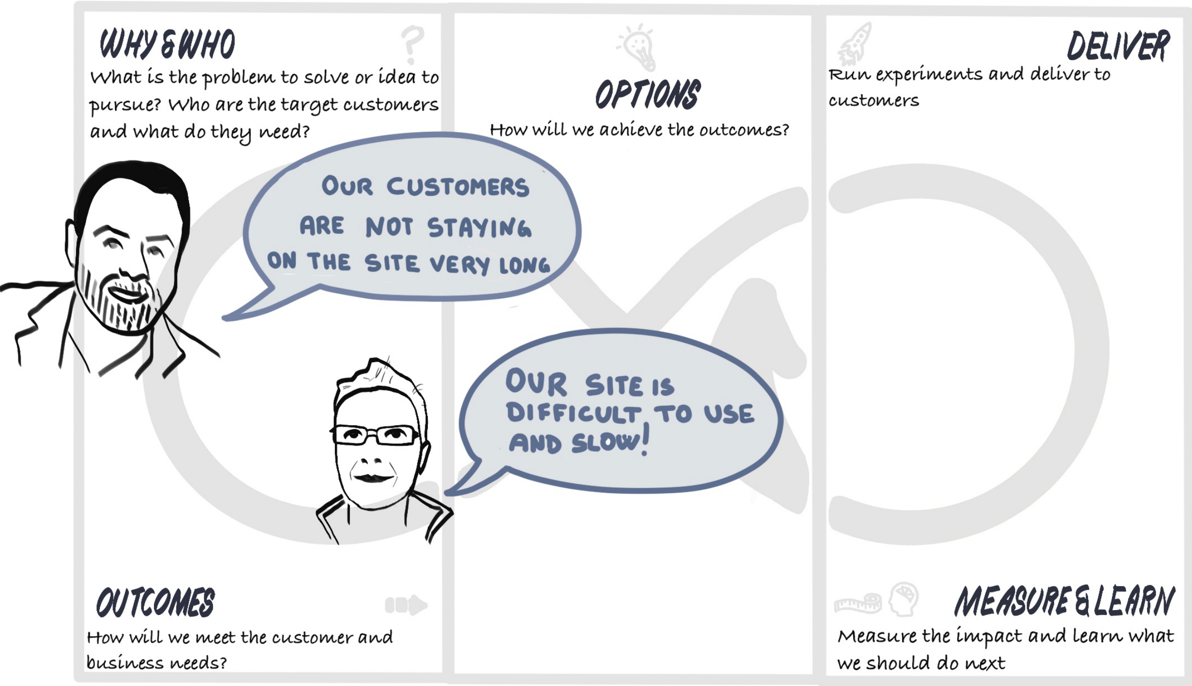


Figure 8.0.1: The Discovery Loop — setting the scene

In *Chapter 8*, *Discovering the Why and Who*, we're going to explore how teams and stakeholders can collaboratively discover outcomes and understand the common purpose underpinning application development. To help navigate through this part of the process, we will be using the Discovery Loop of Mobius:

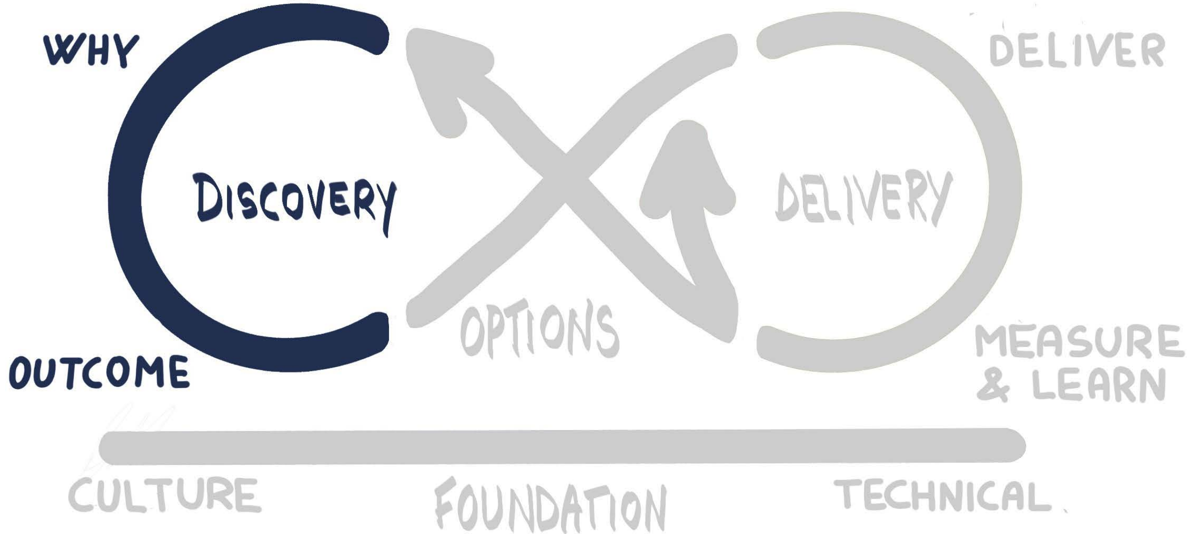


Figure 8.0.2: The Discovery Loop

When we are on the **Discovery Loop**, we identify and use practices that help us answer the question of Why – why are we embarking on this journey? What problems are we trying to solve? Who are we trying to solve them for and what do we know about them? What is our great idea? We also use practices on the Discovery Loop to help us identify and set measurable target outcomes for the business and its customers. *Chapter 8*, *Discovering the Why and Who*, focuses on the upper portion of the loop, helping to answer the **Why and Who** questions.

In *Chapter 9*, *Discovering the How*, we will shift the focus to practices that can help us start building a solution. Our approach to solving problems is incremental and experiment-driven. In particular, we focus on a group of practices that bring about a shared understanding through visualization and modeling techniques. No sticky note will be spared in this chapter!

Chapter 10, *Setting Outcomes*, will focus heavily on those outcomes, what they are, how we derive them, and how we use them. By the end of this chapter, you will be able to clearly articulate and define the team goals and outcomes being delivered.

8

Discovering the Why and Who

Too often, software delivery teams dive straight into delivering features and focus purely on output and deliverables. They don't take sufficient time to understand what business problems they're trying to solve and the people they're trying to solve them for.

There is often a split knowledge distribution of business processes and domain knowledge that sits inside one person's head. The misalignment in knowledge across the team and between different stakeholder groups causes misguided decisions, misunderstandings, and incorrect assumptions to be made, with teams delivering the wrong solutions solving the wrong problems.

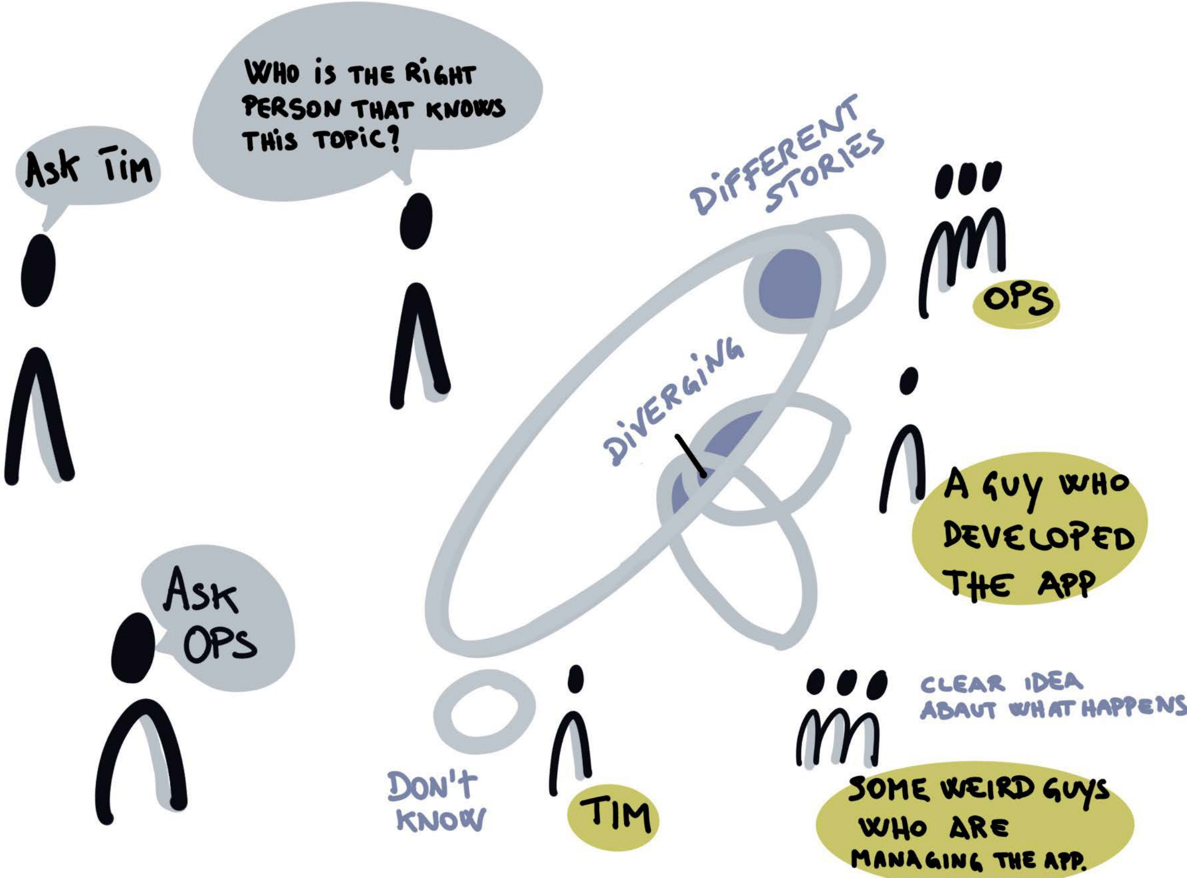


Figure 8.1: The distribution of knowledge

This is why we start with using a selection of discovery practices in order to align all team members and stakeholders with a shared understanding of *why are we doing this? why are we investing time? why are we embarking on this and not something else?*

Furthermore, we explore what problems we are trying to solve and who we are trying to solve them for. Understanding these people, their pain points, and opportunities for improvement is crucial before we can start defining target outcomes for them.

The Discovery practices we are going to explore in this chapter really benefit from a whole-team approach and being inclusive of all interested stakeholders. Gathering a wide range of views and encouraging engagement and collaboration across a broad group of people from different functional areas across the business helps to achieve alignment and a committed group of people who share the same understanding of the problems we're trying to solve.

We intentionally wish to include both technical and non-technical people. Practices that facilitate conversation and alignment between more technical-focused people and less technical- or more business-focused people are particularly powerful.

The use of discovery practices is not intended to be exhaustive – we use these practices to get just enough information to align people, gain a shared understanding, and acquire enough confidence to go into the delivery of early experiments or other activities, which, in turn, drives more learning. This will not be the only opportunity to undertake discovery and use these practices – in fact, our goal is to regularly return and build on discovery using learning from delivery.

In this chapter, we will cover the following topics:

- **The North Star**: A practice that helps achieve alignment on the Why and overall purpose.
- **Impact mapping**: A practice that helps achieve alignment on the Who and the measurable impacts we would like to have on them.
- **Human-centered design**: A group of practices that helps develop empathy with the user and other important stakeholders.

Let's take a look at our first discovery practice that helps align everyone to move in a single direction, the North Star.

The North Star

Creating a shared purpose for a product can be achieved by setting a North Star. The North Star Framework is a model for managing products by identifying a single, crucial metric known as the North Star Metric.[1] According to Sean Ellis, who coined this term, this is *the single metric that best captures the core value that your product delivers to [its] customers*.

Amplitude[2] is a digital analytics company that has freely released an excellent playbook regarding the North Star Framework – what it is, why you might use it, and how to use it. It lists the following seven checklist points about what a North Star Metric is:

1. It expresses value. We can see why it matters to customers.
2. It represents vision and strategy. Our company's product and business strategy are reflected in it.

1 https://growthhackers.com/articles/north-star-metric

2 https://amplitude.com/

3. It's a leading indicator of success. It predicts future results, rather than reflecting past results.
4. It's actionable. We can take action to influence it.
5. It's understandable. It's framed in plain language that non-technical partners can understand.
6. It's measurable. We can instrument our products to track it.
7. It's not a vanity metric. When it changes, we can be confident that the change is meaningful and valuable, rather than being something that doesn't actually predict long-term success, even if it makes the team feel good about itself.

The North Star Framework complements Mobius very well. In the discovery loop (this section), we focus on defining the North Star and getting an aligned and shared understanding across stakeholder groups and team members as to why the North Star Metric has been chosen. In the next section, we'll be exploring the options pivot, which is all about the actionable research, experiments, and implementations we can perform to influence a North Star Metric. In *Section 5*, *Deliver it*, we'll explore the delivery loop, including the practices to measure and learn – our instruments for tracking a product's success against a North Star:

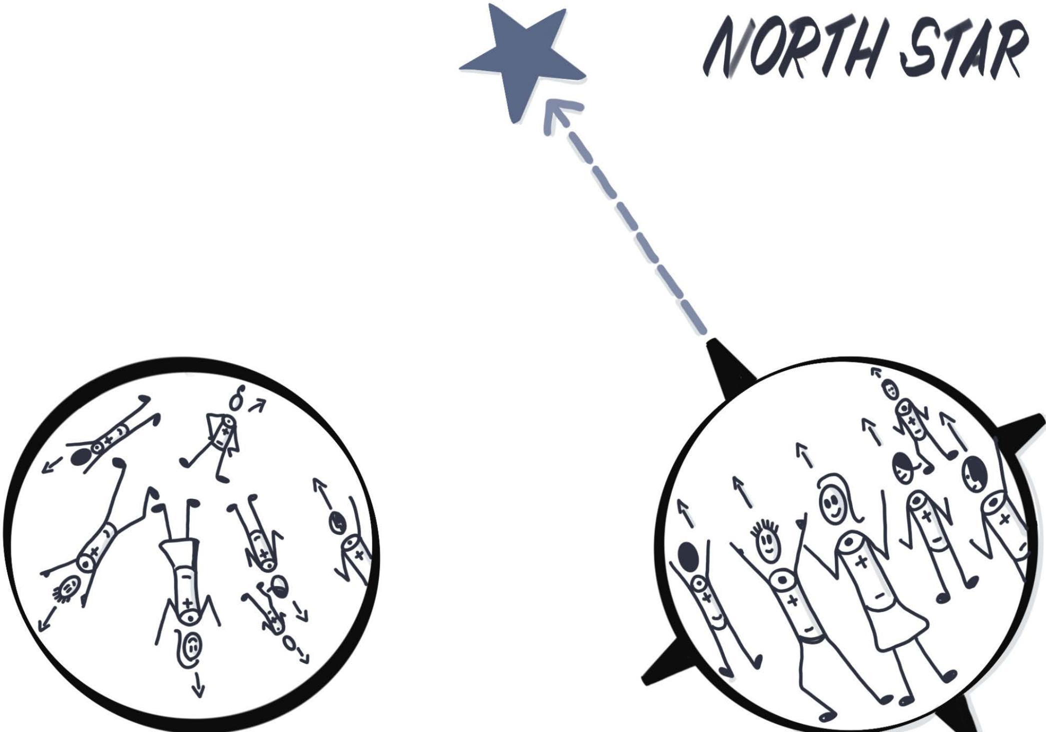


Figure 8.2: The North Star

Using the North Star practice during a discovery loop can help set teams off in the right direction and achieve alignment between all its members and its stakeholders. It can also help teams achieve course correction or re-alignment if they've been running without a North Star. Having a North Star's information radiated on the wall and clearly in view of team members and interested stakeholders can help maintain focus on it and steer the team toward it.

Many of the practices we explained in the previous section to create a foundation of culture and collaboration will benefit the process of defining and using the North Star. For example, having a social contract and retrospective practices that help generate a team's autonomy and psychological safety in terms of contributing to strategically important information will help to promote wider group ownership of this metric. Having practices to promote visualization of work will help share, inspect, and focus on reaching the North Star.

So, the North Star is the first discovery practice that highlights the importance of needing a foundation of culture in place. Its effectiveness is much higher if an open culture exists, with people feeling psychologically safe to contribute and collaborate with each other. We will see this pattern continue.

To help find a North Star, we make use of a simple canvas, such as the one provided in Amplitude's playbook:

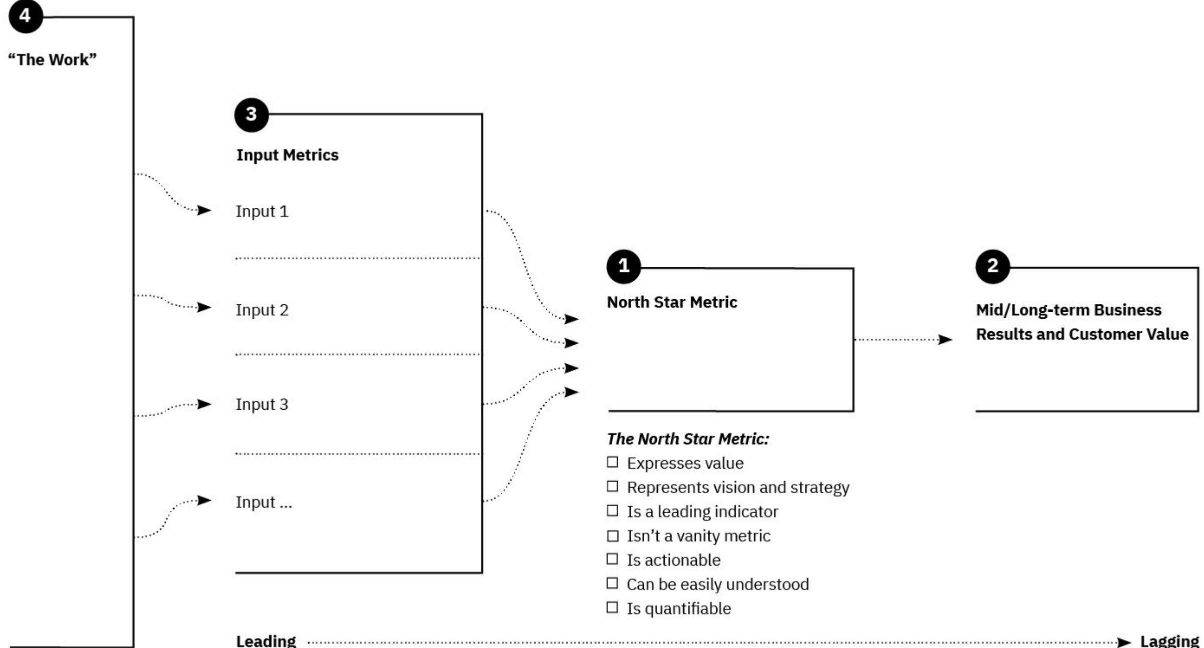


Figure 8.3: North Star playbook from Amplitude

A team using the North Star Framework identifies a single, meaningful metric and a handful of contributing inputs. Product teams work to influence those inputs, which, in turn, drive the metric. The North Star is a leading indicator of sustainable growth and acts as connective tissue between the product and the broader business.

Let's look at an example of a North Star Metric and how it was derived from inputs. OpenTable is a popular platform for booking tables at restaurants. A recommended approach to discovering the different leading inputs is to consider breadth, depth, and frequency; for example:

- **Breadth**: How many users are being engaged? An example leading input indicator for this would be how many **Monthly Active Users** (**MAUs**) there are – users who at least performed a search on the site.
- **Depth**: What level of engagements are users having? An example input indicator for this would be the booking conversion rate per session with a search.
- **Frequency**: How often do users engage? An example input indicator for this would be the average time interval between user searches.

These are all inputs we want to drive the improvement of in our product design decisions, but we require one metric, which is a crucial metric that represents the value of the product. OpenTable's North Star Metric is the number of seated covers per month – in other words, the number of times restaurant tables are used by paying customers each month. If this is high, we have a successful product driving good business. It is the one metric that does not lie and is the one we should be relentlessly focused on as being the indicator of success:

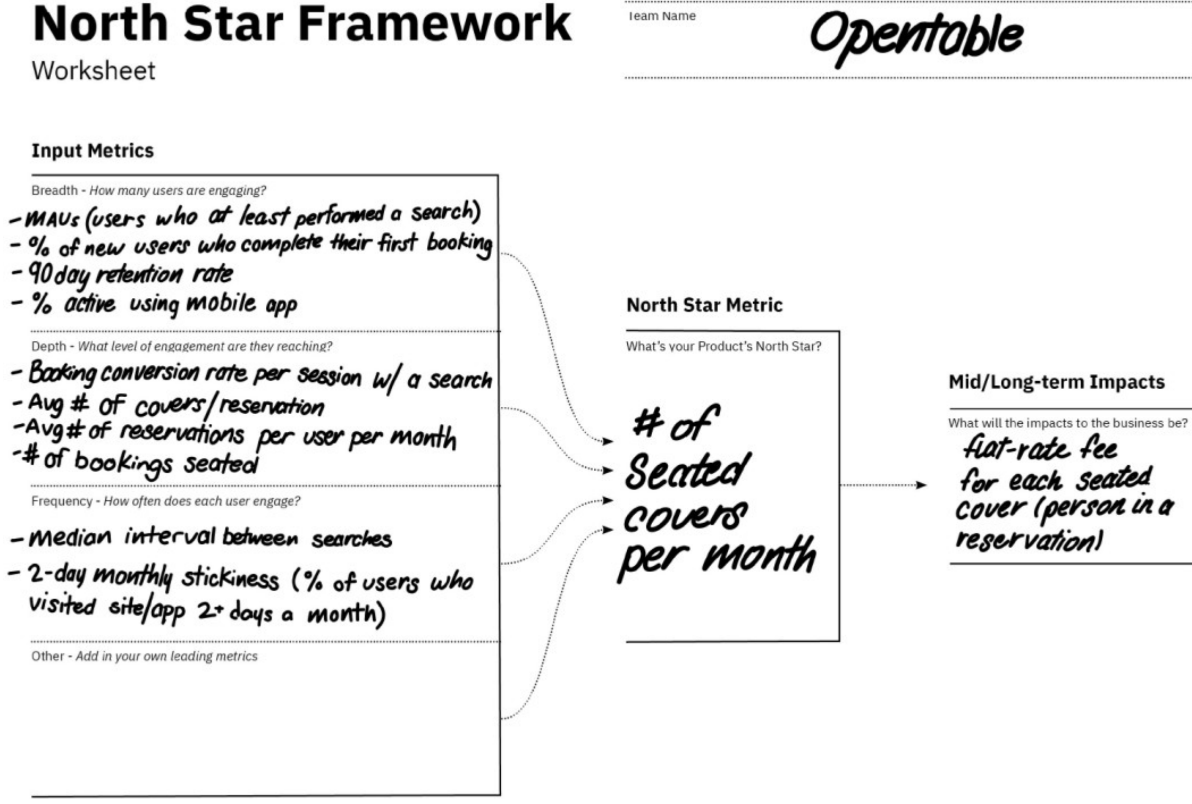


Figure 8.4: North Star canvas for OpenTable

This canvas also captures what the mid/long-term impacts to the business of the North Star Metric will be; in this case, a flat-rate fee taken for each seated cover booked generates revenue for OpenTable.

Let's go back to our PetBattle team and see how they went about finding their North Star Metric.

PetBattle as a Business

PetBattle was initially conceived as a fun idea to experiment with new technologies. However, due to cats being the internet's favorite time-waster, the site quickly went viral. As it was experimental, it wasn't built in the most robust or professional manner. This caused the site to crash due to the overwhelming number of visitors – a cataclysm of purrrfect disasters! Malicious users on the internet have been uploading inappropriate content to the site, which has to be removed manually as the site is intended to be family-friendly.

So, the founders decided to try and monetize the site while still keeping to the simplistic approach of the initial creation.

PetBattle's founders decided to embody the spirit of Dan Pink's book, Drive, when building up their cross-functional product team, so started off by building a cultural foundation. Each new team member adds their unique view, personality, and motives to the culture of PetBattle. In order to retain talent and enhance this great culture, PetBattle must make them feel like they can achieve autonomy, mastery, and purpose.

PetBattle's employees need to feel connected to the vision of the product. They held a North Star workshop to connect all employees to the vision of their product and determine their North Star so that they can regularly inspect and measure against it.

The workshop lasted for two hours and included the following:

- An opening discussion on the reasons for North Star
- Identifying the game PetBattle users would be playing
- A discussion on what makes a good (and bad) North Star
- A discussion on the structure of the North Star and its inputs
- Looking at North Star Metrics for other products
- Drafting PetBattle's North Star

Amplitude has categorized digital products into one of three possible games. The Attention game focuses on how much time customers are willing to spend on a product. The Transaction game focuses on how many transactions customers make in the product. The Productivity game focuses

on how efficiently and effectively a team member can get their work done. Following some discussion on whether PetBattle was more of an Attention game or a Transaction game, the group concluded on it being an Attention game. Ultimately, customers want to spend lots of time enjoying the PetBattle product and the aim is to keep them engaged.

In collecting candidates for the PetBattle North Star, employees used a silent brainstorming technique first, and then paired up with one other person to discuss their ideas. Two pairs then formed a group of four people to merge their collective ideas. Finally, the full group converged all their ideas on a single wall. This facilitation approach is called 1-2-4-all and is a great example of a liberating structure .

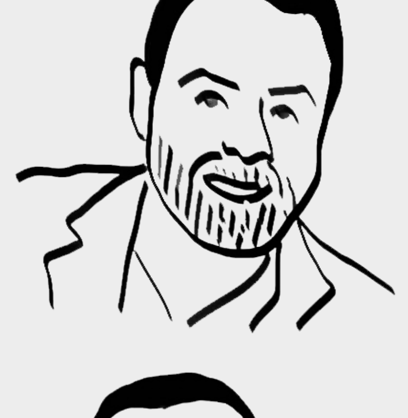
Candidate metrics for the North Star included the following:

- The number of site visitors
- The number of site visitors who have engaged (for example, at least placed one vote for a pet)
- The length of time spent on the site
- Increased follow-through to (eventual) advertisers
- The number of PetBattle competitions
- The number of pets uploaded to the site
- The site's drop-off rate
- The site's stickiness rate (the percentage of users returning three or more times in a month)
- The percentage using the mobile app

During convergence of all employees' ideas, some great conversation started to materialize about what they really saw as success for PetBattle. Yes, they wanted more people to visit and yes, they eventually wanted advertising revenue, but the consensus seemed to be that the one crucial metric that was going to take PetBattle to the next level was the number of pet uploads. Most of the other metrics were inputs that could all help drive up the number of uploads. If there are more users, eventually more of them will engage. As more engage, eventually more will want to engage more and participate. The mid/long-term impact of participation would drive the impact of increasing the likelihood of advertising.

We can use a North Star Metric in both product and services organizations, as demonstrated by our own use in Red Hat's Open Innovation Labs.

Our North Star at Open Innovation Labs



Open Innovation Labs is a professional services organization. We are often asked whether we can take a product mindset to a services organization.

The answer is yes! We use all of the practices we talk about in this book on ourselves to grow a product mentality for our business.

This includes the North Star. I recall our team meeting when forming the concept of labs to consider what our North Star Metric might be. Our customers come from both the commercial and public sectors. Our overall vision is to unlock our customers' greatest possible potential by leveraging open technology, open processes, and open culture. Our business value stream involves collaboration with lots of customers in a pre-sales capacity, understanding pain points and opportunities for optimization, and identifying suitable use cases that would be good candidates for our residency program. We hold sessions to qualify, discover, plan, engage, and deliver these services. All of these have their own metrics.

Using a similar collaborative approach to that of the PetBattle story above, we arrived at a *number of qualified use cases for labs* as our North Star. It became clear (just like a North Star) that this is the crucial metric that we can use to assess the health of our business. When we have qualified a use case, we are very confident as regards our customer that our lab's residency program is the correct fit for their business problem. We have done sufficient discovery so that we collectively recognize the value it will deliver. We have high confidence in the engagement being sold and being delivered successfully.

Knowing, understanding, and aligning on your North Star provides an excellent basis to move to the next discovery practice, which explores the linkage of strategic goals to people, measurable impacts, and deliverables. This practice is known as impact mapping.

Impact Mapping

Impact mapping is a technique that helps build shared understanding and alignment between leaders of organization strategy and product delivery teams. *Gojko Adzic* first documented the technique in his 2011 brochure,[3] which provides an excellent guide for people who want to facilitate the workshop.

Adzic defines impact mapping as *a lightweight, collaborative planning technique for teams that want to make a big impact with software products. It is based on user interaction design, outcome driven planning and mind mapping. Impact maps help delivery teams and stakeholders visualize roadmaps, explain how deliverables connect to user needs, and communicate how user outcomes relate to higher level organizational goals.*

At the end of the impact mapping exercise, you should have:

- A shared understanding of the goal or problem statement.
- An inventory of human behavior changes in users and stakeholders that must occur (or not occur) in order for your project to be successful. These are the *impacts* from which the technique gets its name.
- Optionally, an inventory of project deliverables that could be delivered to achieve the aforementioned impacts.
- Some sort of prioritization of scope – one or more deliverables – and/or impacts.
- A graphic presenting the preceding information.

3 https://www.impactmapping.org/

Using this approach allows us to move to an experimental and hypothesis-driven model to feature development – one that quickly allows us to prove or disprove hypotheses. That is, we hypothesize that building a deliverable will achieve a measurable impact on a defined actor to help deliver a strategic goal. The learning associated with running these experiments to prove or disprove a hypothesis is what drives our continuous improvement to deliver outcomes that matter.

Impact mapping enables us to focus on business objectives and orient product teams toward delivering business value and not just delivering more and more features. It is a popular practice and one we use early on in most of our engagements because it provides a very simple human-centered discovery process. You don't need fancy tools or software to run it – some sticky notes, pens, an open space, and, most importantly, people are all you need. The resulting artifact is a graphical mind map of all the conversations.

How do you do this? Well, start by reading Adzic's brochure. It is very easy to consume and will set you on your way. What follows is our experience of using impact mapping on a number of engagements. We'll break this down into the four components that make up an Impact Map:

1. The goal
2. The actors
3. The impacts
4. The deliverables

For each of these components, we'll do a short overview and then look at our PetBattle case study to see how their Impact Map evolved.

Building impact maps can be done with a group of people co-located using a big wall space. It can also be done virtually using a collaborative tool such as Miro, Mural, Google Slides, or PowerPoint. We have provided a PDF of a template which may be useful to get you started.

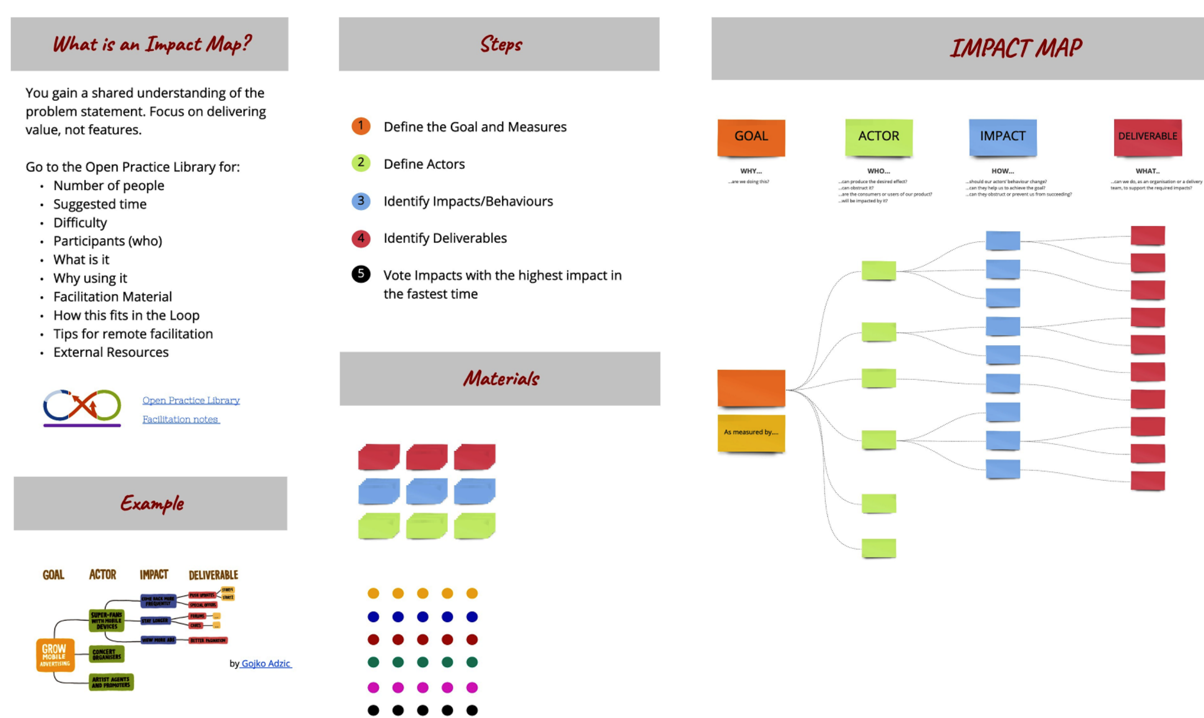


Figure 8.5 Impact Map digital template

We'll then bring the Impact Map together and show how we use it to form hypothesis statements that can be used to drive experiments.

Start with the WHY — the Goal

At the very beginning of this book, we said we always want to start with the Why. Why are we doing this? Why are we here? Collaborating over, aligning, and agreeing on the goal is doing this – starting with the Why.

This first section of an Impact Map is often the hardest. The group of stakeholders and team members should frame the goal in terms of the company's measurable success and not a technical solution. As technologists, we often focus on the What without asking Why.

By focusing our minds on why feature A is important, we can connect engineering practices and priorities to a company's success. Goals should be measurable so that we can quantify whether we are successful. The key output of setting a goal within a team is having a shared understanding and alignment on why we are setting out on an initiative.

Goals could be structured using the **SMART** acronym:

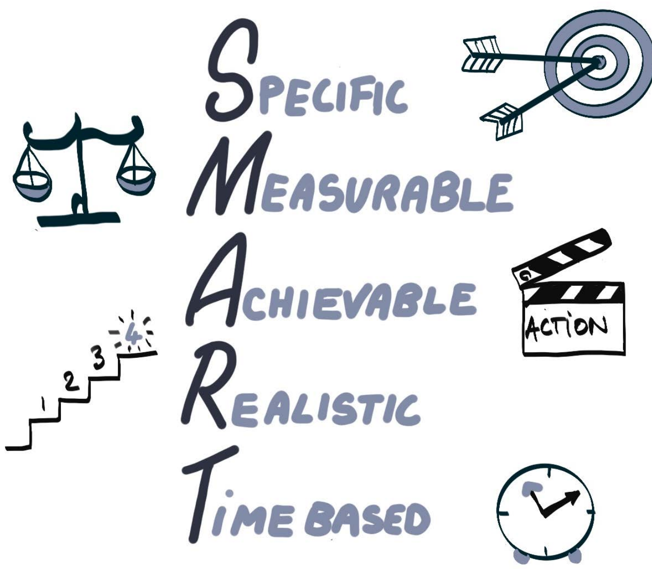


Figure 8.6: SMART goals

- **Specific**: A goal that is vague or can be misinterpreted has a lower chance of being accomplished.
- **Measurable**: How do we know that we have accomplished the goal? It should be measurable.
- **Achievable**: Can we realistically complete our goal; do we have the skills and things needed?
- **Realistic**: Can we complete our goal given time and money restrictions?
- **Time-based**: How long does it take to complete our goal?

Goal setting can be a difficult thing to agree upon. Often, there are competing views on what direction we are headed as an organization with the product or business area we are impact mapping. However, clarity of the goal for an organization can trickle down to all levels of a company, thereby creating a shared purpose for the individuals.

Let's look at the goal set by the PetBattle team.

PetBattle – the Goal

In our example PetBattle application, we are pivoting from a hobbyist application to one that can (hopefully) generate revenue. The team has split into two groups and, using a whiteboard, shared all their ideas and iterated over their proposed goal statement. The two groups came up with these goals:

1. Generate $100,000 in advertising revenue by the end of this year.
2. Grow our active user base by a factor of 100 by the end of the third quarter.

Following some discussion between the whole group, there was a consensus to use the first goal and, during discussions, they added some measures to it. So, in our PetBattle example, we have set our goal to: *Generate $100K revenue with our existing and new user base by the end of the year*. We can measure our goal against the current revenue, which is zero, or may even be negative if we include our current hosting costs. We are losing money as we speak!

Setting a goal amongst a small group can be straightforward. What about larger groups? Let's look at some anti-patterns and tips for gaining consensus and inclusivity.

> **Note**
>
> When undertaking any group activity, often the person who shouts loudest is the one who's listened to the most. Sometimes, the most senior person in the room is given the most airtime in discussions – this is known as the **Highest Paid Person's Opinion** (**HIPPO**) effect. This leads to one-sided views being portrayed as the norm and makes people who are quiet or shy often feel unheard. If you're beginning to write your goal, a good method to get everyone's voice heard is Liberating Structures' 1-2-4-all practice (http://www.liberatingstructures.com/1-1-2-4-all/). It's a simple practice to use, but immensely powerful. Start by setting a one-to-two-minute timer and having each individual in the group write down what they think their business goal is. Next, pair up and give another two minutes to the groups to agree on a shared goal between the pairs. Now repeat in groups of four, before finally coming together as a whole, focusing on just one goal:

Figure 8.7: 1-2-4-all Liberating Structures

When all the members in the team align on a goal, it can be very helpful to write it up on a large board or flipchart paper. Having something that everyone can nod their head to and agree *this is why we are here*, *this is our* Why, *this is our shared purpose* makes a very powerful information radiator. Let's remind ourselves of PetBattle's goal on a big sticky note:

Generate 100K in revenue with our new and existing customer base by the end of the year

Figure 8.8: Goal from the Impact Map

The next part of the impact mapping process is to consider Who can help us reach our goal.

WHO Can Help Us Reach the Desired Effect? The Actors

As a group, we identify all the people or groups of people who can help us to achieve our goal. This includes the people who will interact with our new application, either directly or indirectly.

This is a great opportunity to use a diverge/converge approach to identifying new actors. We can create a number of possible ideas (divergent thinking) before refining and narrowing down to the best idea (convergent thinking). We can focus on the people who can help or hinder us in achieving our goal (divergent thinking) and then focus on a few high-value or key actors (convergent thinking). Deciding on which actors are high-value can be a bit of an informed guess, but that's OK; we are placing bets on who can help us reach the desired effect.

Actors can be more than just user groups. They may be a combination of specific types of behavior, for example, extroverts or late-night shoppers.

Let's look at the actors in PetBattle.

PetBattle – the Actors

After some brainstorming and divergent thinking, the group came up with the following list of actors:

- **Uploaders**: These are players of our PetBattle game. They contribute new images of cats and go head-to-head with other players and random cats downloaded from the internet.
- **Animal lovers**: This group of people is fond of cats, cute games, and has time to spend on our site. They play the game by voting, but do not submit their cat photos for battle.
- **Casual Viewers**: These are people who stumble onto our site and represent a group of people we would like to turn into return visitors.
- **Advertisers**: The pet food, toys, and accessory companies who could stand to gain from our active user base.
- **Families**: Groups of parents and children who play on the site and potentially create content too.

The Impact Map starts to take shape by visualizing and connecting these actors to the goal. These groups of people are the stakeholders and interested parties who will help us to achieve our goal:

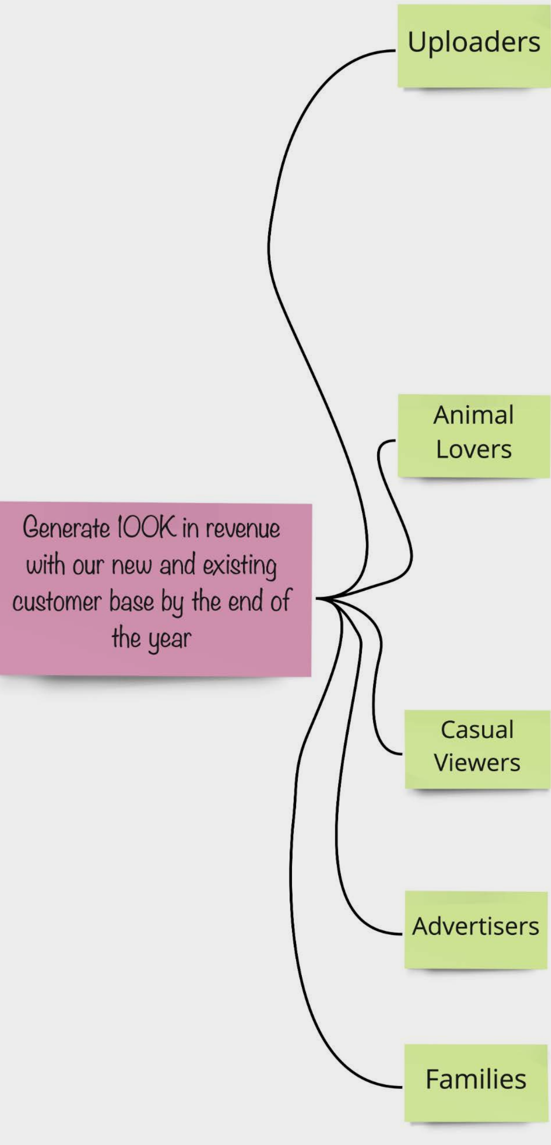


Figure 8.9: Goal and actors

We have now visualized the connection between the goal and all the actors that can achieve or impede it.

> **Note**
>
> The founders of PetBattle struggled when defining the initial set of actors and mistakenly broke the group into authenticated and non-authenticated user groups without tying it back to the goal and the problem statement. This is a classic example of trying to engineer a solution too early without identifying why these classifications exist. Identifying the boundaries between groups is very useful. Sometimes, you can have coarse groups. By splitting them out, you may find additional impacts that are only applicable to those sub-groups.

One of the greatest benefits we see to the impact mapping exercise is all the conversation and shared understanding it generates. When naming actor groups, we find that different people can have different understandings or terms they use. In the preceding example, we may get a short conversation to differentiate between a **casual viewer** and an **uploader**. Some people may have just called these generic users. Having ubiquitous language from the outset can really help with the software development process. This is true for teams that focus on both the technical and business aspects of the product. Already, we are seeing the need to differentiate between different user groups in our software design.

The next stage is arguably the most important and key to the impact mapping practice – defining the measurable impacts we plan to have on each of the actor groups.

HOW Should Our Actors' Behaviors Change? The Impacts

What is the measurable change in behavior our actors need to exhibit to help drive our business results? What do we want our actors to start or stop doing that can take us to our goal?

What could hinder us from achieving the goal?

Let's look at the impacts considered for PetBattle.

PetBattle – the Impacts

For PetBattle, the team spent 30 minutes brainstorming all the impacts. It's ideal if they are measurable. The team came up with several impacts for each of the identified actors:

- **Uploaders:** Increase site engagement and increase the sharing of wins with social networks.

- **Animal lovers:** We want to enhance ease of access to the site.
- **Advertisers:** Increase the desirability of the site to advertisers and increase the number of sponsored competitions.
- **Families:** Decrease site misuse (inappropriate content).

Moving to the graphical Impact Map, we can visualize the impacts we want to have on each actor to help achieve the goal:

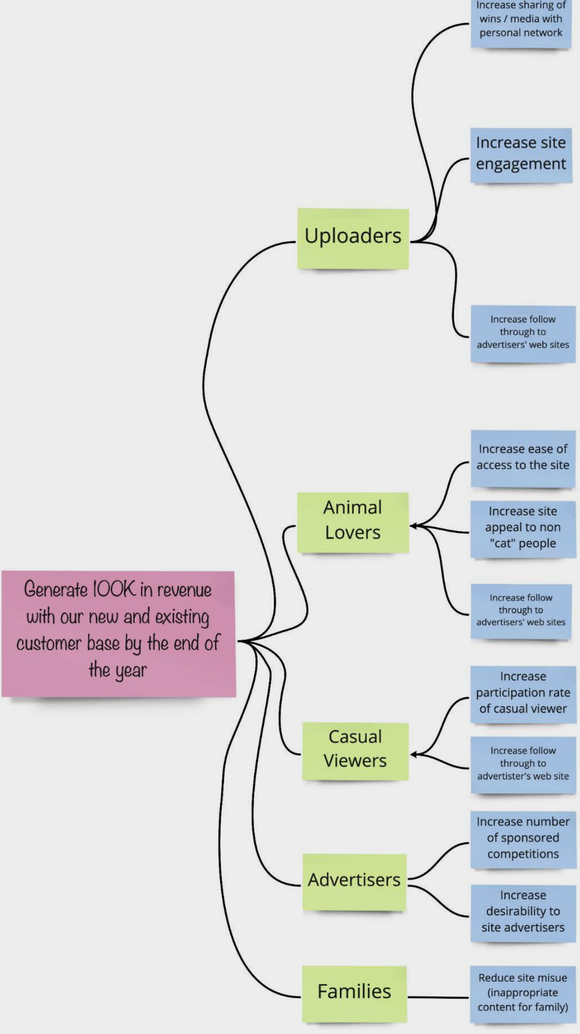


Figure 8.10: Goal, actors, and impacts

> Occasionally, you may discover an impact that applies to more than one actor group. For example, in PetBattle, an overarching impact for both animal lovers, uploaders, and casual viewers is to increase the follow-through to our advertisers' websites.
>
> An impact that spans many actors could be a good place to place a bet as you can have a greater impact across your groups with potentially less effort. It should be noted that the deliverables for this impact could differ for each actor.

We have now visualized the connection between the goal, all the actors that can achieve or impede it and the measurable impact we want to have on those actors.

> **Note**
>
> Always start an impact statement with the term **Increase** or **Decrease** (or **Reduce**). By doing this, we avoid accidentally slipping into the territory of capturing features or outputs when we want to focus on behavioral change on the actors. It also means the impact should be measurable. By thinking about how we want to increase or decrease something, we are quantifying it. We could apply a measure to the impact now (even an arbitrary measure) and come back in the future to see whether that impact has moved up or down.

With the goals and impacts defined, we are coming close to the aspects we are often most comfortable with – the deliverables. The deliverables could be features or groups of features where a feature is a slice of the functionality of the application. What are the tangible things we could do to create this impact? The temptation is to jump straight to features, but by describing impacts as a measurable change in human behavior, we are forced to ask why *feature* X is important and relate it back to our business goal.

Let's now look at the deliverables considered for PetBattle.

WHAT Should We Build? The Deliverables

What are the things we think we could build that would produce the desired impact for a given actor/set of actors? This is where all the things we'd like to implement can be added to the mind map and, more critically, tied back to the business value driving this feature.

This part of the Impact Map is not really the focus of this particular practice and we've seen teams not complete this part. Impact mapping is all about capturing the impacts on actor groups through conversation and collaboration so that we can start to measure what matters. We have other practices that we'll explore later in this chapter that are better at defining the outputs and deliverables. That said, it's virtually impossible not to think about what you might build to achieve these impacts, so it's good to capture them during the conversation.

PetBattle – the Deliverables

From the outset, the PetBattle team were full of ideas on stuff they could do to build the PetBattle app and group the PetBattle business. They'd been written on sticky notes and placed on a *feature ideas* board. They included:

- Social media sharing links
- A loyalty scheme
- A scoreboard for players
- Different animal filters
- A daily tournament
- Discounts for sponsors
- Printing postcards from the site
- Search engine optimization
- Click counting and tracking
- Cross-platform
- A new cat battle notice
- Incentives, charity donations
- Breaking out into non-feline-based battles
- Social logins
- Personalization
- Targeting advertising

...and many more!

Lots of great ideas for features. But can we connect them to having a measurable impact that will drive the overall goal? That's what this part of the impact mapping process will help us to achieve.

Where we can identify an impact that we believe will trigger a defined impact, we place the deliverable and impact next to each other on the Impact Map. For example, the group thought having a daily tournament may well increase site engagement:

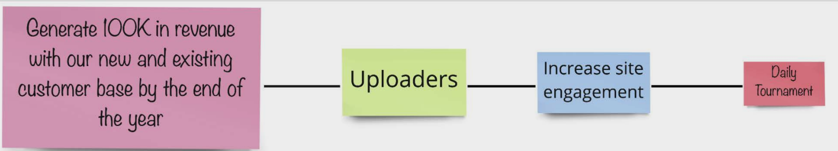


Figure 8.11: First full impact statement

We can translate this to a hypothesis statement:

We hypothesize that if we run a daily tournament, it would increase the site engagement for our uploaders. This, in turn, would help us achieve our goal of growing our user base and raising $100K in revenue.

A second hypothesis statement is generated from the idea of allowing additional advertising space for competition sponsors:

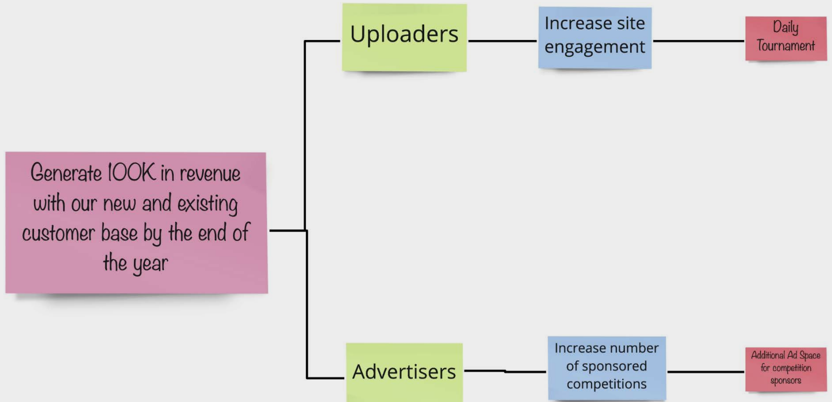


Figure 8.12: Second full impact statement

If we provide premium advertising space, we could increase the number of sponsored tournaments and turn advertisers into sponsors, thereby helping us to generate $100K in revenue.

Some features could not be connected to an impact statement and the team could not think of one. They therefore decided to de-prioritize the idea given there was not a traceable benefit.

The resulting Impact Map was big. It was full of ideas for deliverables all lined up to anticipated measurable impacts:

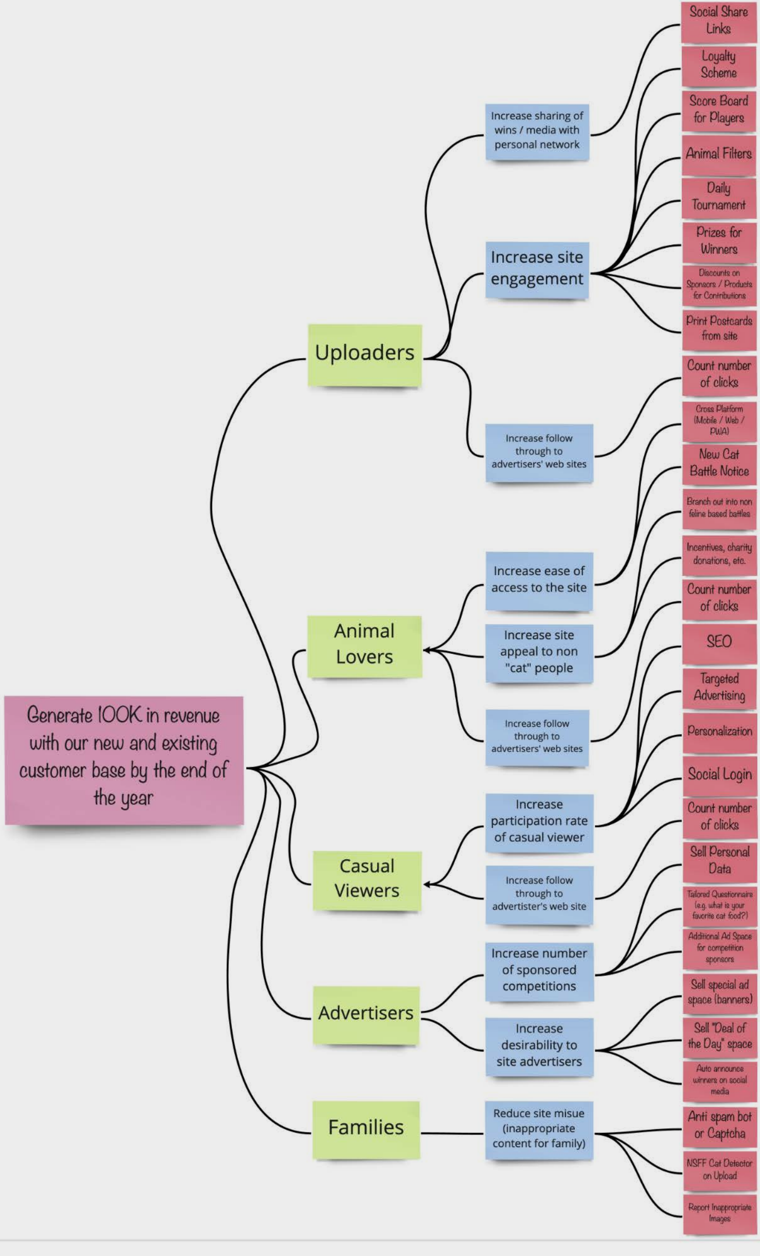


Figure 8.13: The PetBattle Impact Map

Our Impact Map is complete. It connects the **deliverables** (the **What**) to the **impacts** (the **How**) on the **actors** (the **Who**), helping to achieve the overall **goal** (the **Why**):

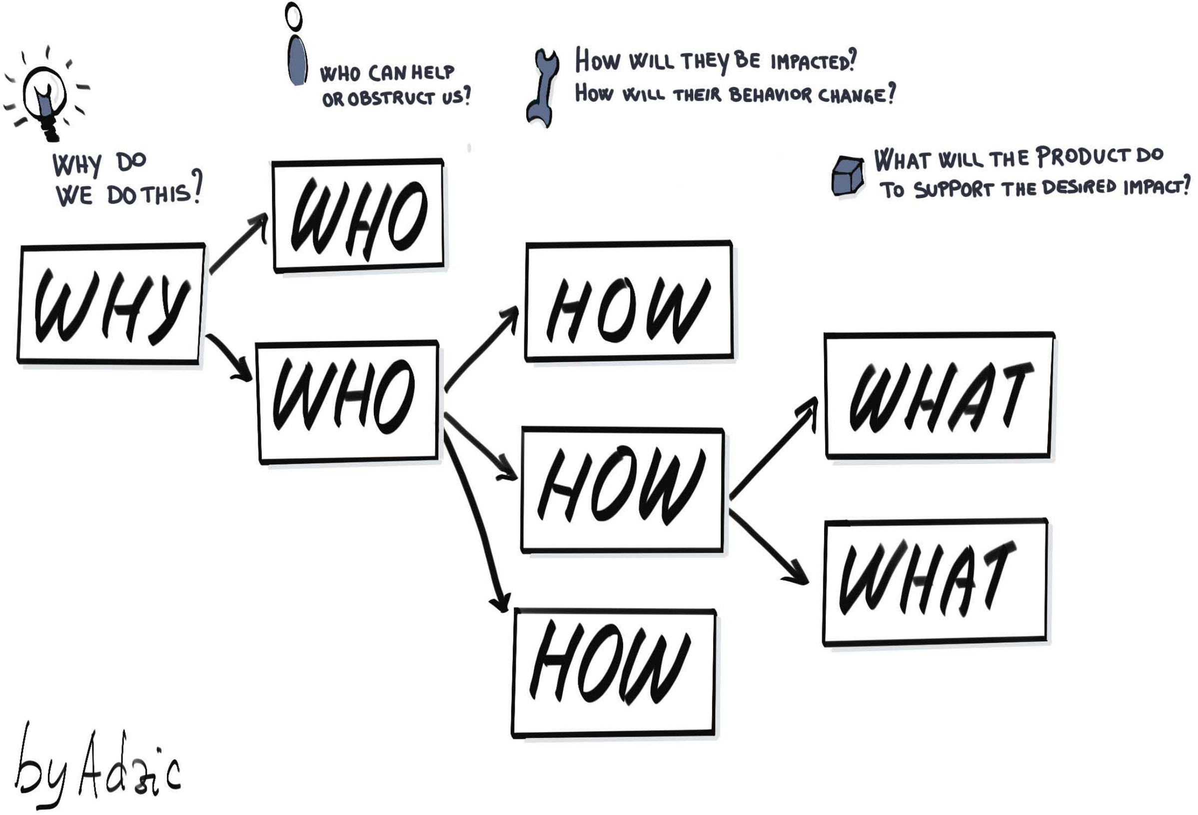


Figure 8.14: Impact Map summary

At this point, we can start to think about some prioritization – at least among the stakeholders who participated in the practice. Using the impact mapping practice, we have come up with a set of hypothesis statements. We don't know whether those statements will turn out to be true. They're just intelligent guesses at this point. To prove or disprove hypotheses, we need to run experiments. Each hypothesis statement can be turned into an experiment. But which experiment do we run first and what order do we run them in?

Time to place your bets! An interesting way to approach this is to ask stakeholders where they want to place their bets. To simulate this, we can give each stakeholder some fake money. $10,000 Beattie Bucks or $10,000 Donal Dollars or $10,000 Mike Money. (Noel, the fourth author, decided he didn't want to enter the currency market!) This is a simple practice to simulate what it would be like if each person in the room was the owner of the company and it was their money.

Attendees of the session now place a bet with their fake money. They bet on which impact statement they would like to invest in an experiment first. They may choose to invest in just one impact or spread their bet across multiple impacts. This allows everyone to pick their top-priority deliverables, showing an aggregated view of what the team should focus on building first.

Let's look into an example to understand this better:

PetBattle – Placing Bets

Each of the four stakeholders investing in PetBattle was given five Donal Dollars and asked to look at the Impact Map:

Figure 8.15: Donal Dollars

They were asked which impact statements they would like to invest in and were encouraged either to spread their money across different impacts or place multiple Donal Dollars on one impact if that was deemed of very high value to them.

Some stakeholders chose to invest all their Donal Dollars on just one impact, while others spread them out. The Impact Map, with Donal Dollars attached, showed a clear winner:

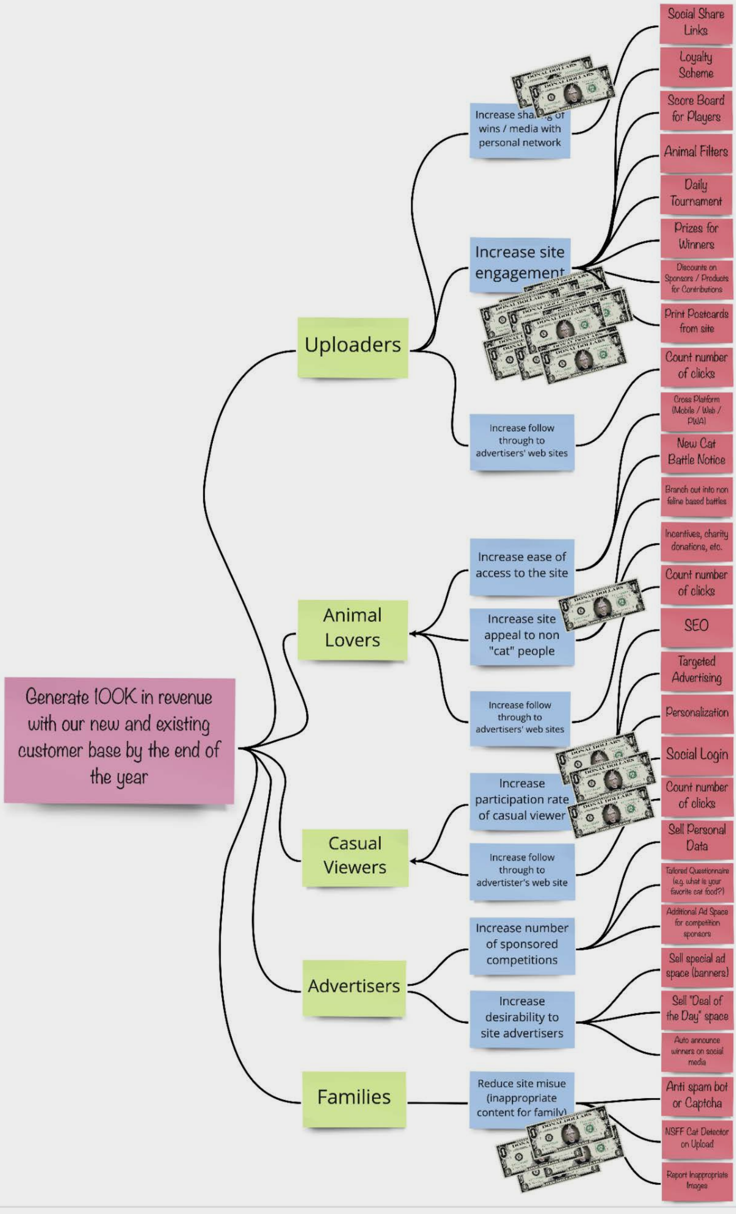


Figure 8.16: Donal Dollars – Impact Map

It was clear that the investing stakeholders wanted to invest in increasing site engagement. The Impact Map already had seven deliverables associated with this, so the team was ready to run some experiments to prove or disprove the associated hypothesis statements.

The stakeholders looked forward to coming back to this Impact Map to see whether their investment had paid off (and maybe spend some more Donal Dollars!)

In the next section of the book, *Section 4, Prioritize it*, we'll look at how we prioritize all the learning and outputs from the practices we used on the Discovery Loop. One of the viewpoints will be the outcome of investment betting on the Impact Map.

A final point before we share a couple of real-world stories about impact mapping is to mention the fact that the Impact Map is never complete. It is not static. It is not a one-time thing. It will always evolve. Teams should regularly come back to their Impact Map after they deliver some features, run some experiments, and learn from their users. They should converse on the results of experiments and conclude whether existing hypotheses statements were proven or disproven.

We introduced psychological safety in the previous section. A helpful test of the current psychological culture is to inspect what happens when a hypothesis statement from an Impact Map turns out to be wrong. A failed hypothesis could impact team morale and result in a lack of psychological safety. If the hypothesis succeeded and created room for further learning and improvisation, it could boost morale and establish an open culture in the team.

Hypothesis Examples

Let's see how we used impact mapping at our Open Innovation Labs residency to help a global bank that was seeking to improve its HR system.

The business goal was for HR to have an integrated workflow across HR systems, providing an improved colleague experience. Some metrics were identified at the goal level, including colleague satisfaction and the time to change a service lead.

The brainstorming of actors identified colleagues, which the group agreed would be the ubiquitous term they would use for all employees, managers, second line managers, partners, and HR operations, as well as management and technology stakeholders.

Measurable impact statements and possible deliverables were identified for each actor group, leading to a number of powerful hypothesis statements, including the following:

We hypothesize that delivering a single sign-on solution will speed up the service, thereby helping colleagues and managers achieve the above goal.

We hypothesize that integrating with a data hub will increase the quality of colleague data available to HR operators, thereby helping to achieve the above goal.

We hypothesize that developing custom dashboard features will increase insights into the management information group.

We hypothesize that designing a new governance model will help the HR specialist speed up the remodeling process.

This is a great example for demonstrating how impact mapping can help drive much larger enterprise systems (well beyond standalone applications) and can also identify outputs beyond software (for example, the governance model is an organizational feature).

Here is part of the Impact Map:

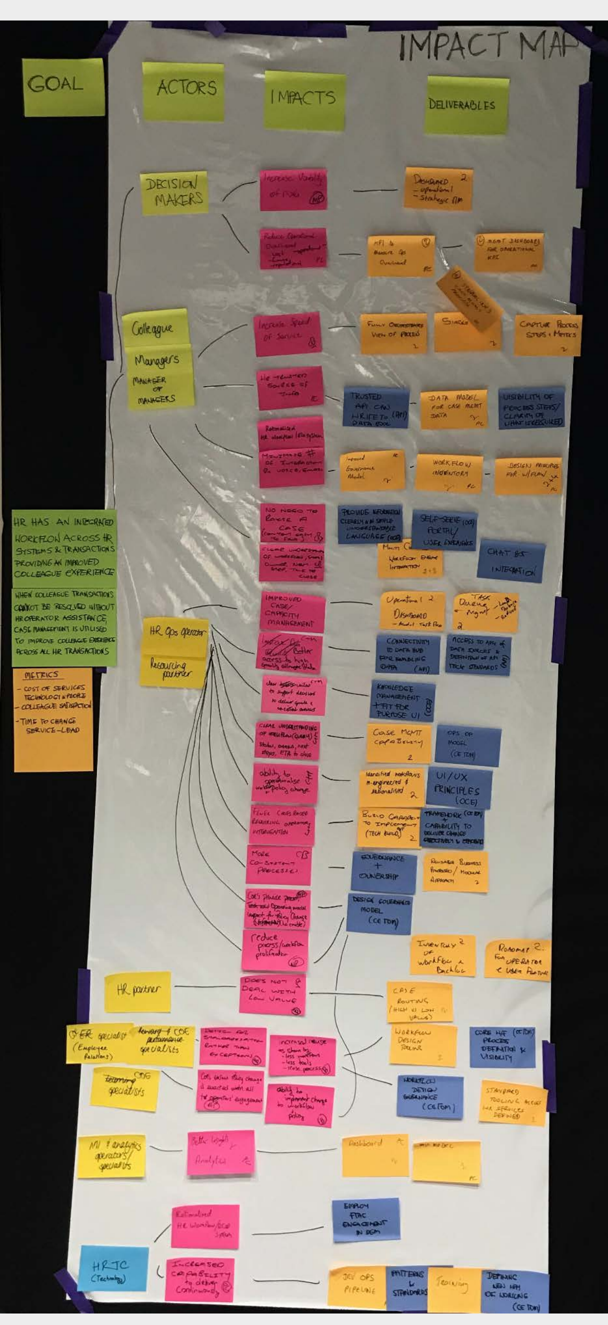


Figure 8.17: Impact Map example

Let's now take a look at another instance of impact mapping.

Connecting Engineers to Business Outcomes

One of my favorite moments of impact mapping came several weeks after the original Impact Map was produced.

Working with a European automotive company to help re-imagine and kick-start the development of their technical support knowledge management system, we used impact mapping to start discovery.

Following three sprints of development, our sprint retrospective led the team to take some actions for the next sprint. The team decided it would be worth re-exploring the Impact Map now that they had done three iterations of development, as we had an app that was being tested with end users.

We brought the big portable board out onto the balcony of the venue we were running the residency from. It was quite a windy day and one lesson I took away was not to bring boards of sticky notes outside when there's a bit of wind!

The proud moment is from the following photo, where an engineer was leading the conversation about a feature he had developed. He was assessing whether the expected impact had been achieved and what they had collectively learned:

Figure 8.18: Revisiting the Impact Map

I thought about the many development teams I'd worked with. It was refreshing to see technical staff so connected with the Why. They were engaged in ensuring that their outputs were delivering measurable outcomes.

For more information and examples and to have further conversation with community members about the Impact Mapping practice, please visit the Open Practice Library page at openpracticelibrary.com/practice/impact-mapping

Impact mapping is, without doubt, one of the best practices to start your discovery conversations. What else could we do to enhance this? We're now going to focus on the actors' perspective, in other words, what the product could do for them, using Human-centered design.

Human-Centered Design

Impact mapping identified all the different actors and groups of people that can help achieve our goal. It also identified the measurable impacts we would like to have on them. Wouldn't it be great if we could actually meet and engage with some of these people? We could validate our Impact Map with them. We could check whether the deliverables we have in mind would be supported by them. We could learn from them before we write a line of code! We could build a trusted relationship with them whereby we regularly check in with them, test, measure, and learn from them as we start to build out applications.

Human-centered design is an approach to system development that aims to make systems usable and useful by focusing on the users, their needs, and requirements.

There are several major topics that relate to **User Experience** (**UX**). Each of these is large enough to fill books of their own and include:

- Design thinking
- Human-centered design
- The Double Diamond process model
- Lean UX

In the software development world, we can start with people and end up with innovative applications that are tailor-made to suit our users' needs. By understanding our users and looking at things from their perspective, we can ultimately produce a product that they will love to use and engage with.

People react to what they see – visual design. However, there are many layers below the visual that UX designers must consider in order to make an application engaging and useful:

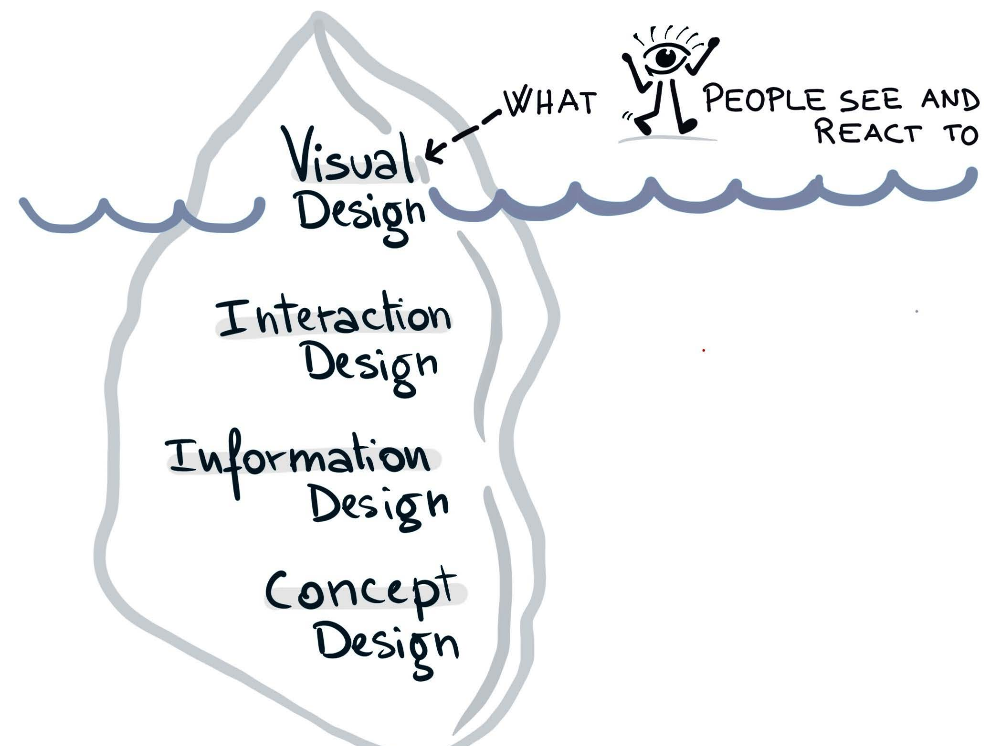


Figure 8.19: The UX design iceberg

Let's explore some of the different components of UX design that should be considered in product development:

- **Facilitation of product discovery** involves leading the process of finding a suitable product or service, focusing on product desirability, the viability of a business model, and the feasibility of technical possibilities.
- **Qualitative user research** includes running interviews with users and performing contextual inquiries and other observation studies whereby the designer learns by watching users carry out their current job.
- **Conceptual design** facilitates and contributes to the creative work at the early stages of a product or service formulation.
- **Prototyping** ranges from paper prototypes to low-fidelity prototypes, to higher-fidelity mockups that can be used to test ideas with users and other stakeholders.
- **Interaction design** starts to look at what the new experience could look like by using visual storyboards and wireframes.
- **Human-centered service design** takes a wider, macro-based approach to performing customer journey mapping, service blueprinting, and increasing the quality of all touchpoints within a service delivery.

- **Quantitative data analytics** processes feedback from tools such as Google Analytics and A/B testing.
- **Visual design** is the design of the visual elements that users will see and encompasses the learning from many of the above practices.
- **Accessibility of IT services** considers web content accessibility guidelines and other non-functional aspects.

The common denominator of all of the above practices is learning through better information and understanding of the user stakeholder. This will validate and improve the predictions made during, for example, finding the North Star and impact mapping.

A popular practice that we use at many different stages of UX design is empathy mapping. The empathy map is a visual canvas that captures what a stakeholder is seeing, thinking, doing, and feeling within a certain context. The empathy map can be used to surface out pain points and opportunities for improvement. These then become an important source of value propositions that we can consider in our product development.

We use empathy maps during the facilitation of product discovery, user research (for example, to capture notes during a user interview), getting early feedback on conceptual design, prototypes, and visual designs. It can also be used to capture feedback regarding the user testing of developed application features:

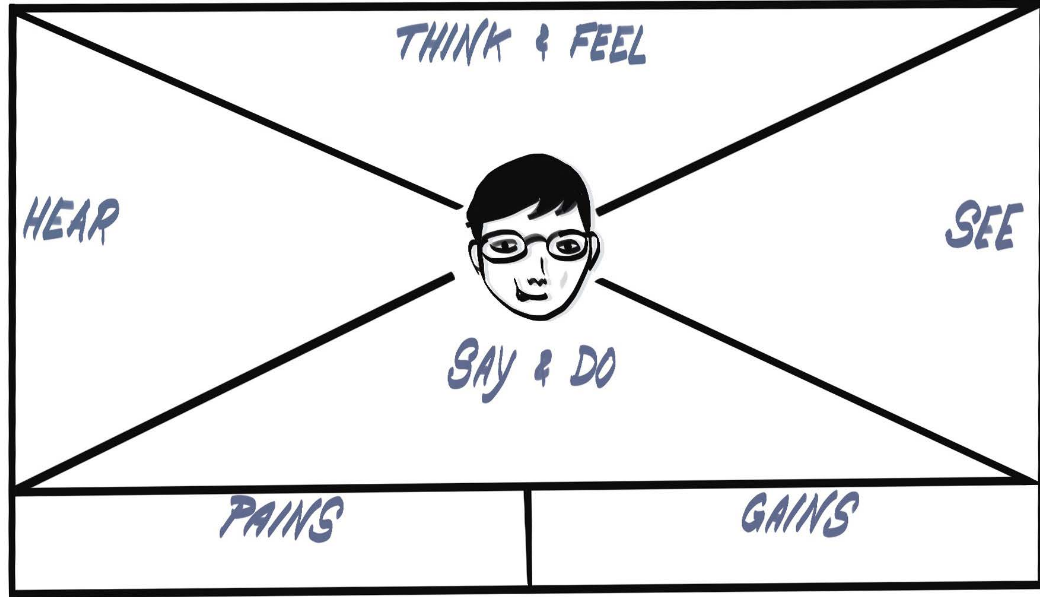


Figure 8.20: The empathy map

One of the simplest human-centered design practices that employs rapid prototyping is called Think, Make, Check. This involves creating a rapid prototype and interviewing users about it. We use iterative steps to get to a design. During the user interview, the team can listen in to help complete the empathy map:

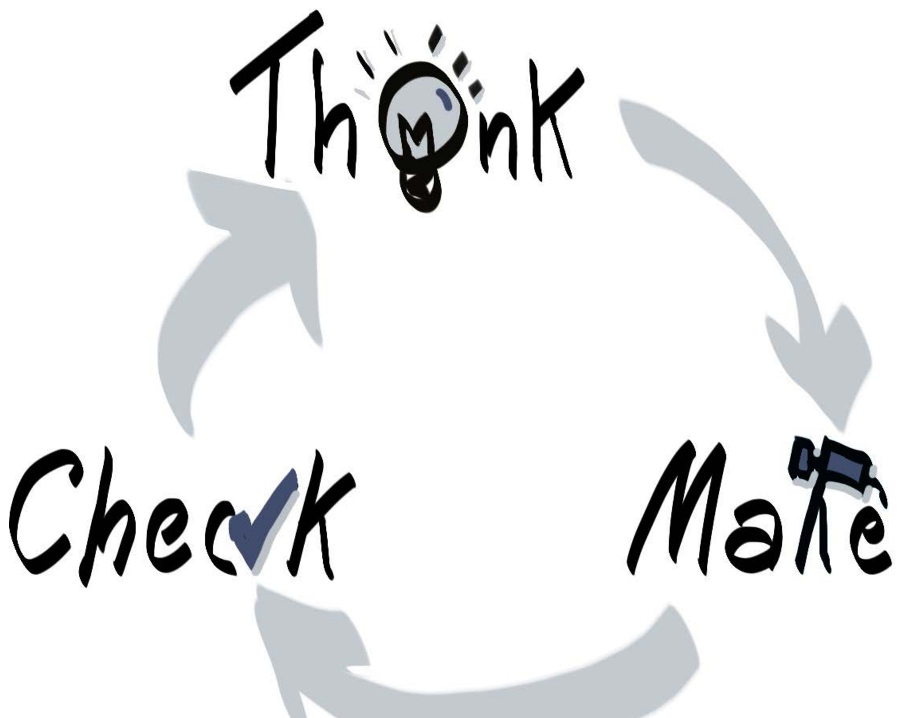


Figure 8.21: Think, Make, Check

In *Section 2*, *Establishing the foundation*, we introduced Mary, and we explored how the PetBattle organization created autonomy, mastery, and purpose. Mary is a self-confessed cat lover who played PetBattle regularly when it was a simple hobbyist app. She connected with Ciarán (a member of the PetBattle engineering team) who observed her using the application. Let's now see how Ciarán used an empathy map to capture all his learnings.

UX Design and Empathy Mapping a PetBattle User

Ciarán met Mary after she shared on Twitter how much she was loving PetBattle. They exchanged a few direct messages where Mary alluded to not fully understanding some of the quirky user interface aspects. She also felt PetBattle has great potential to move to the next level in online time consumption!

Ciarán thought Mary could really help shape the thinking of the next iteration of PetBattle and validate some of the team's early assumptions. So, in return for a $20 Starbucks voucher, Mary agreed to a user interview. The rest of the PetBattle engineering team and the product owner sat in on the interview.

Ciarán opened the interview by asking a few simple questions:

1. What is your name?
2. What do you do for a living?
3. Where do you live?
4. Do you own a pet?
5. How did you learn about PetBattle?
6. How and where do you tend to play PetBattle?
7. What pet would you like to use in PetBattle?
8. When online, what do you do when you *like* something? (When Mary struggled to answer, Ciaran gave examples of a *thumbs up* or *like* button.)
9. How would you like to battle against someone else's pet?
10. What would make you use PetBattle more?
11. Does the use of theming matter to you?

Ciarán then asked Mary to play the old version of PetBattle so the team could see her interact with the application.

She uploaded a photo of her cat she had on her phone. She then struggled to understand why her cat was not appearing immediately in the competition once she'd uploaded it. Ciarán and the team knew she had to vote for another cat before hers would appear on the list, which seemed obvious to him.

The team observed a few other surprising things about how Mary interacted with the application (and some of the things she didn't see!).

Ciarán and the team thanked Mary for her time and asked whether she would be interested in returning in a week or so to help with some prototyping. Mary was glad to help.

Meanwhile, the team had been building Mary's empathy map which they would evolve and later play back to her to confirm their understanding and learning.

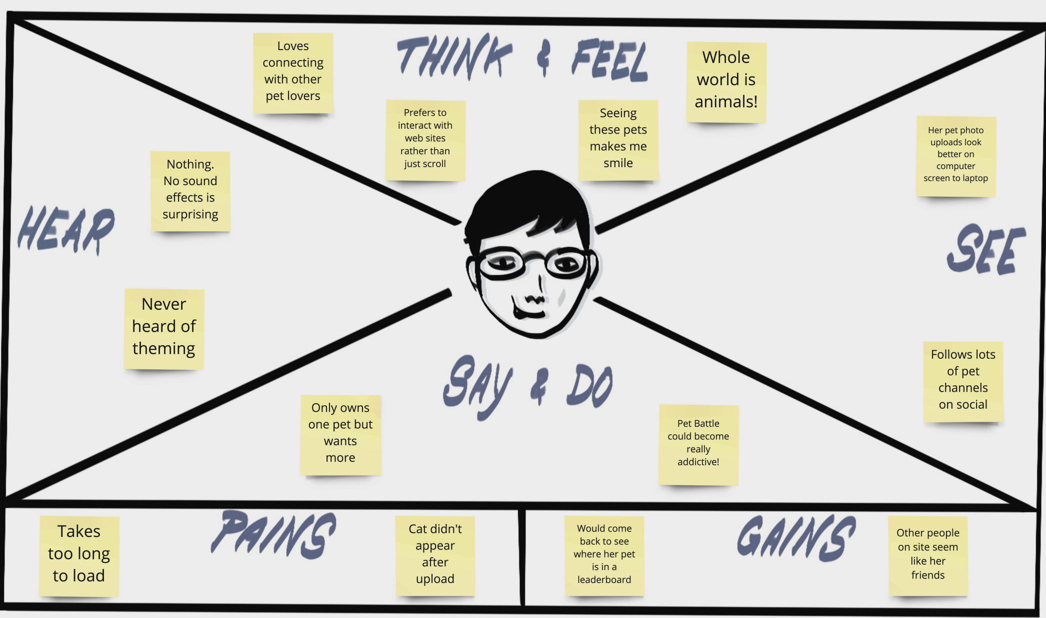


Figure 8.22: PetBattle user's Empathy Map

A week later, Mary returned to PetBattle. The team had been busy and produced lots of paper prototypes of some of the ideas that had come out of the impact mapping session. This included a discussion forum, targeted advertisements, a deal of the day, the scoreboard feature, and a loyalty scheme.

The prototypes ranged from sticky notes with sketches on to higher-fidelity wireframes. The scoreboard feature had actually been mocked up using the Sketch visual design application.

Different members of the team explained each idea and the prototype. Mary entered into the conversation and plenty of time was given for her to comment, feedback, query, challenge, and add her own ideas. The team who was not sharing the prototype followed the conversation with a big portable board that had Mary's empathy map on.

Again, Mary had been immensely helpful. Ciarán reckoned the details she had shared were gold dust and they never would have considered some of the aspects she was challenging.

Mary agreed to revisit every two weeks going forward so she could be the first to see and test the new features developed. This would coincide with the end of each delivery iteration and the team decided they would continue to empathy map Mary to determine what she likes about new features and what doesn't work so well and should be changed.

Here's another case study of this very effective practice:

Users Do Strange and Unexpected Things

As a software developer, you may look at a practice such as empathy mapping and think it sounds kind of fluffy, so why should I care about it. The first time I heard about this practice being used, I was skeptical; I couldn't connect why I should care about it as a developer. But when I realized I was missing out on so much rich contextual information regarding what users do, I was completely sold.

I worked with a Rockstar designer, Bríd, building out a replacement for an HR workflow engine that was coming to its end of support with a banking customer. We used this practice against the as-is application with Vijay, one of the HR Ops team members who was responsible for responding to tickets being raised and assigning them to his team.

While Bríd interviewed Vijay, the rest of the team captured notes and put them on an empathy map. We captured what activities Vijay would do using the tool and how he felt about them. The key for us was identifying gaps in the current tool and some of the shortcuts he might take. Through this research, we discovered a massive gap in the solution they had been using up to this point.

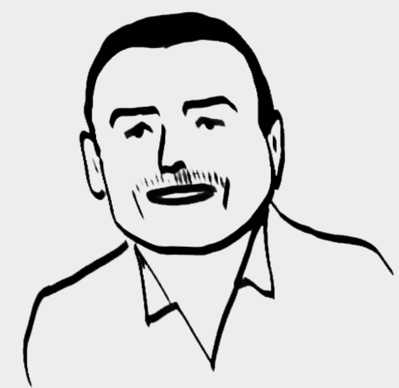
Vijay, in his day-to-day use of the current application, was cutting data out of the app and opening up an Excel spreadsheet. Here he would paste a big dataset to sort and manipulate it in batches before copying it back into the tool! We were all pretty surprised to see this and it showed us that there was a huge gap in our understanding of the type of activity the current tool was built for. Had we not run this exercise with Vijay, I don't think we'd ever have found this out. Vijay and his team were going out of the app as there was critical functionality that was required in order for them to be effective.

When I saw this, it was definitely a turning point for me. I could see first-hand how getting into the mind of a user was just as important for designers as it was for the techies in my team.

Later, when we were building out some functionality in the app for batch updates, I was thinking of Vijay in my head. I strongly believe that if we had not used this user-centric practice, which can help uncover genuine user behavior, we would not have discovered the new functionality we needed to design and build for.

While working for a Finnish telecoms company, our team used this practice to bridge the gap between the Dev and Ops teams:

Empathy Mapping an Organization — Dev versus Ops



An empathy map is a great tool for identifying the pains and gains of an existing or newly built application, especially ones that have a user interface or some elements of human interaction. It really can help get into the mind of some users and how they're responding to new functionality. I have found another area where this can be used – empathy mapping silos within your organization.

Dev and Ops were traditionally two silos within lots of organizations. Developers can be perceived as throwing their work over the fence where the Ops team is supposed to pick it up and run with it. Sometimes, this can create friction between the two sides of the wall, often creating a blame culture with each side pointing fingers at each other as things go wrong. While working for a Finnish telecoms company, I tried this practice out in conjunction with a demo the Ops and Dev teams gave me of how they build, deploy, and monitor their applications. It worked quite well and gave me some insight into how each team was judged by the other.

The image here is the empathy map from Kalle – our developer. From our session, we discovered that he felt overworked and that he was unable to go fast because of the slowness of the other team.

Figure 8.23: Creating an empathy map for a Finnish telecom company

He would raise requests and the Ops team would be slow to respond, impacting his ability to progress with any agility.

And of course, on the other side of the divide, we had Timo in Ops, and his view was different from that of Kalle, our developer. Timo's empathy map revealed that it was a challenge for him to have to keep repeating himself to the developers and that requests coming into him were never accurate, so he'd have to push back or do lots of reworking.

We played both empathy maps back to each team to start to ideate how we can work together in solving this divide.

It's a simple practice to use and the information it revealed could be viewed as predictable, but from here I could start to see where the pain points lay within the company structure. It became super clear that the teams were not sharing the responsibility of delivering new functionality or fixes, but instead blaming each other when things went wrong.

Empathy maps play a vital role in acquiring a deep understanding of end users. Through the following case study, we discuss our take on this practice.

Engineers Build Out Empathy Maps during User Interviews

It's a misconception that UX design is for UX designers to run by themselves. It's also a misconception that UX design should be done as a standalone phase of a project.

Some of the most powerful UX designs I've seen are where the engineering community, developers, and operators have been fully engaged in the process.

One example occurred at our Open Innovation Labs at Red Hat in London. During this residency engagement, our UX designer led the interview in a nicely equipped interview room in the Red Hat building. With the interviewee's permission, the whole interview was screen-cast live to the Labs space, where all of the development team were watching and capturing vital information on an empathy map.

In the following photograph, you'll see the team has split into two groups, each watching two different live user interviews and building their own empathy map:

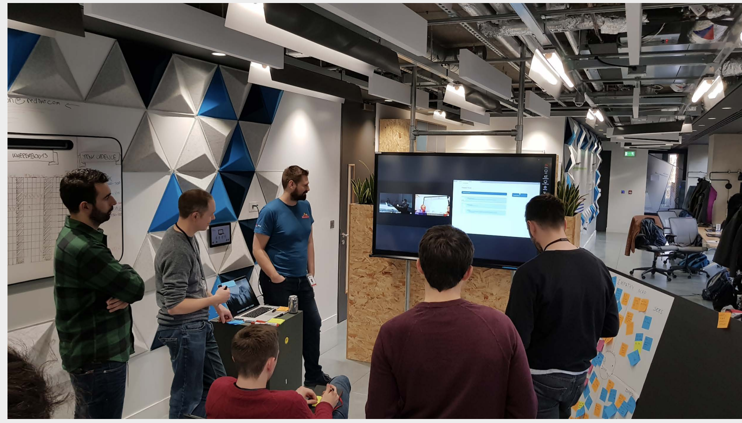


Figure 8.24: Engineers creating empathy maps

There are several practice pages in the Open Practice Library that relate to human-centered design practices. These include:

- openpracticelibrary.com/practice/aeiou-observation-framework/
- openpracticelibrary.com/practice/proto-persona
- openpracticelibrary.com/practice/stakeholders-interview
- openpracticelibrary.com/practice/empathy-mapping
- openpracticelibrary.com/practice/lean-ux-workshop
- openpracticelibrary.com/practice/ui-design-workshop

These pages provide an excellent overview and starting point to explore these practices in more detail.

In the next chapter, we will take a look at a discovery practice that can help our team visualize the end-to-end business process in a way that everyone can understand – the Event Storm.

Conclusion

In this chapter, we have been on the Discovery Loop with a focus on discovering the Why and Who of the product - Why are we building this product and who are we building it for? We started by finding our North Star Metric - a crucial metric that would be an indicator of product success.

We introduced the Impact Mapping practice which helps us converge on our overall goal and understand all the different Actors that will help or impede us reaching this goal. We also learned how to define and map measurable impacts and deliverables that we can work with our Actor groups to form hypotheses statements as a basis for experimentation.

We explored the different human-centered design practices that can be used with the Actor groups to form empathy and support user experience design.

Figure 8.25: Adding practices to the Delivery Loop to discovery the Why and Who

In *Chapter 9, Discovering the How*, we will continue on our Discovery Loop but focus more on discovering how we might build our solution. To be specific, it focuses on how to start the architecture and the architecture will emerge over time. We'll be exploring domain driven design and event storming as practices that help us do this. We'll also look at practices that help with non-functional aspects of our architecture - the non-functional map and metrics-based-process-mapping.

Discovering the How

In the previous chapter, we started our first iteration of the Discovery loop. We focused on practices that helped us discover why and who we're trying to solve problems or realize opportunities for.

In this chapter, we will shift our focus onto collaboratively learning how we might start building a solution that fixes problems for these people or realize some new opportunities for them. The practices that we're going to explore to help do this include:

- **Event storming**: A colorful practice that visualizes business processes into event-driven domains with an emerging architecture
- **Emerging architecture**: A principle we follow whereby we capture just enough information from the Event Storm and other artifacts to start our architecture and allow it to emerge over time as we add more complexity
- **Non-functional map**: A practice that facilitates conversations to explore different non-functional aspects of the solution
- **Metrics-based process map**: A practice that allows us to capture some baseline measurements of legacy systems and identify bottlenecks in systems and that provides a strong case for moving to a more continuous delivery mode of delivery

Collectively, one of our favorite practices is event storming because it is so collaborative and inclusive. It provides something for every role in a team. Customers often tell us that what we achieve in a few days using event storming would have taken months to capture in a business analysis project.

Event Storming

Event storming is an agile and lean way of getting teams to collaborate on technical domain projects, combine thoughts, and learn to work with a shared understanding.

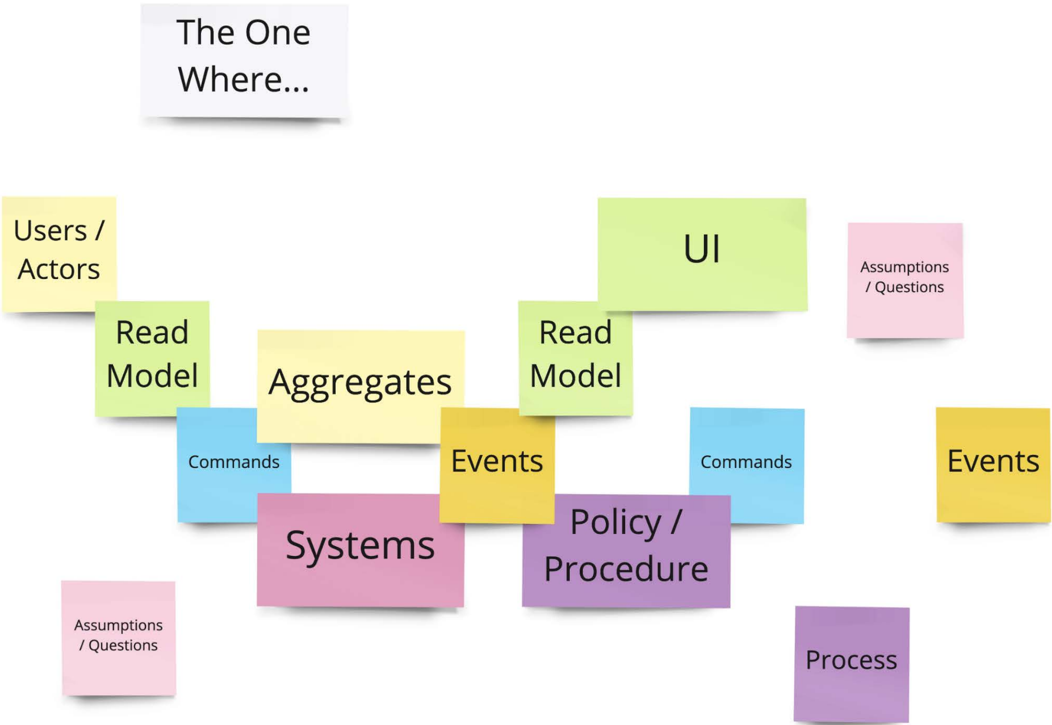


Figure 9.1: The Event Storm color puzzle

Event storming is a modeling workshop for visualizing processes within a business or a system. It can give perspectives from differing heights within an organization. At the business operation level, Big Picture Event Storming can identify gaps and areas of frustration between organizational structures, highlighting where tradeoffs exist and thus highlighting areas that need improvement. We can zoom in to a lower level of detail by using business process modeling through to application feature modeling. Collectively, these allow developers, designers, end users, and business stakeholders to speak a shared language where complicated technological wizardry is not needed.

While at first the process can seem complex, it has become one of our favorite practices at Open Innovation Labs to use with our customers. The short time taken to break down silos of knowledge and visualize a business process with this technique is what impresses us the most. Event storming is the technique we use for gaining a shared understanding of how a business process operates while also visualizing any potential issues and solutions. Having facilitated many event storms, this part of the chapter is our guide to carrying one out and our commentary on how best to apply it.

Firstly, you need a big modeling surface, ideally a long wall with much more space than you can imagine using. You also need plenty of space for people to stand around and see the modeling surface.

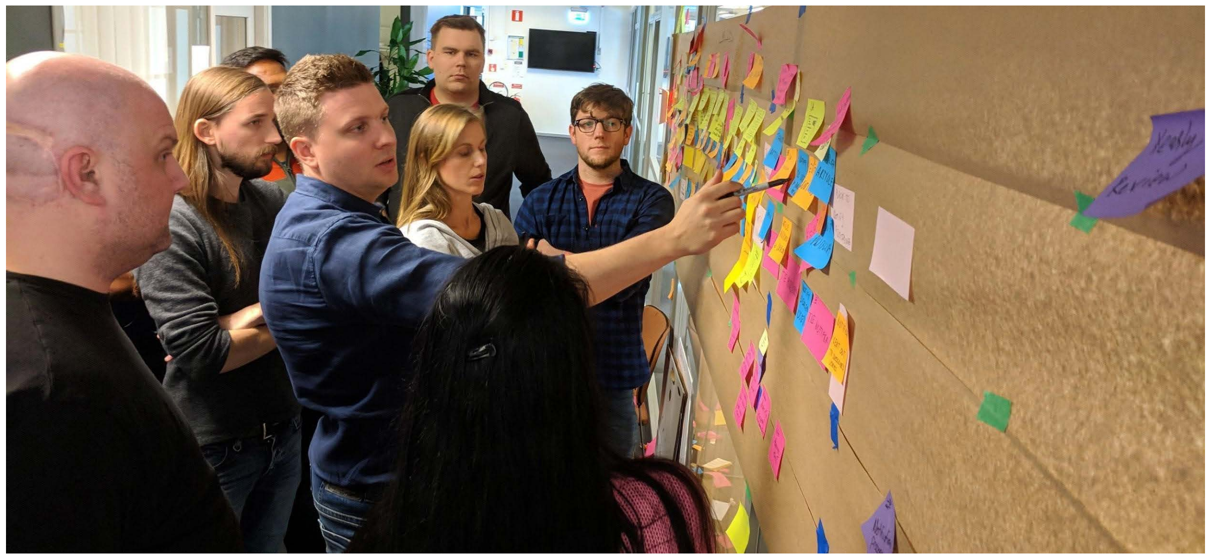


Figure 9.2: A group storming around the modeling surface

What Is Event Storming?

Event storming was created by *Alberto Brandolini*[1] and has been documented on numerous blog sites, Wikipedia, and YouTube. We will try to summarize what we have learned while using and facilitating this practice.

1 https://medium.com/@ziobrando

Figure 9.3: A full Event Storm detailing two customer journeys

In essence, event storming is a group of people having a conversation, discussing the business objectives, and capturing these things on sticky notes. If you invite the right people, the conversation and visual mapping will help emphasize dependencies that may have previously remained hidden. Highlighting these dependencies can help teams to avoid making the wrong decisions about a product from a technical and business perspective.

Event storming is a pattern for exploration and discovery. It comes from the world of **Domain-Driven Design** (**DDD**). DDD is a language- and domain-centric approach to software design for complex domains. We like to think of event storming as being a stripped-down DDD – DDD-Lite – but with more business focus and less of the jargon and complexity. By bringing business domain experts together with developers, we can collaboratively build process models by applying a common, ubiquitous language that enables developers to model an event-driven system and discover microservices.

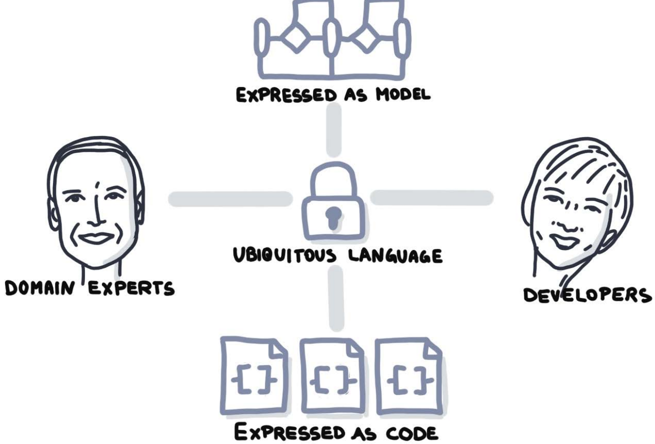


Figure 9.4: Event storming approach and people

An Event Storm workshop gathers people from across the organization. It does not focus on any specific technology, so those skills can be left outside for the time being. The only requirement of attendees is their energy, attention, and willingness to give it a go. During an Event Storm workshop, everyone is armed with orange sticky notes – the **events** – and the knowledge about their part of the company that they bring with them.

Software creation is an exploratory task, and while exploring, more learning occurs. Capturing this is critically important. Event storming is about visualizing all that knowledge as an event-based mind map and identifying the gaps, unknowns, and pain points within. With the right audience for an Event Storm, you can get harmony between groups who traditionally might never meet and, more importantly, bring alignment where previously there may have been misunderstanding.

In an Event Storm workshop, you may have the business analysts, who know the business needs and demands, identifying the **commands** and **events** alongside the developers, who will implement these as features. Couple this with having the UX designers (working with end users) doing UI design and data model validation to help support this and you all of a sudden have alignment from end to end. You also get early verification of what could work and what won't work before you write a single line of code.

For a more comprehensive background of event storming, check out the Open Practice Library, where you will find more links, articles, and examples of event storming being used in the field with our customers. We are going to focus on an Event Storm at the process and feature level.

Let's take a look at what you need in terms of ingredients to run a great Event Storm.

The Ingredients

The Event Storm ingredients are all the things you need for a successful Event Storm.

First up, you need sticky notes and you need lots of them. Think of a very large number of them, then double it. Event storming uses a very specific color-coded key. It is important to stick to the authors' colors as it gives consistency as you move from Event Storm 1 to Event Storm 2.

Next, you need energy. Event storming is an intense workshop that requires people's attention the whole time. Ironing out misconceptions requires the right people and for them to be able to speak out. Bring lots of good coffee and plenty of water and fruit!

You will need a good open space. Give yourself an unlimited modeling surface. Roll out some plotter paper, or if you're struggling for wall space, use big foam boards that can be moved and added to. You don't want to constrain the amount of information gathered by lack of wall space.

People are the most important ingredient. Invite the right people. Get your end users, the business analysts, architects, business owners, product owners, and developers. Get all the people. The more people you can get, the more fleshed out your Event Storm will be. A great event storming coach and facilitator will seek to bring everyone who knows the product and everyone who is collectively involved in building the product to the Event Storm. If that's too many people, they'll look for representatives from each department or team across the organization that makes up this cross-functional group.

We have created an Amazon shopping list[2] for those looking for a shopping-list-as-code Event Storm experience. Once you have all the ingredients, let's take a look at the recipes to cook up a great Event Storm.

The Recipe

The event storming recipe is the list of things to do and the order to do them in!

First up, you need to preheat your oven. By that, we mean hack the space. Remove all the chairs from the room and mark a big, empty wall space for you to roll out the modeling surface. Chairs lead to people sitting down, which leads to people not participating, which leads to people falling asleep! Give yourself lots of room and roll out your plotted paper/foam boards. Do not start in a corner; start in the middle of the room if possible. Starting in a corner will mean only 50% of the audience will be able to gather around it, therefore removing half of the knowledge being offered.

Figure 9.5: Rolling out the modeling surface

2 http://amzn.eu/dViputa

Bring out the **Event Storm Flow**, which is a key or legend to the Event Storm. We usually make posters (on large, flip chart-sized sticky notes) the night before running the workshop, as this can be time-consuming. Hang it so it is within eyeshot of all the attendees. People who are new to event storming will want to keep referring to it to guide them initially.

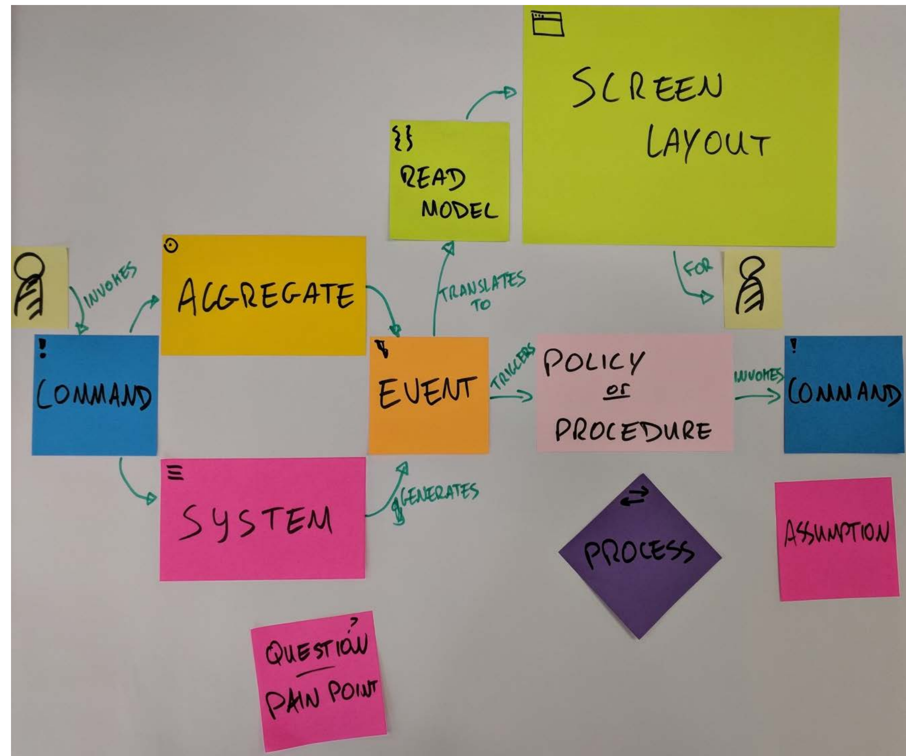


Figure 9.6: Event Storm Flow

A more digitized version of this could be added to a digital tool (such as Miro or Mural) if you are going to try to run the Event Storm with distributed team members.

We normally walk through the flow of the Event Storm at this stage with the attendees. This can be fairly overwhelming as there is a lot to introduce. We create and walk through an example based on something fabricated but relatable to everyone present to help hammer home the key points. Usually, we will go for either a to-do list app or something more tangible such as an online Amazon purchase. Everyone can relate to these.

With the ingredients all laid out and the oven preheated, it's time to start creating the mixture! We start with **Events**.

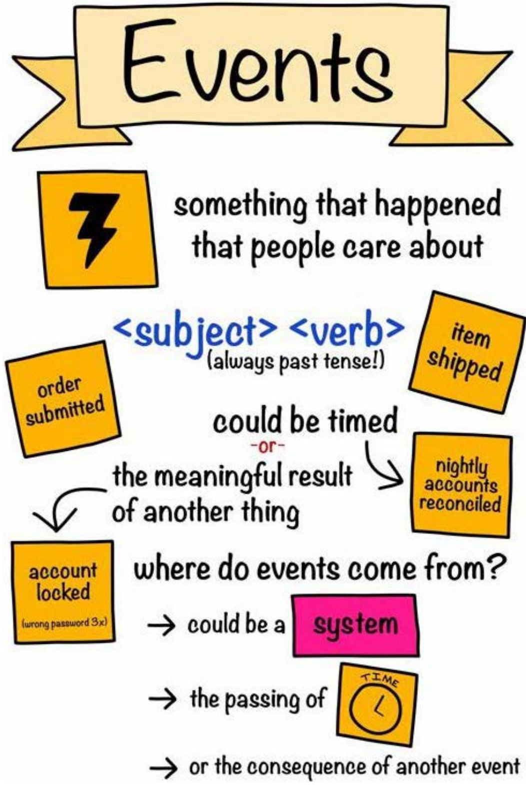


Figure 9.7: The Events key

The first thing we do is set the goal of the event storming workshop. This could be the entry point of the flow, the endpoint, or both. Introduce the event poster shown in the previous image. In its simplest form, an event is just something that happened in the past that someone cares about. With the starting point set, ask the attendees to divide into groups of two to three and identify all the events they can think of in the system. Timebox this activity. We usually go for 15 to 20 mins initially and hover around making sure people are on task and conversing, clarifying things if needs be. If there are SMEs in the groups, we make sure they are divided into the groups equally and not bunched together in one. The SMEs are often in the best position to guide this initial building out of the event spine because they understand a lot about the domain in question.

With the team's events identified, ask for a volunteer group to play their events back to the group by adding them to the modeling surface and begin telling the story. Enforce the timeline of the events, moving from left to right. The first person in a group to volunteer their spine should be rewarded with a gold medal for breaking the ice. We often draw a medal on a sticky note and award it to this person so they can wear it with pride! A top tip is to not start fully on the left-hand side of the modeling space. When things start to get moved around as more knowledge is uncovered, this space can be useful to grow into.

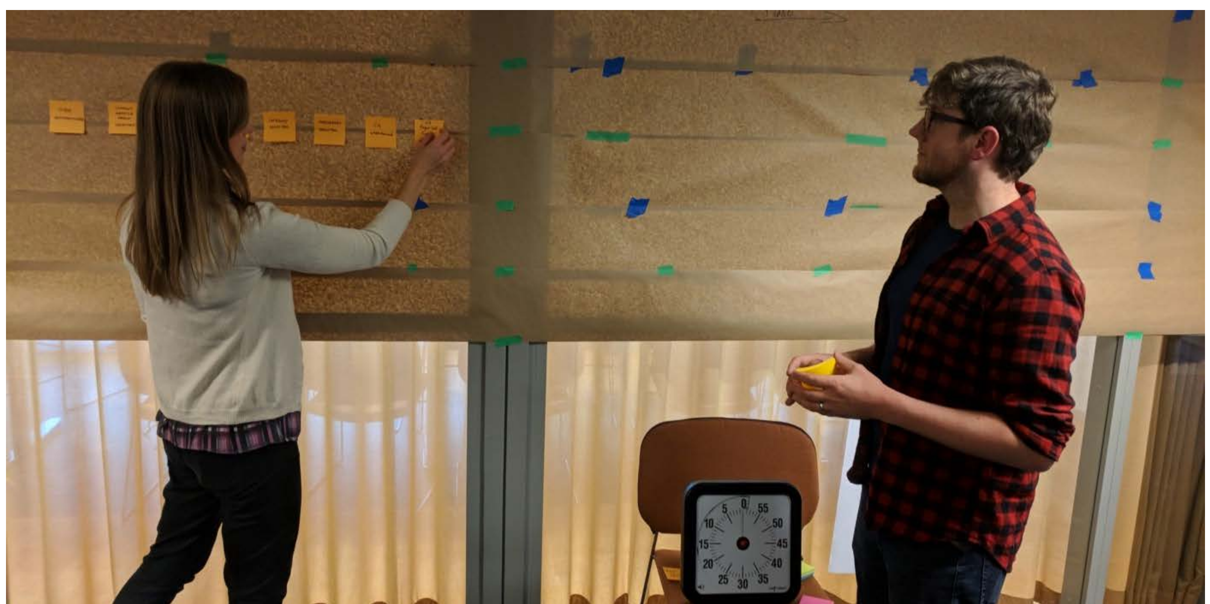


Figure 9.8: Hanging events on behalf of her team

Ask the other groups to add in their events along the timeline. Encourage the attendees to shuffle theirs along the surface to make room for more events to follow. If teams come up with different words to describe the same event, try to get a consensus on the language. Clarify events that are not fully developed. For example, if a group has a very high-level event such as *Item Ordered*, break it down into lower-level details such as *Item Added to Basket* or *Checkout Opened*. If there are any questions or assumptions being made, mark them if they cannot be answered just yet.

It's important to park the things that cannot be answered confidently and move on or you can end up in the weeds very quickly. Ensure enough time is given to the discussion in case the answers can be uncovered, but if not, mark them with a pink sticky note representing a question. Initially, the **question** key will be used very frequently. That is OK; this is the point where we have the least collective knowledge of the system. When marking an area with a question, we are stating that the conversation will not be forgotten and can be returned to when needed. A good idea can be to hold off on revealing this card until enough conversation has been had and it is needed in order to move on. Be sure to capture all sides of the discussion using multiple cards if need be.

With the spine of events created, have the teams play through the story from front to back and back to front. Add any missing events through the conversation that occurs. At this stage, we like to mark **pivot events** with some tape coming vertically from a single event. These can be useful in framing the boundaries between key things and can be great for quickly identifying key sections of the timeline.

The next section introduces the bulk of the keys in an Event Storm puzzle. It's time to introduce the **actors**, **commands**, **read model**, and **systems**. Add the charts for each so they're within eyeshot. Walk through them with the group and clarify if anyone has any misunderstandings.

The command represents a decision made by a user in response to some information retrieved from the read model. The actor is the person who issues the command and the system is the thing that receives the command. It is the responsibility of the system to respond to the command and therefore trigger an event.

There is quite a lot introduced in this section, so its important people do not get overwhelmed with the next pieces of the puzzle. It can be handy to go through another simple example if the audience needs something more relatable. We tend to add a few of these pieces to the spine so the example is now in the context of the flow being created by the wider group.

The next part of our key provides a mechanism to capture **Questions** and assumptions as well as **External Systems**.

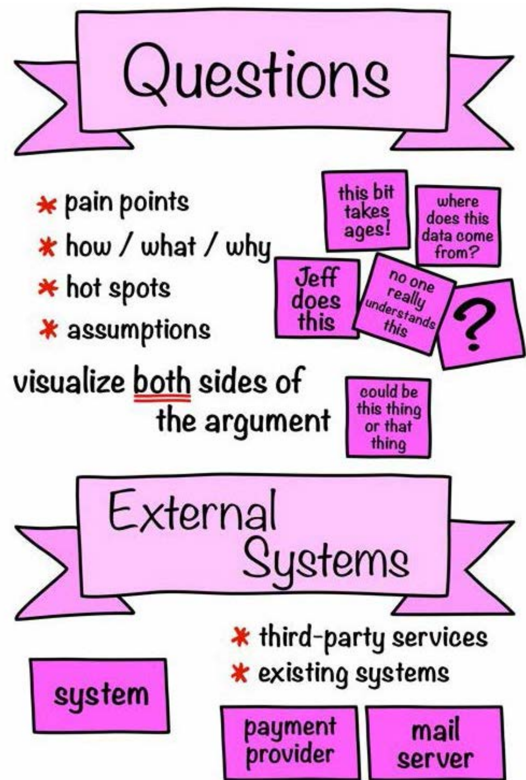


Figure 9.9: The Questions and External Systems key

This is followed by **Commands** and **Actors**.

Figure 9.10: The Commands key

One more part (for now) to add to the key is the **Read Model**.

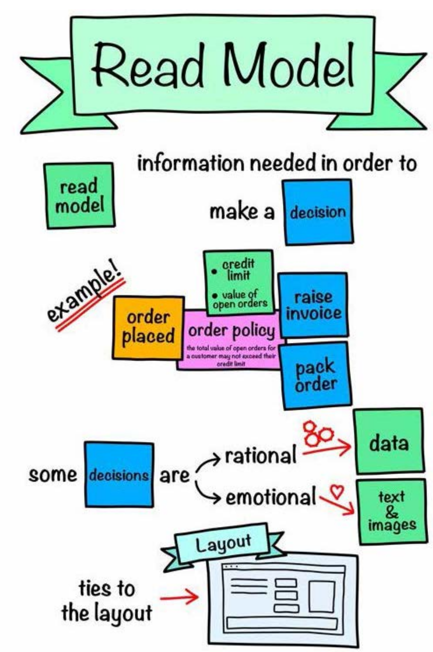


Figure 9.11: The Read Model key

The Event Storm posters should detail parts of the key with examples. Teams should now prepare to add the new parts of the flow again! Break into groups but rotate the members. If there are SMEs in given areas, again, make sure they're distributed throughout the teams that are formed.

Ask the groups to come up with the actors, commands, and systems that the command is issued to. Timebox this to 15 minutes again for the first pass. If the volume of events is quite large and the group is too, it can be useful for the teams to take the events between the two Pivot Points and flesh them out. This can speed things up, and all will be replayed together afterward so the shared understanding can still be achieved.

We try to steer groups away from being bogged down on whether the system is internal/external or how it should be named. It's good to keep things fuzzy until the last point where we really need to know the low-level details. For us, the primary thing with the system at this stage is just to identify that there is a thing to which the command is issued, thus triggering the corresponding event. In our experience, being fuzzy at this stage prevents people from falling down a rabbit hole, and keeps the discussions at a high level so everyone feels included.

Have the teams replay their additions to the Event Storm to the rest of the group. By doing this, more discussion will be generated and there will be more gaps identified. Capture assumptions and add the missing pieces as part of this replay.

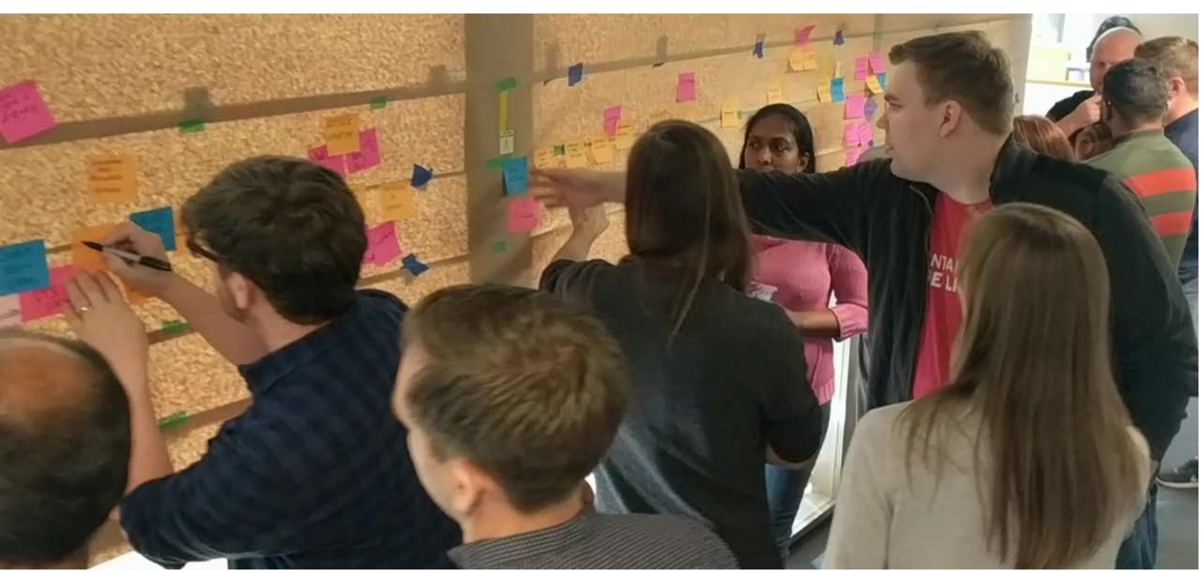


Figure 9.12: Three groups storming around the pivot events

If the teams have divided around pivot events and more time is desired by the groups to flesh out the flow, we will often rotate the groups to a new section between different pivot events, set a new time box, and go again. This helps to validate other groups' work and also flesh out the flow further with fresh eyes, revealing more knowledge.

During the next group replay of the full Event Storm, we introduce the next part of the key. The Policies and Procedures are introduced along with the Subprocess.

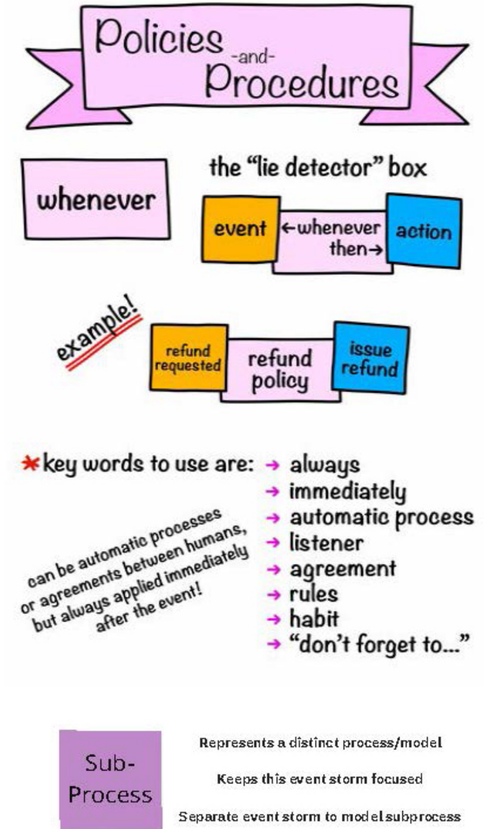


Figure 9.13: The Policies and Procedures Key

The policies and procedures are a great way to flesh out some of the gaps between commands and events. While the group is replaying the story, ask questions such as *Does this always happen*? or say things such as *Whenever a command is issued, we always do this event*. Doing so will tease out small things between the events that have not yet been thought of. We think this is why Brandolini refers to this card as the **Lie Detector**.

The subprocess can be a great way of parking content that will not be stormed during the scope of the workshop, but will be returned to later, usually in a separate event storming session.

A customer we worked with had a process for third-party fulfillment that was out of scope for the process they were trying to uncover. The purple subprocess sticky note was used to denote this unexplored area and was returned to in later workshops. A simple piece of tape was used to connect the two processes once they had been nested beneath each other so the visual connection between the two flows was not lost. A top tip is not to use a marker or pen as the flow may be (re)moved at a later date!

Figure 9.14: Storming around some of the parked subprocesses

Branches in the flow will inevitably occur. We mark these in a visual way using happy or sad stickers to denote the positive and negative sides of the flow. We tend to model the happy path first. Usually, for every happy flow, there is a corresponding sad flow. The subprocess in my eyes is a great way to capture these without getting bogged down in branches of the flow that we are not trying to explore in the scope of the workshop or wish to return to in more detail later.

If an event results in lots of commands being made available, we stack them vertically so the flow can be fleshed out one at a time. If time is tight, it can be good to just pick one or two flows to flesh them out and return to the others later. It's important when branching occurs to focus on the goal of the Event Storm and not getting bogged down in the weeds with low-quality information.

With most of the puzzle pieces in place, by now the Event Storm should start to tell a more detailed story. Continue to replay the story forward and backward, starting in the middle and going in all directions. Challenge all the items that are there, and don't be afraid to add or tear up misconceptions. You should start to notice the volume of question sticky notes drop off as knowledge is added to the Event Storm.

If the system you're modeling has UI components to it, a great add-on is to include some high-level UIs to the read model. Simple sketches on sticky notes can quickly validate what components the UI might need, as well as validating that the data that's needed for them can be retrieved. If you invited any end users of the application to the Event Storm, it can be really powerful to replay the Event Storm with any UI designs with them to further validate the hypothesis of the flow.

Figure 9.15: The group adding UI to the Event Storm on green sticky notes

Finally, it's time to identify and name the aggregate. The Aggregate is the state machine of the system.

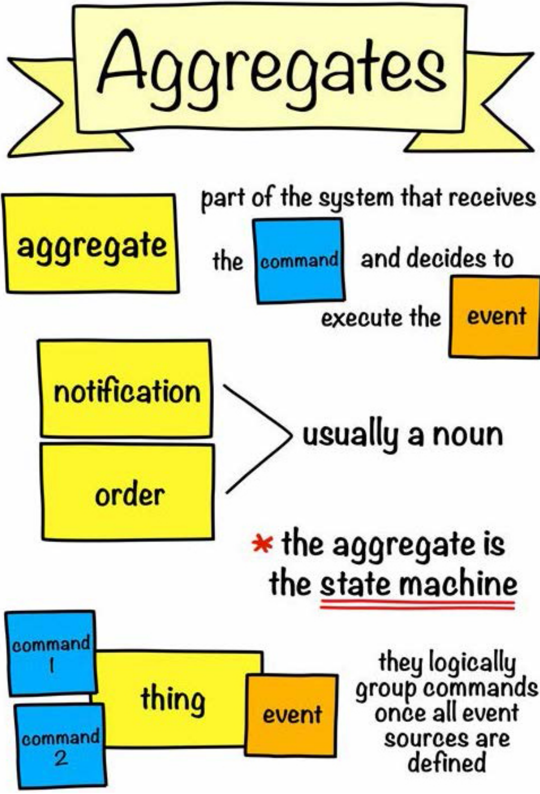


Figure 9.16: The Aggregates key

It's the thing that receives a command and decides to act upon it or not. We only introduce the system early in the flow as the aggregate can be the bit that leads to the most confusion and be the biggest time sponge – especially when trying to name it!

By holding off until the end, a large portion of the unknowns is cleared up. When the flow is more fleshed out, we get the group to replay the Event Storm and identify where a system is either existing or has to be built. If it has to be built, we add the yellow aggregate card and give it a nonsense name such as combobulator. By doing so, it helps people not get bogged down in the boundary of where the aggregate should be defined.

Continue to replay the flow and the group will naturally start to identify that some commands are issued to the combobulator and others to something else. Capture these as something else. When a full pass of the Event Storm has been completed, then return to naming it. The depth of understanding should be so high that the name and boundary of the aggregate should naturally fall out.

Event Storming with Doubters

It's no secret – event storming is my favorite practice. I think it is an immensely versatile practice that can be used at various levels of detail to speak a coherent language with people from all walks of a company. You can play it back to end users to validate a user journey while also keeping the techies involved by identifying the system design required to support it.

When I worked for a Swedish car manufacturer, we used event storming to design a solution for our app. One of their Java engineers working with us was extremely skeptical about this and our process initially. He admitted to me that he saw us coming in and thought *Oh, here comes the Post-it brigade*. But once we got into it, he rapidly became convinced, he could see the solution unfolding in front of us as the team of designers, business analysts, developers, and users all stormed around the modeling surface together.

After a couple of days at it (yes, really days!) he came up to me and said how shocked he was that in such a short amount of time we'd uncovered so much domain knowledge and had revealed enough of the solution for us to start playing with the technology and building it out. He was convinced!

Event storming is so versatile – in that same engagement, we were building our Event Storm around how a person would author new content within their application. It was a knowledge-based application so content manipulation was a key component – think of it as a Wikipedia for cars. While modeling it, we came to one section of the flow where in the journey, a user would edit some text and save their progress. This set off some alarm bells for me as the product owner kept saying the user would save their article and then publish it. Having a save button that someone would need to press in order to progress seemed odd. In my view, the app should have just been saving edits transparently to the user. Suddenly to me it felt like we were designing Microsoft Word 97 as opposed to a modern web app in the 2020s.

So, I challenged the product owner and said, "Should we not have the app autosave for us?" He was not in agreement and thought the users would want to save their progress manually. I figured, let's use the Event Storm to run an experiment! We captured both sides of our arguments for and against a manual save and put them on a pink Post-it to represent an assumption we're making. I tend to put a dot sticker on Post-its that require further exploration or experimentation. This makes it easier to view when looking at the full model. We put a placeholder event on the model to say *progress saved* until we'd done some further research.

Figure 9.17: Using an Event Storm to discuss the autosave feature

This assumption was then written into our backlog, where our designers were able to run an experiment with a clickable prototype to see what the users wanted or needed. In the clickable prototype, there was no save button and when asked after the tests if they felt it was missing, all of the respondents said they assumed their progress was being saved as they updated the content.

Returning to the Event Storm, now we had validated an assumption we've made. This meant we could proceed to model out our understanding of the user flow at this point. It also meant we could extend the things we'd need to build from a technical standpoint with this understanding.

I always think an Event Storm is the best guess at a point in time with the knowledge of the people we have in the room. Setting up our environment with a social contract to ensure psychological safety means we can freely explore our system. We are able to make guesses and return to them or put placeholders on the map until we do some further research. This is why event storming, especially when paired with all the other practices, is such a great fit for discovering and designing a system.

Let's dive back into the PetBattle story and look at how their team built up their Event Storm.

PetBattle Event Storm

The PetBattle team has decided to use event storming to design part of their system. As with all great teams, they started by defining the example they would map out. This is important as it frames the end-to-end journey and stops them from modeling too big a piece of the application.

!TOP-TIP => The team is using Friends Notation for their examples. This is based on the US sitcom Friends,[3] where every episode title starts with the words The One Where. They have taken some of the deliverables from the Impact Map to drill down into the system design and have come up with *The One Where Mary enters the daily Tournament and wins a prize.*

3 https://en.wikipedia.org/wiki/Friends

They add this to their Event Storm in the top-left corner and add Time before coming up with the spine of the events.

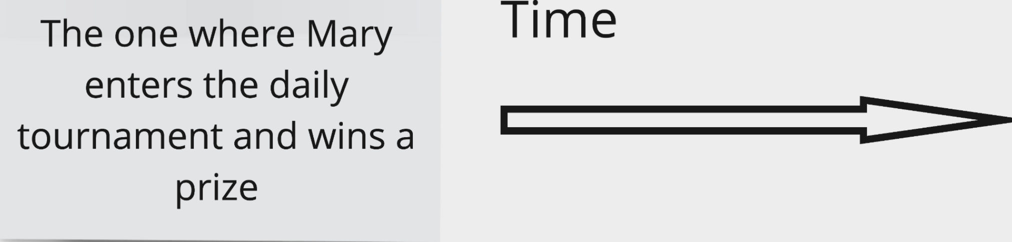


Figure 9.18: The One Where Mary enters the daily Tournament and wins a prize

Individually, the team members came up with all the events within the boundaries of the example defined. Starting in the center of the map, one team member put their events on the map, describing the story to the team.

Other team members filled in the gaps to include their events, thus forming the overarching spine of the story. Here are the first stickies from the PetBattle Event Storm. There may be other events that sit between these ones, but the team is happy with the depth they have gone to for the moment.

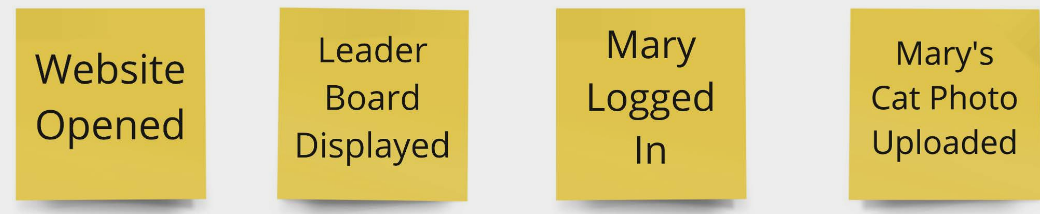


Figure 9.19: First events added to the board

When two team members had similar events written differently, both were initially put on the board and discussed further before agreeing on one phrasing for the event.

In the example, *Cat Photo Reviewed* versus *Cat Photo Validated* was up for discussion. Ironing out the language of the events is important to bring the team to use a consistent vernacular.

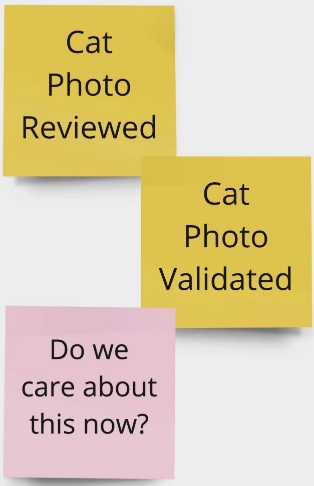


Figure 9.20: Capturing the first unknown / question

If they cannot agree, or need to make an assumption on the flow, we use a pink sticky note to highlight the discussion and capture it on the modeling surface. For example, the team decided they didn't need to go into the detail of describing how image validation would work at this stage, but they may return to it later.

The team continued to explore the event spine. They filled it out by reading the story from left to right and again from right to left, adding new events as required. Once they were satisfied with the event spine, they moved on to the commands and the data.

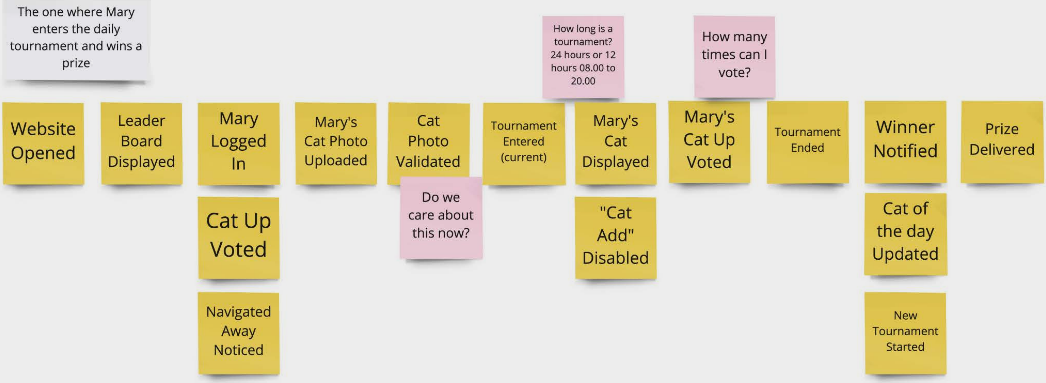


Figure 9.21: The PetBattle event spine

The team now introduced three more pieces to the canvas:

- **The actor** – the who of the system
- **The command** – the action or decision taken by the actor
- **The read model** – the data needed in order to be able to take an action

We can start to tell the narrative of our users' journey. In this example, our actor, Mary, *Opens PetBattle*. Whenever this occurs, we display the current leaders including information such as the number of votes each cat has, an image of the cat, and the owner's name.

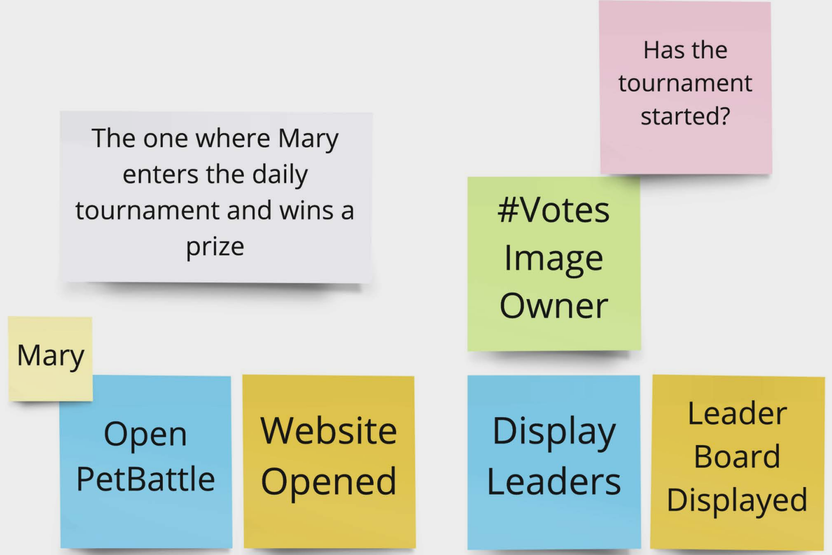


Figure 9.22: Narrating Mary's PetBattle journey through Event Storm

While looking at the leaderboard, Mary has a number of options available to her. In an Event Storm, things happen sequentially over time. Some commands may be carried out in parallel or at any given moment in the flow. A good convention is to add commands to the map as soon as they become available to an actor.

When Mary lands on the leaderboard page, there are many journeys that she could take. She could log in to play the game, she could vote for a cat in the currently running game, or she could simply navigate away from the site. Our convention is to stack these commands as each could represent another flow that needs to be mapped out. It's important for the team to not drift too far away from the example they are mapping out, so they are captured and added to the board but not further explored.

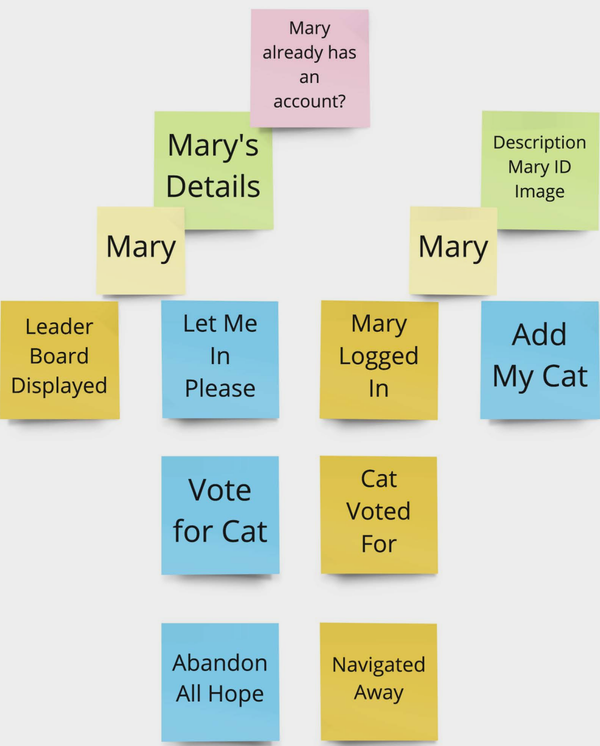


Figure 9.23: Stacking commands

The read model may be more than data. It could also include a low-fidelity UI mockup. Small UI experiments provide a great opportunity to generate user feedback. This provides us with product direction without writing a single line of code.

For example, once the *Leader Board Displayed* event occurs, we could sketch out what it looks like. This gives our actor an overview of what options are available to them and how they could interact with the system.

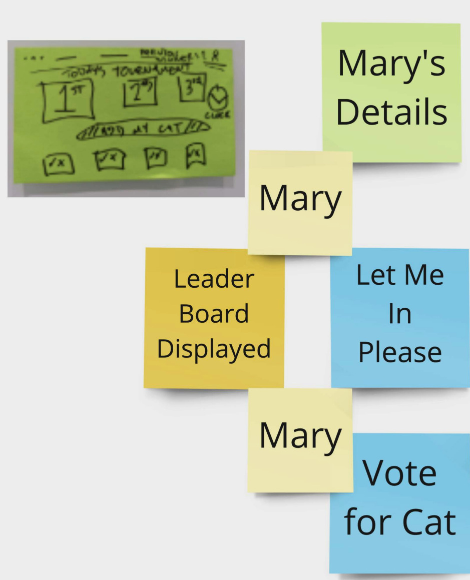


Figure 9.24: Sketching out UI ideas

The PetBattle team's Event Storm is starting to take shape with the addition of commands, read models, and actors.

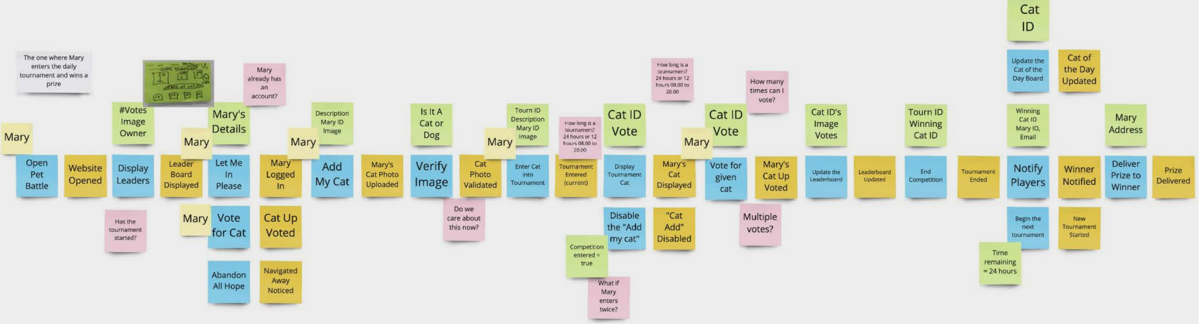


Figure 9.25: Full Event Storm taking shape

The PetBattle team now add any external systems they're integrating with along with policies/procedures and aggregates.

The purple Policy sticky note is a great way to add rules or habits to the Event Storm tying commands to events that were not initiated by an actor.

Figure 9.26: Adding the first Policy

In our example, whenever we open the application, we immediately display the leaderboard. The actor does not ask for them to be shown so we can connect the *Website Opened* event to the *Display Leaders* command by a *Display Leaderboard* procedure.

A command must always be issued to a system or an aggregate of systems. This system must decide to accept or reject the command and propagate the corresponding event(s). This results in happy path events (everything goes OK) or sad path events (some error). Generally speaking, there is a lot more value in happy path events so we Event Storm the happy path. There is still value in capturing and questioning sad path events, for example, the *Cat Photo Invalid* event – *do we care about this now*?

In our example, the *Display Leaders* command is issued to the *Tournament Service*. The Tournament Service is a new entity (an aggregate) that is responsible for holding the state of who's winning the current (and possibly the previous) tournaments.

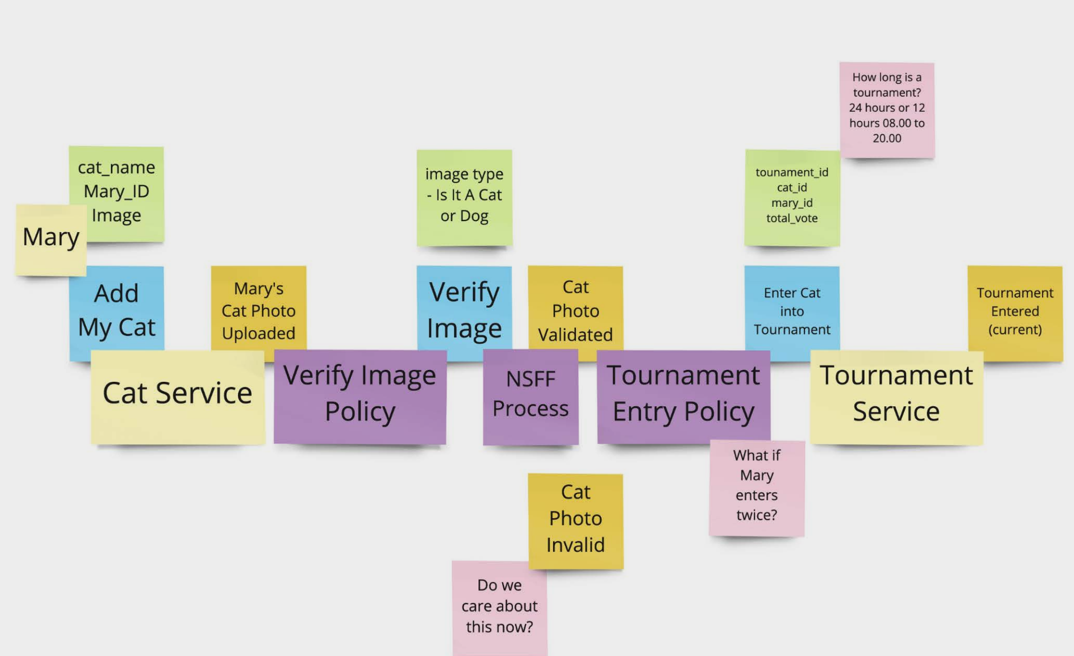


Figure 9.27: Processes, policies, and services added to the Event Storm

The PetBattle team has continued along the flow and event stormed how Mary can enter the competition. The Event Storm can now be read much like a story, capturing key things the team could explore now or later. Reading from left to right, Mary uses her image and ID to add her cat to the competition, then this command is issued to a cat service that is responsible for holding the cats' properties, such as image and owner.

Whenever the photo is uploaded, we execute the verify image policy. The team knows they're storming a happy path of image upload so they make an assumption that the image is a cat. In order to not think about how image verification could work, they put a purple diamond sticky note to return to later. The **Not Safe for Families** (**NSFF**) process is a process about ensuring images used on the internet are safe for children to see. The need for this can be traced back to our impact map's deliverables. The team essentially puts a pin in it here with the purple sticky note, knowing they can return to it later when other bets have been verified. There may be a lot of information required for the full implementation of the NSFF process. It may require further discovery steps. We have now gathered just enough information to carry on with our Event Storm.

Assuming the cat photo is valid, we immediately enter Mary's cat into the tournament through the *Tournament Entry Policy*. The *Tournament* service aggregates information from multiple sources into one new meaningful object. It takes *cat.id* that's just been uploaded, Mary's ID, and the number of votes Mary's cat received. The *Tournament* service processes where Mary's cat sits in the leaderboard along with the other competition entries. When the *Animal Lovers* play the game now, they will amend the total count for each cat in the current competition.

In an early version of PetBattle, cats were being randomly fetched from the internet when there were not enough players uploading their own cats. In an Event Storm, this external system is captured on a pink Post-it. The following example shows how we could model this interaction. We fetch the tournament cats from the *Tournament* service, which results in no cats being found, triggering the *Empty Competition* policy. Whenever this occurs, we always fetch random cats from the internet and add them to the game. Capturing third-party systems in an Event Storm can highlight dependencies that exist within our application.

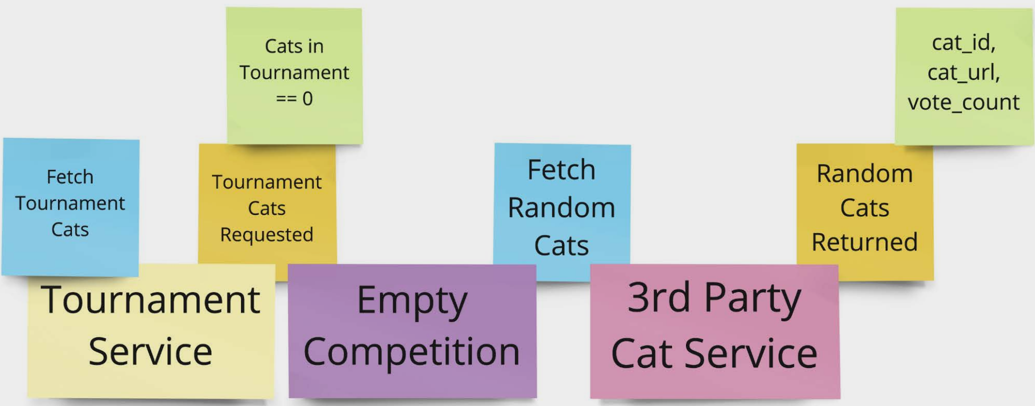


Figure 9.28: Using a pink Post-it to capture external elements

The cross-functional team working on PetBattle's Event Storm includes representatives of their UX team. The *User Interface* card provides us with the ability to create high-fidelity prototypes. If the group cannot agree on something or have to make an assumption, then validation with some end users may be required.

In the case of PetBattle, we had some heated debates around the design of the leaderboard. We couldn't agree on how many cats to show, that is, the top 10 or just the top three. We also couldn't agree on whether we should always show Mary's uploaded cat, even if it was not attracting any votes to be in the top.

We decided to capture this discussion on a pink Post-it. This Post-it represents work that needs further exploration and possibly some user research to find out what functionality the players of the game would like to see here. As with all work, this can be captured and added to our backlog. The team could also build out multiple versions of the application, showing the top three cats and the top ten. Using OpenShift, we could deploy both versions of the app and gather some feedback on the user engagement metrics to help determine which version is preferred by the players. In the next section of this book, we will explore some of the options around deployment strategy, such as A/B testing, which will enable us to learn about the optimal solutions based on behavior and impact.

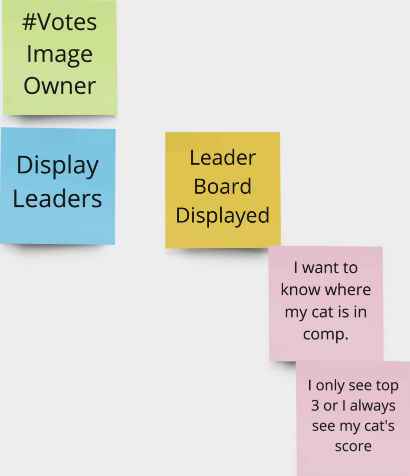


Figure 9.29: Using a pink Post-it for a feature that is yet to be finalized

At the end of the Event Storm exercise, the flow represents a best guess made by the team knowing only what they currently know. It is not a static document but one that is revisited as they build some of the features identified on the map. We now have just enough information to be able to design how the components of the system could interact.

We'll see how all of this information is used by other practices on the Options Pivot section of the Mobius Loop in *Section 4*, *Prioritize It*.

Final Thoughts on Event Storming

Event storming is a brilliant technique. The domain knowledge it extracts and the time taken to accomplish it is powerful. It can be used at a business level to identify pain points and areas to improve, much like a Value Stream map. Lower-level event storming is a great, immersive, collaborative process for designing software. We would highly recommend you try it; it's a tough one to get right but practice makes perfect.

In the next section, we are going to talk about our approach to discovering the technical architecture. Now that we have completed the first run-through of our Event Storm, we should be getting ready to run some experiments and write some code!

Emerging Architecture

At this point in time, a traditional architecture approach would be to write a high-level or detailed design document. This would normally take months and is an activity that would be carried out by enterprise or solution architects. It usually involves a high dosage of UML (Universal Modelling Language) diagramming and many pages of technical discussion and validation. Let's look at a different approach that better aligns with our discovery practices so far.

The idea behind emerging architecture is that we get *just enough* knowledge and shared understanding of a business domain with *just enough* understanding of the logical architecture needed to deliver high-business value features. The major architectural approaches can be decided at this stage, such as decoupled services, event-driven architecture, APIs, and streaming applications – and subsequently, this will be refined in the following iterations.

In our Event Storm, we drilled down on one or two of the TOWs (The One Wheres). We want to be confident that the aggregates, commands, and read models would deliver just the capability described in that part of the business domain – no more and no less.

By taking an emerging architecture approach, we do not invest time in building a comprehensive architecture that is not yet going to add business value. It may be that the future direction of the product or re-prioritization of features means that this architecture is actually never used and therefore time is not wasted.

The team will gather an increased level of shared understanding and confidence in the overall solution as they incrementally deliver it. This means that any future solutioning conversations benefit from that additional context and understanding.

After our first pass of the Event Storm, our initial draft of the architecture is depicted here.

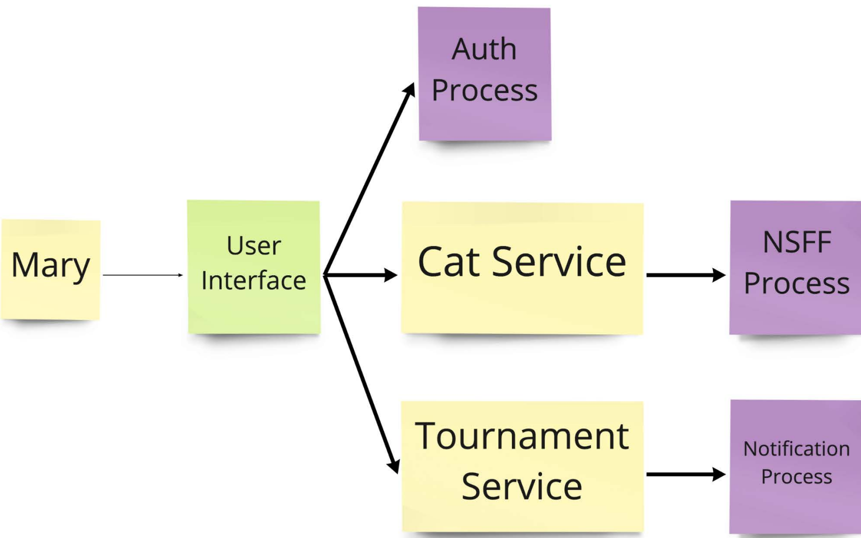


Figure 9.30: First pass of the PetBattle Event Storm

We haven't yet fleshed out the details of the AUTH (Authentication), NSFF, and Notification processes. These would usually be expanded in subsequent iterations of the Event Storm practice. However, given the linearity of books, it was decided to add these here to give you an indication of what a future architecture could potentially look like.

Transitioning an Event Storm to an Emergent Architecture

The Event Storm should lend itself nicely to an emergent architecture. The Event Storm identifies the systems and aggregates that need to be built. This can give you a good idea of where you need to take the design of your software but doesn't mean it all needs to be built upfront. You may learn from a user experiment that some functionality is not needed or that your initial thoughts on the Event Storm were not refined enough. This might sound like bad design but in fact it's just better-qualified design that's evolving through experimentation.

We ran an engagement with the **World Health Organization** (**WHO**)[4] that was all about trying to prove out their architecture and building a strong cross-functional team around their new learning experience platform. This is the platform that health professionals around the world would use to consume learning content from the WHO.

The initial Event Storm with the team threw up some interesting services that were needed. The course service, which was a type of catalog, as well as a highly available and scalable authentication provider were identified as key components. As the reader, you may not be able to see all the detail on the figure below and it is not necessary - it's simply included to show the amount of information already collected.

4 https://www.redhat.com/en/success-stories/world-health-organization

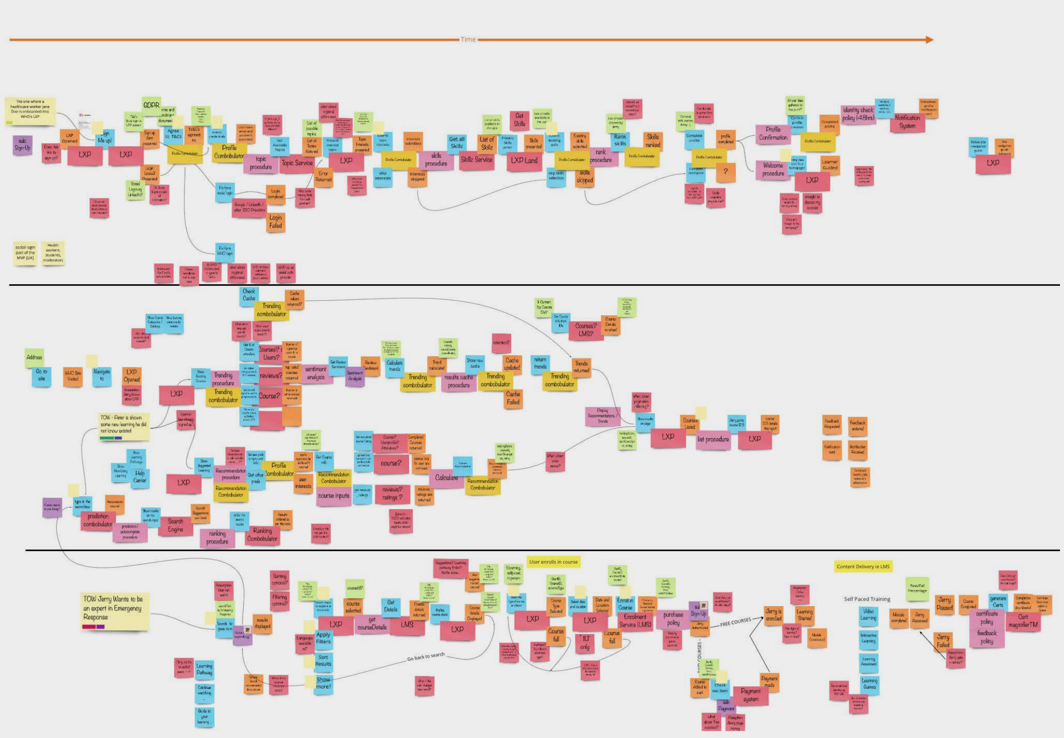


Figure 9.31: Event storm of our WHO project

Our Event Storm was not overly complex but it looked like this after many weeks and informed revisions. The details of each item are not super important; suffice to say the long pink and yellow stickies represented unique APIs we needed to build from a technical point of view. Ultimately, we discovered through our Event Storm a large number of components, not all of which were validated with end users:

Figure 9.32: Components discovered during WHO Event Storm

We knew we had lots to do but we also knew we didn't need to do it all at once. From our Event Storm, we iteratively built functions and features, adding to the architecture as required. This meant we could be focused and lean, keeping one eye on the big picture without over-complicating our goal.

Sprint 1

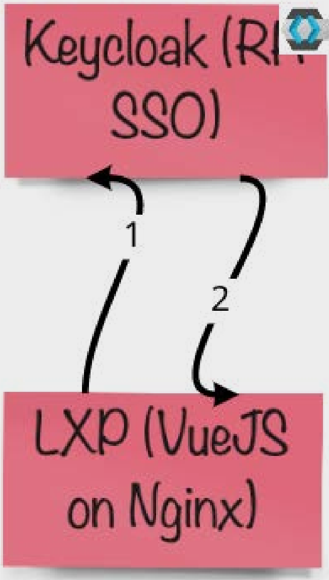


Figure 9.33: Sprint 1 emerging architecture

We evolved the solution design at every sprint. We revalidated the design against the Event Storm and only added new components when they were needed. Our first sprint introduced the basic elements: a simple frontend application talking to an instance of the Red Hat Single Sign-On server (the upstream project is called Keycloak).

The second sprint saw us add our first microservices, discovered as systems or aggregates on the Event Storm. In subsequent sprints we added more functionality, introduced new components, and hardened previous ones. We added caching to previously built services to speed up lookup performance. We were able to build incrementally, then add complexity such as a cache after laying the groundwork.

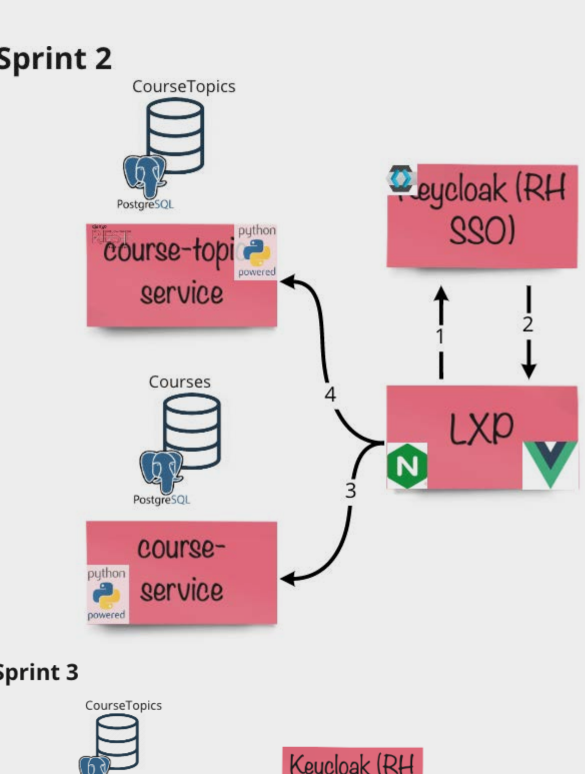


Figure 9.34: Sprint 2 and 3 emerging architecture

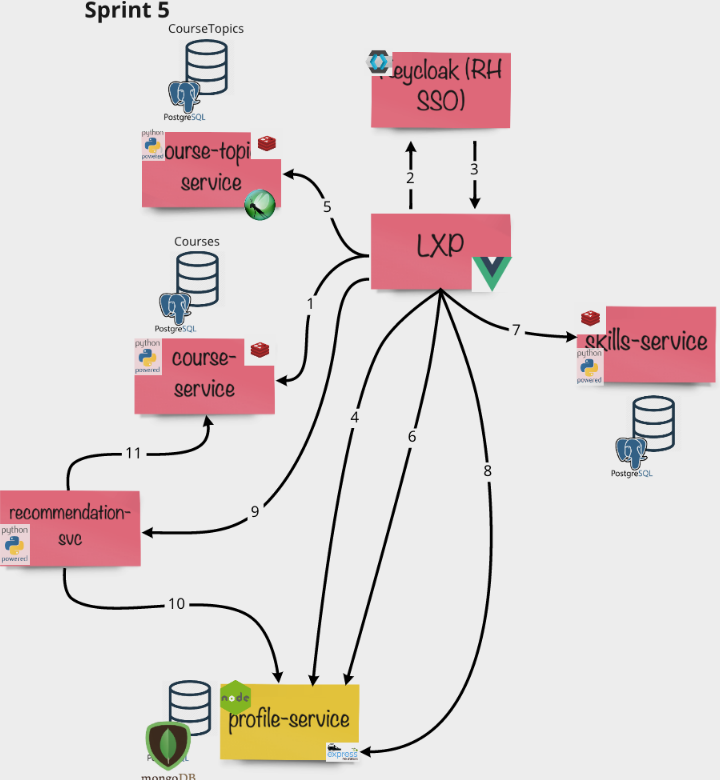


Figure 9.35: Sprint 5 emerging architecture

We evolved the architecture of our system by continuously going back to the Event Storm and modeling more The One Where journeys. Each time we did this, new services were discovered.

In one example, the WHO wanted to create a rich recommendation engine that would help health care workers find relevant health information courses faster. We needed to gather preference data to be able to give a recommendation. So, we created an onboarding process to help gather this information.

We were constantly connecting with our end users as we added new functionality, providing a feedback loop that helped us validate our initial design assumptions. By the final sprint, the complexity of our system had grown but each item on our architecture could be traced back to its purpose for a user on the Event Storm.

In the previous real-world example, we made use of a cache to speed up API calls. Next, we will look at a practice that will help surface these non-functional aspects of our solutions.

The Non-Functional Map

Non-functional attributes help define the usability and effectiveness of the entire solution. The non-functional map is a tool to capture and articulate these non-functional requirements for the system that a team is striving to deliver. Most of the practices so far have focused on the functional aspects of a solution that relate to business features or functions that end users can directly relate to.

Non-functional requirements are broken down into areas represented by each segment on a canvas, which are examined and populated in turn as a group.

Figure 9.36: The non-functional wheel

We can use this mapping tool to gain a shared understanding of non-functional aspects of a system in a visually structured representation. On completion, you should have a good view of the key non-functional considerations for the solution. Mapping these visually can highlight where work is required to realize them.

It's a very simple practice to run and could easily be done in an hour with the right people in the room:

1. Ensure that everyone participating is comfortable with the headings and what they mean.
2. For each heading, capture important considerations or requirements for the system or solution and add stickies for each item. Depending on the number of participants, you could use facilitation techniques such as 1-2-4-all or splitting into smaller groups to capture items.
3. For each heading, play back the stickies as a group.
4. During the playback, think about whether each sticky should be:
5. An item added to the backlog and prioritized as it requires work to realize.
6. An item that should be included in the Definition of Done for other backlog items.
7. Part of the acceptance criteria for other backlog items.

From Non-Functional Map to Backlog

Discovery practices such as event storming or user story mapping are brilliant tools that facilitate conversation and generate work for a backlog. However, they are not great when trying to surface non-functional requirements for a solution as they tend to be more business process-oriented, driving functional features and end user aspects of the solution. We used this practice with a large team in a recent engagement. We split into three teams, each armed with our own Post-it color.

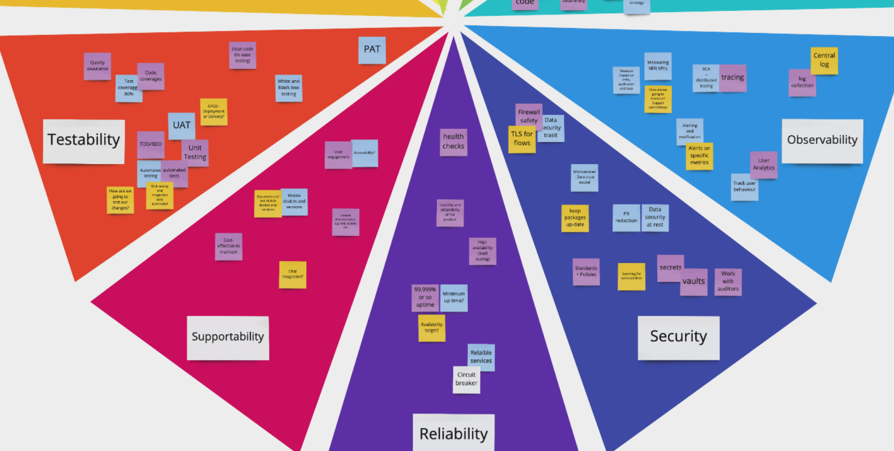


Figure 9.37: The non-functional requirements map

The activity provided a great opportunity for the techies to discuss non-functional aspects of our application. The tool proved to be a useful platform for inviting other stakeholders into the mix, those not directly involved in building the solution but who had a vested stake in the design, such as members of the security team. The team voted on some of the items they thought were the most important to focus on for further discussion.

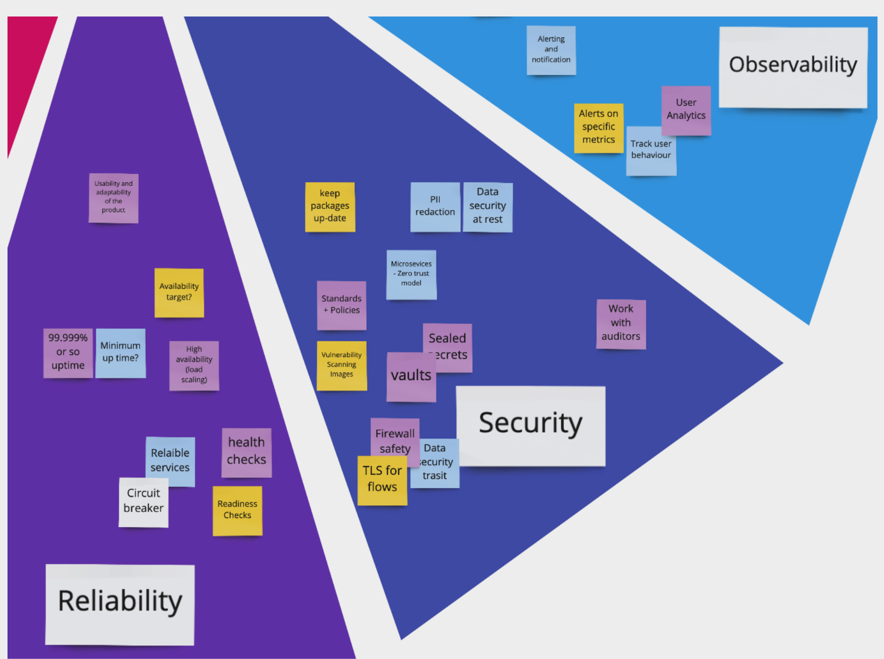


Figure 9.38: Focusing on some items of the non-functional requirements map

Zooming in on some of the sections of the wheel, such as Reliability, the team came up with ideas such as health checks and readiness checks. High availability and vulnerability scanning came up as topics to explore from a Security perspective. We discussed the items we thought were most important, then transcribed them into our backlog. Nearly all of the items on our map represented work that could be done by the team.

Some items went into our backlog as spikes (or unknowns) to warrant further exploration and others were easily written as user stories for the team to implement, such as the Kubernetes readiness and liveness probes.

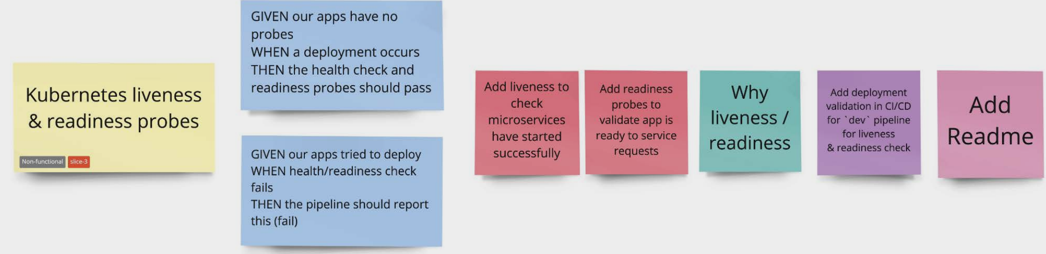


Figure 9.39: Sprint backlog items resulting from non-functional map

The non-functional wheel is a great practice to start a conversation about all the different non-functional aspects of the solution. One area it will not necessarily facilitate conversation in is the speed of delivery and the lead time to deliver code to production. This is a big focus for DevOps and therefore something we want to pay attention to during our Discovery loop. To shift focus toward continuous delivery, it can be helpful to capture some baseline metrics around delivery processes.

Discovering the Case for Continuous Delivery

Throughout this chapter, we have explored some of our favorite practices that help us with product discovery: discovery of the Why using impact mapping, discovery of the Who using human-centered design practices, and discovery of the What using event storming and emerging architecture. We started to explore some of the How's using the non-functional map. The final set of practices we'd like to introduce promotes how we are going to develop our software, and it does so, based on metrics.

One of the most common questions we get from our customers is how they can release their software to users faster and more frequently. Often, they have adopted agile techniques for delivery (for example, they may be using the Scrum framework) but still only seem to be able to do one or two releases to production every year. Why is that?

The first clue may be to heavy a focus on the delivery framework. As we explained in *Section 1*, *Practices Make Perfect*, when we introduced the Mobius loop, achieving successful DevOps culture and practice requires a balanced focus of continuous discovery and continuous delivery, and a strong foundation of culture and technical practices.

To really understand why it is taking so long to release software into production, we need to understand where the bottlenecks are. We need to visualize everything that happens from the moment a business stakeholder asks for a new feature all the way to the moment that that feature is running in production and being used.

In *Section 1*, *Practices Make Perfect*, we introduced the idea of a value chain. The value chain focused on exploring everything from generating the initial list of business requirements through to the point when a big release of software is running in production. DevOps is focused on speeding up that value chain. A practice that will help visualize all these steps and capture some metrics about how long they take is called **Metric-Based Process Mapping** (**MBPM**).

We introduced **Value Stream Mapping** (**vsm**) in *Section 1*, *Practices Make Perfect*, so you know it's a macro-level map of the end-to-end delivery flow of a particular service or product from its inception until it reaches a customer. MBPM, on the other hand, is a more detailed or micro-level view of how some of the stages or single processes of a vsm deliver value. These detailed-level views show flow and metrics that help provide insights into bottlenecks.

Metrics-Based Process Map

Let's start with a simple example of a software development flow. In the real world, the map would be more granular, giving you more detailed steps and roles, but for the sake of clarity, let's look at this simplified example:

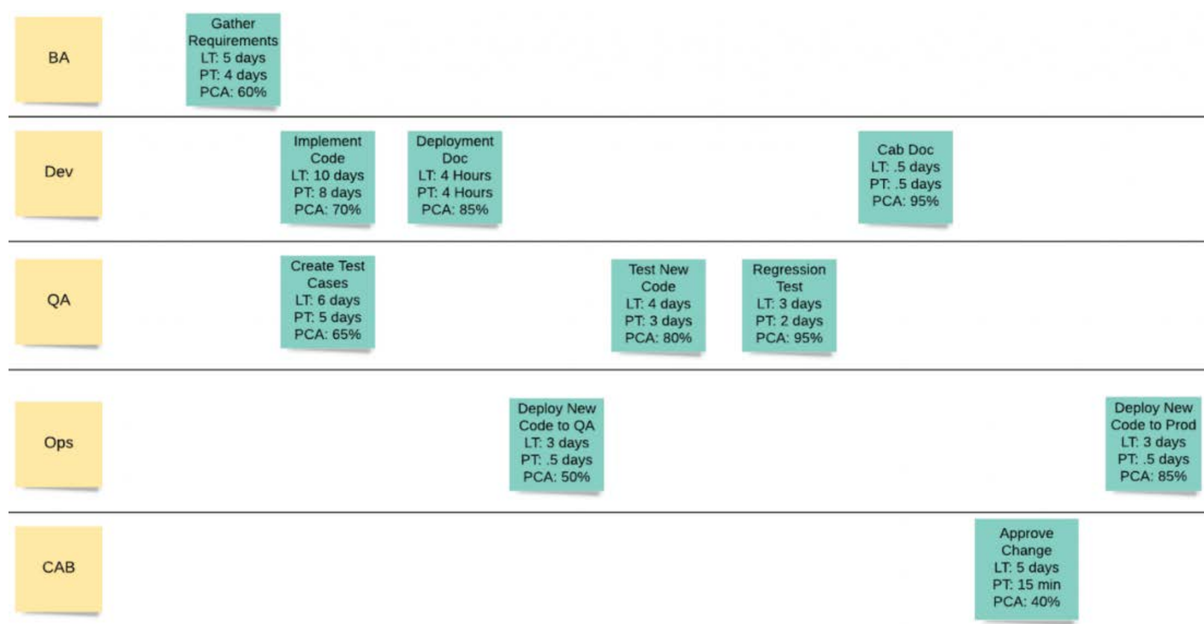


Figure 9.40: The MBPM

The map's layout should be read from left to right:

- Each actor or role has a horizontal swim lane. The roles are the leftmost column (yellow stickies in the example, BA, Dev, QA, and so on).
- There is a defined starting point and ending point that is identified before starting the exercise. In the example, the start is defined as Gather Requirements and the end with Deploy New Code to Prod.
- Each non-actor sticky note has a title that corresponds with an event.
- The actor role that owns the work for that event lines up horizontally with the actor sticky note.
- The events flow from left to right, with earlier events on the left and later events to the right.
- Events that happen in parallel are lined up vertically. If there is shared work between roles, those should also be separate stickies lined up vertically as well.

- Each card contains three metrics:
  - **Lead Time** (**LT**): The total amount of time within normal business hours from start to finish that it takes to complete the task. This includes any idle time or time waiting on other roles to complete work.
  - **Process Time** (**PT**): The total amount of time someone is actively engaged in working on the task. This is a subset of lead time.
  - **Percentage Complete and Accurate** (**PCA**): The percentage of time a task is completed accurately on the first attempt. This is a compounding percentage with earlier dependent steps.
  - Any useful information about the execution of the process, such as whether it is batched or scheduled (and, if so, at what time) and how many people are involved in executing the task.

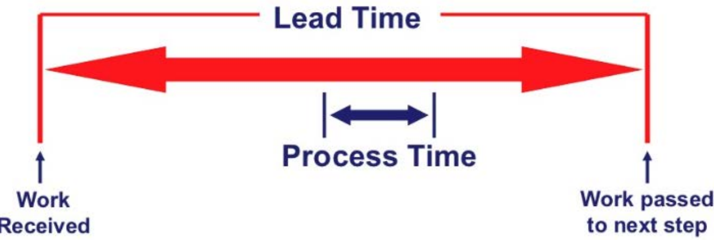


Figure 9.41: Lead Time versus Process Time

In this example, we have a software delivery flow that looks like a traditional waterfall pattern of gathering requirements, developing, testing, and deploying code. Typically, MBPM would be more granular; for example, instead of saying the developer *implements code*, you could break that down to:

- Writing unit tests
- Writing implementation code
- Building code
- Deploying code
- The developer's integration test
- Peer review
- Fixing identified issues from peer review
- Documentation

Finding and Making Improvements

Once you have completed your map, you now have a communication tool for the team and stakeholders to absorb. The map also helps management see the pain points. These costs have a time value and that time value has a cost as these issues continually occur. If you have more forward-thinking management, then they may be thinking about lost opportunity costs as well. You may have faster-moving competitors generally delivering value faster, possibly cheaper, and this is eating into your market share.

To make improvements, look for the following things:

- **Lead time is much higher than process time**: This may indicate things sitting in queues or waiting on someone else; this typically happens at handoffs. Is there any way to reduce this waiting time?
- **Low percentage complete and accurate**: Why is it so low? If the answer is not obvious, this is a great time for a blameless retro with the team. I bet someone has some insights as to why it is happening.
- **Unnecessary steps**: Can you deliver value without a step or a scaled-back step?
- **Things that could be automated**: Reducing manual steps can increase speed and accuracy.
- Could changing the order of operations help? Maybe completing one step sooner or in parallel would improve accuracy.

You may see many problems, and it may be tempting to try to tackle every obvious problem at once. A better strategy is to start with the biggest and easiest wins first and prioritize a list of ideas from there. Remember to be agile about the improvements themselves: focus on small chunks of work to ensure faster delivery of work. Get a fast feedback loop and re-prioritize based on feedback.

Once you make a change, you need to adjust your map accordingly. You should pay close attention to the metrics: are the metrics improving given the volume of work flowing through?

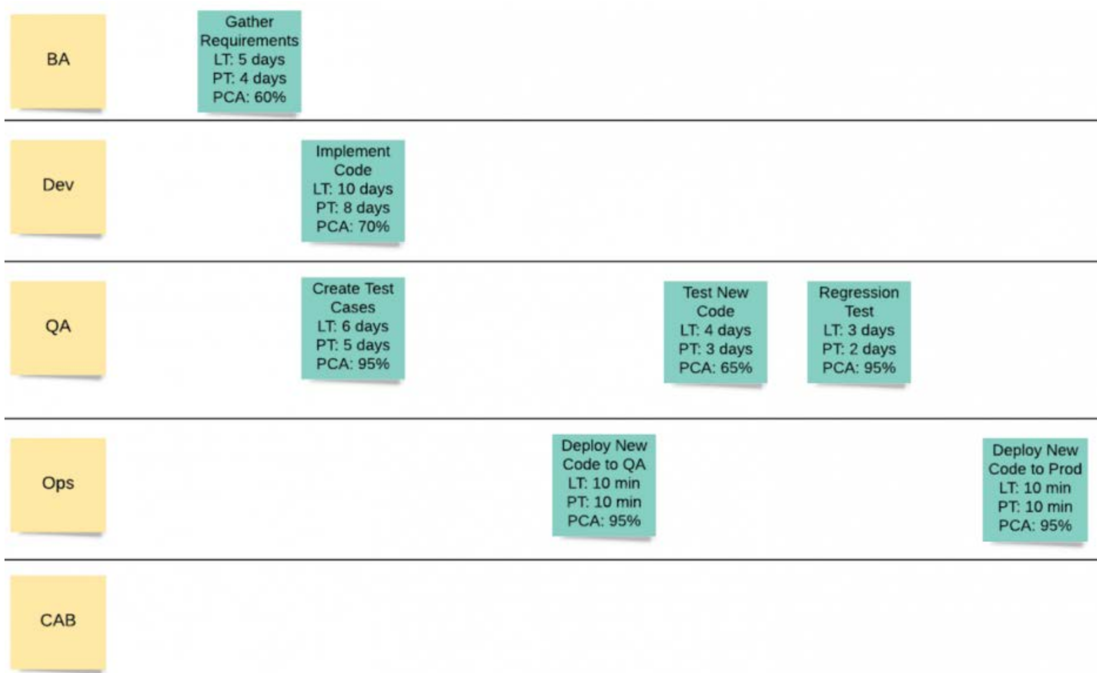


Figure 9.42: The MBPM

Improving through Iteration

A tool such as this should not be a one-and-done type of improvement. It needs to be used as an iterative endeavor where you make constant improvements. Your map should be wall art (if you have a physically co-located team) that is living and changing as you improve. It's there to remind and inspire future improvements. It's also great for onboarding new team members as you can show them how the process works visually.

In our example, perhaps there is an additional improvement that could be added around testing, so maybe the next step might be some sort of test automation.

As you tackle the obvious issues and implement more modern industry practices, you may find yourself getting to a point where the improvements are not as obvious or even questionable. This is where metrics can help inform you if your changes make you faster. However, in order to figure this out, you need to experiment. Technically, when you were knocking out the obvious changes early on, you were experimenting, but given that these changes had such a high chance of succeeding, it may not have seemed like an experiment. As you move on to the less obvious changes, you may get it wrong and implement something slower based on your metrics. This is okay; you are learning (the top performers are always learning) – just stop and reflect. This is a good place for a retro. Maybe you can change the idea or perhaps you need to roll back that change and try something completely new.

I like to think about improvements through automation in layers. If you are in an IT department of a non-tech company, the department was likely founded to automate business processes in the industry you are serving. However, over time, the IT department itself needed to be automated to deliver faster business process automation. This is the second layer of improvement. Within this second layer, you could bring in a tool such as MBPM to help identify areas within the IT department that can be improved. You may end up with common industry solutions such as automated testing or CI/CD pipelines, for instance. You can take the automation even further; you may quickly realize you need automation around the area of metrics collection. Manual MBPM metrics collection can be time-consuming and repetitive and have human errors. In order to speed up the feedback loop and free people up for other work, you may want to automate the collection of metrics.

Of course, all this automation work comes at a cost but also can benefit from it, and that is why metrics are so important to help inform you if the benefits are worth the costs. Also, you should not wait until you have metric collection automated before you even start using MBPM. Start simple, understand the process and what information you want to collect first, then automate once you get your rhythm going.

Scoping an Entire Engagement Using MBPM

At Open Innovation Labs, we run time-boxed residency engagements with our customers. They last between 4 and 12 weeks.

When leading an engagement with a Finnish telecommunications company, their focus was on improving the lead time for feature development, adopting some self-healing and auto-scaling features of OpenShift, and learning more about Kubernetes and site-reliability engineering.

Unlike most of my other engagements I'd led, we were not going to perform any application development on this residency. No new features. No new business functionality. No new functional release. This was purely non-functional and focused on improving the operation of the service the team was responsible for.

This felt like a perfect situation to bring out the MBPM practice. It wasn't a practice I'd used previously on new greenfield product development engagements. But, given this was a legacy brownfield application and focused on improvement and optimization, this discovery practice would be helpful.

The team consisted of a representative from the application development team, a representative from the application operations team, and an infrastructure specialist. Over the course of a couple of days, these three people combined all their knowledge of development and operations processes to create an MBPM.

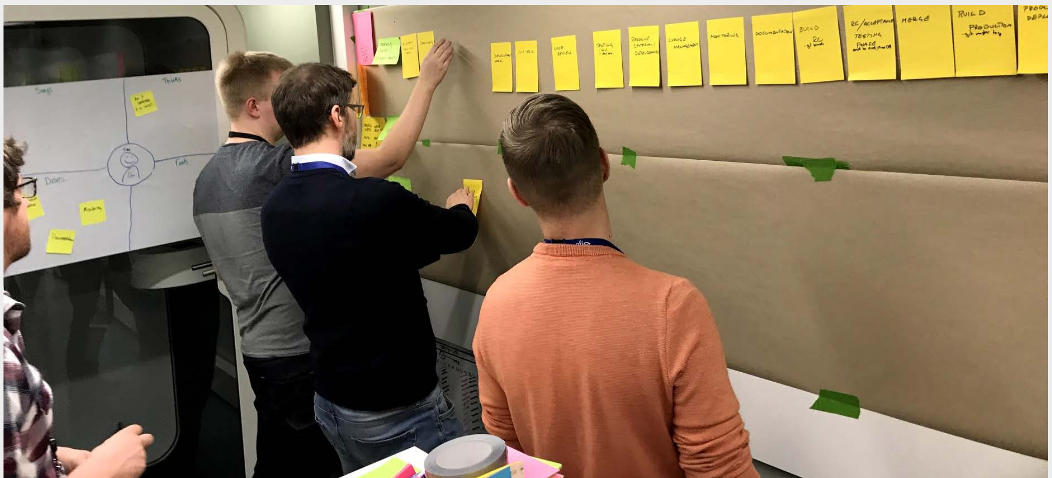


Figure 9.43: Cross functional team building their MBPM

Some of the key learnings that came out from this were:

- There were long lead times for any development to start because of the SLA for booking virtual machine time.
- There was a lot of wasted time waiting for builds and deployments to complete because there was a dependency on a third-party team to trigger these.
- There was low completeness and accuracy in some of the quality assurance due to the large amount of manual testing performed.

The yellow sticky notes in the following image show all the processes mapped into the MBPM. The pink sticky notes represented ideas for how we could optimize the metrics or even make the process obsolete altogether.

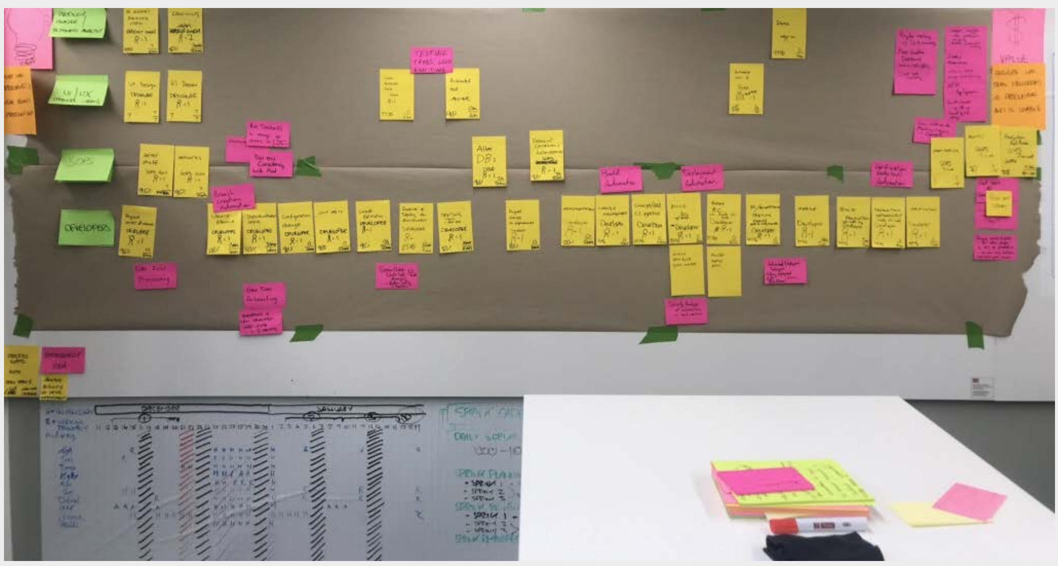


Figure 9.44: An MBPM created for a Finnish telecommunications company

When we returned to the MBPM three weeks later (after three one-week delivery iterations), we captured on blue sticky notes where there had been metrics-based improvements as a result of the continuous delivery infrastructure that had been introduced.

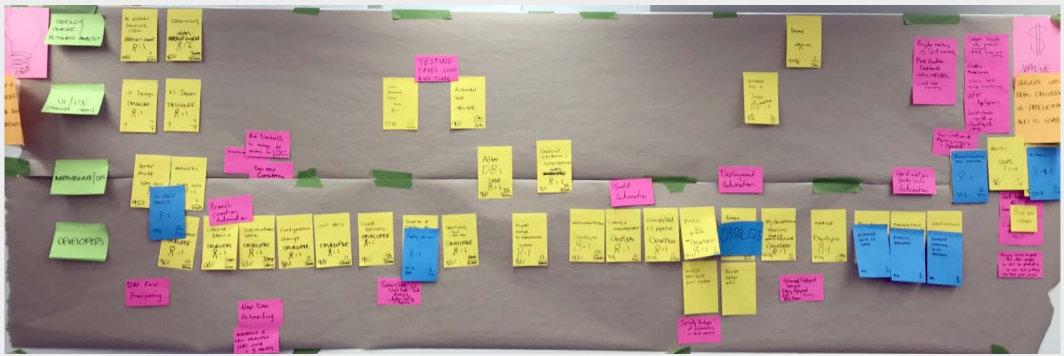


Figure 9.45: Updated MBPM created for a Finnish telecommunications company

This visualized and quantified the benefits of introducing containers, continuous integration, continuous delivery, infrastructure as code, and automation across the pipeline.

Let's see how PetBattle ran the MBPM exercise to capture some baseline metrics.

PetBattle – MBPM

The current deployment of the PetBattle application is torturous, taking many hours to manually copy and deploy artifacts to a Raspberry Pi cluster that also happens to be a media server collection that is installed under the table belonging to one of the developers.

This seemed concerning to the new engineering team and they suggested visualizing everything that happens in the existing software delivery process. This would mean a shared understanding of how things worked in the hobbyist app and allow them to ideate how they can optimize and improve as PetBattle starts to scale.

The team, led by Tim, created an MBPM on the wall. There was a good bit of discussion about who was involved in releasing PetBattle, the actors, and when certain events occurred. The team settled on the following flow as representative of what happened during a release cycle.

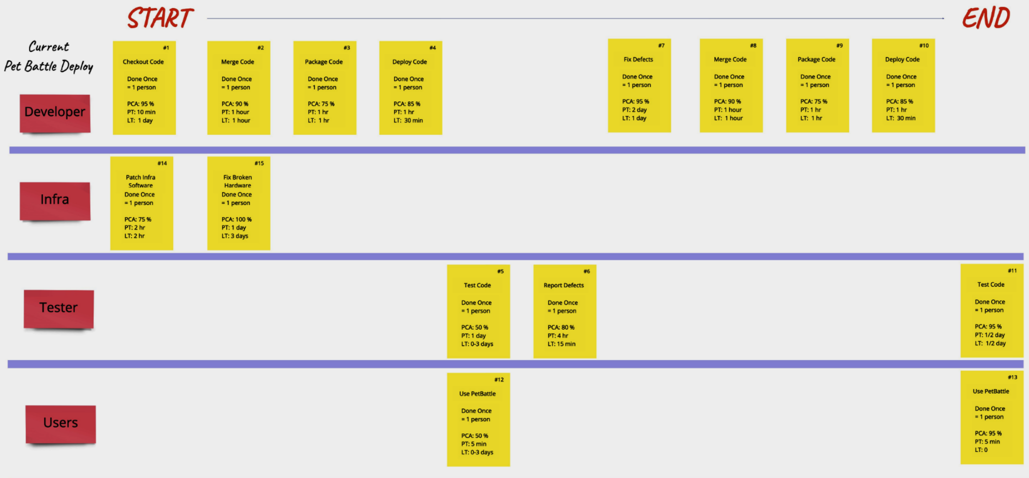


Figure 9.46: Starting phase of the PetBattle MBPM

One of the PetBattle developers (Mike) would begin the release cycle by checking out all of the code. The next step was to try and figure out whether all of the feature branches that needed merging had been merged correctly. There was some delay in finding all the right code branches. Packaging the code failed 25% of the time because updating the configuration values was always manual and error-prone. The deployment to the Raspberry Pi cluster could begin, which usually took 1 hour because it was a CPU-intensive task. There was only one cluster, so this took down the PetBattle service while deployment happened.

Normally, one of the other PetBattle engineers who understood the Raspberry Pi hardware the best (Noel) took time to patch and update the cluster operating system and tools. Often while doing this work, hardware issues were discovered that required new parts to be ordered. It could take up to three days to wait for the parts to arrive and often made the whole cluster unavailable, leading to a long wait time for testing a release.

Next, the best tester among the team (Donal) would run through the manual test plan. Because the release was already deployed, users would unknowingly use the new version, often with a lot of failures. Donal's' testing often failed about half the time usually because the test plan needed updating or there were just a lot of bugs! Donal diligently entered the bugs he found into a spreadsheet to share with the other developers. Often, he got the details mixed up, meaning that 20% of the reported defects had the wrong information recorded.

Figure 9.47: Ending phase of the PetBattle MBPM

The developers would take a couple of days to fix all of the defects reported in Donal's spreadsheet. The code was again merged, packaged, and deployed, often with similar failure rates to the first round of deployments.

Now, the end users could use the polished code. Often, end users were testing features at the same time as Donal. They would report defects and outages to the PetBattle email list.

From this practice, the team identified several initiatives that they agreed should be put in place:

- Use of containers in cloud infrastructure
- Automation of infrastructure
- Continuous integration
- Continuous delivery
- Test automation
- Security automation

MBPM is an excellent practice for providing baseline metrics, identifying bottlenecks in processes, and driving the case for improvement. It does require a psychologically safe environment as the metrics may not be particularly pleasing. Team members should not be afraid to radiate this information and should also feel safe in sharing information honestly.

We will return to MBPM in *Section 5 – Deliver It* when we measure and learn about the improvements we've made in the Delivery loop.

Conclusion

In this chapter, we continued our journey on the Discovery loop with a focus on how we are going to deliver solutions – or, at least, how we are going to start.

We looked at DDD and, in particular, the event storming practice, which helped visualize the business process flow. Our event-driven architectures started to emerge from the Event Storm. We started to further drive user research by making bets to form our backlog for feature development.

We also looked at a couple of practices that help form non-functional work we should consider – the non-functional map and the MBPM. The latter allows us to take key measurements associated with legacy software delivery processes and identify some of the continuous delivery practices we could employ to help improve and optimize them.

This chapter and the previous chapter have enabled lots of ideas and candidates for development/configuration work. They are all visualized on the artifacts produced by the practices.

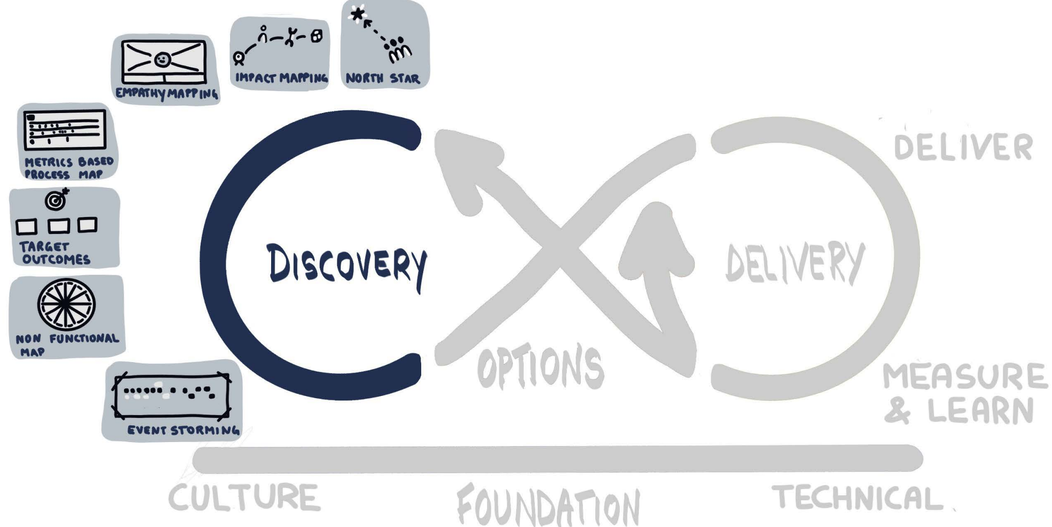


Figure 9.48: Discovery phase of the Mobius loop

In the next chapter, *Chapter 10, Setting Outcomes*, we're going to explore how we distill all this information into setting measurable outcomes. We'll then look at how we organize and prioritize all these ideas in a way where we can start to iteratively deliver value and do so regularly.

10

Setting Outcomes

In the previous chapter, we used practices to help extract lots of detail to confirm what problems we're going to solve, who we're solving them for, and our approach to solving them in an experimental and incremental way.

By using practices such as Impact Mapping and human-centered design, we should now have a good understanding of the customer and business pain points. Before we dive into developing software solutions, we need to translate those problems into potential business and customer outcomes. And we need to use all the information captured from those artifacts and the information radiated from those practices to ensure we deliver outcomes that will really matter.

In this chapter, we're going to explore all the learning done in the *Discovery* section of the Mobius Loop so far to clearly define and articulate the goals and outcomes the team is striving to deliver to happy stakeholders and end users. This includes:

- Explaining the difference between outputs and outcomes
- Why we should have Target Outcomes
- How to capture Target Outcomes
- Some examples of Target Outcomes
- How to visualize Target Outcomes
- How to optimize and chain Target Outcomes with other practices

Setting Target Outcomes can include end user (customer-)based outcomes as well as business- and capability-based outcomes. When using Mobius, outcomes should be measurable. We should always be able to assign a numerical value to the current state and the target state. By regularly inspecting these measures, we can establish whether we are moving toward the outcome, whether we've met it, or if we need to pivot and change our approach if we're not meeting them.

What Is an Outcome?

As described in *Section 1, Practices Make Perfect*, the Mobius Loop encourages an iterative, experimental approach to achieving outcomes through a continuous flow of innovation using continuous discovery and continuous delivery. An outcome is a result that occurs from doing something. In our case, the outcome is a change in human behavior that drives change in the culture, impacting long-term business results.

The Target Outcomes practice helps teams discover, write, align on, and share the desired behavioral change they would like to see achieved by application products. Sharing Target Outcomes helps the team deliver measurable results and align with stakeholders and team members. Referring to Target Outcomes can help prioritize and filter work so that we are always focused on delivering measurable impact.

The Target Outcomes practice involves creating a canvas that summarizes anticipated and hopeful measurable change captured by the team and stakeholders. This artifact serves as an information radiator, reinforcing Target Outcomes used during other practices, activities, and discussions from other parts of the Mobius Loop. The Measure and Learn element of the Delivery Loop focuses on evaluating the outcomes. We'll explore that in *Section 5, Deliver It*. This chapter is about setting the outcomes when in the Discovery Loop.

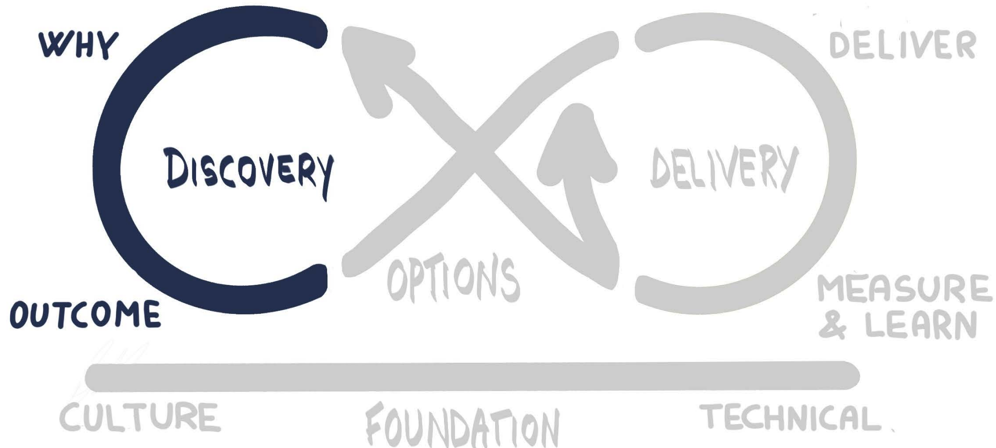


Figure 10.1: The Discovery Loop

In the previous chapter, we introduced Impact Mapping and described the differences between Impacts, which are measurable changes we want to achieve on the Actors, and a Delivery, which comprises of implementable pieces of work. Let's dive deeper into the difference between outcomes and outputs because it is very important to get this understanding.

Outcomes versus Outputs

An output is a piece of work that has a defined scope that a team completes. For example, implementing a feature, executing an experiment or carrying out some research, a UI prototype, or a technical spike. Organizations sometimes push to increase outputs because they think more features mean an increase in value. But that is not always the case. In fact, more often than not, less is more, and not all outputs translate directly into value. For example, implementing a feature does not add value if customers do not ever use the feature or, for that matter, never requested it in the first place. Think of the systems you have used that are overloaded with features that are never used, but where one or two key features are used all the time. When a team emphasizes outcomes over outputs, the team can iterate to find the right features that deliver value.

Teams sometimes have difficulty identifying shared outcomes because the differences between outcomes and outputs are unclear or they are unable to see the linkage between outputs they are proposing to deliver and the outcomes the organization wants them to achieve.

Joshua Seiden describes, in his book Outcomes Over Output,[1] how **an outcome is a change in human behavior that drives business results**. We can show the subtle but very important differences between outputs, outcomes, and impacts in the figure below.

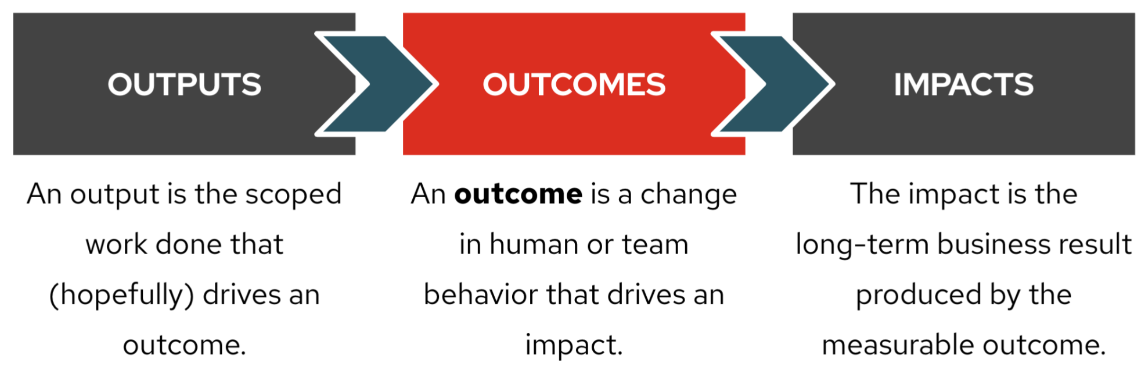


Figure 10.2: Outputs, Outcomes, and Impacts

1 https://www.senseandrespondpress.com/managing-outcomes

By switching the conversation to be focused on outcomes, we are thinking about the impact we want to have on users and other stakeholders as demonstrated by the Impact Map through the connection between the Impacts and Actors. From our Impact Map, we have identified the different Actor groups we can actively engage with to test our hypothesis and learn by experimentation, delivery, and feedback.

Why Have Target Outcomes?

The Target Outcomes practice creates shared alignment and purpose with a team, their stakeholders, and their customers.

By constantly displaying Target Outcomes on a big visible information radiator, the team can be reminded of the success criteria for the project. These reminders guide the team toward project success in day-to-day activities, such as:

- Evaluation and prioritization of work items
- Discussions that occur in other Mobius Loop practices

Emphasizing outcomes over outputs is a fundamental concept in the Open Practice Library. Identify Target Outcomes early on in new projects to ensure alignment from the beginning. Existing projects can run the Target Outcomes practice to assess the team's alignment and verify that the project has shared outcomes.

Any one of the following scenarios might indicate that a team will benefit from establishing Target Outcomes:

- Your team receives praise for successfully delivering a new product or feature within a given time frame and budget. However, after delivery, the team is surprised to learn that customers do not use the new product. Because the project is finished, the team cannot quickly adapt the product or feature in response to the feedback.
- Daily team interactions focus on completing the feature instead of completing a version of the feature that achieves desired results.
- Over-engineering of a feature or solution such that it is gold plated and goes way beyond what was really needed.
- Your teams celebrate a successful delivery before customers interact with the product or provide product feedback.
- The final product deliverable fails in the marketplace, despite a considerable investment of people and adhering to agile delivery methods. The use of iterations during delivery did not result in the delivery of a successful product.
- The scope for a team deliverable changed unexpectedly.

Let's look at how we run a short workshop to capture Target Outcomes.

How to Capture Target Outcomes

We often run the Target Outcomes practice immediately after running Discovery – Why and Who practices such as creating a North Star and/or an Impact Map. We often conclude these sessions with a confidence vote by everyone involved to establish a shared view on how aligned we, as a group, feel we are.

This should have resulted in a shared understanding of context across the team. If this hasn't worked, consider some of the other practices in this part of the Open Practice Library, such as Start at the End,[2] News Headlines,[3] or Start with Why,[4] to establish a shared team purpose or context. Remember, the Open Practice Library is a toolbox, so if the tools you've tried have not quite worked, try another one!

Based on the group's learnings from these activities, use a silent collaboration technique, or another brainstorming technique, to create a collection of potential Target Outcomes.

Use affinity mapping[5] to identify patterns and groupings of potential outcomes. Use a group discussion, or another technique such as Dot Voting, to achieve consensus on a small set of potential outcomes.

Review the potential outcomes and structure them to ensure they are measurable outcomes. Identify how to quantify and measure the outcomes. Identify a common structure to express the outcomes. Review the list of measurable outcomes. Discuss the outcomes with all the stakeholders to ensure that the outcomes are not too narrow in scope and are satisfactory to everyone. Ensure everyone in the meeting consents to the measurable outcomes.

To help articulate valid outcomes and evaluate proposed outcomes, the Mobius QuickStart Guide[6] by *Gabrielle Benefield* and *Ryan Shriver* suggests focusing on the following five attributes:

---

2 https://openpracticelibrary.com/practice/start-at-the-end/

3 https://openpracticelibrary.com/practice/news-headlines-aka-cover-story/

4 https://openpracticelibrary.com/practice/start-with-why/

5 https://openpracticelibrary.com/practice/affinity-mapping/

6 https://mobiusloop.com/kit/

1. **Name** provides a simple description statement such as **Decrease Time-to-Checkout**. The verb is typically the direction of improvement (increase or decrease) and the noun is the subject of improvement.
2. **What to Measure (Scale)** provides a consistent unit of measurement. Examples could be seconds to complete checkout.
3. **How to Measure (Method)** indicates how and when to measure, such as with a daily web analytics report including who is responsible for gathering and publishing the data and how frequently this is done.
4. **Baseline Measure** provides the current level and the starting point for comparing future measurements against. Examples could be 200 seconds to check out.
5. **Target Measure** is the desired success level in absolute or relative terms. Examples could be checkout times of 60 seconds or 75% improvement checkout speed by Q4. Target Measures can include a date clarifying by when the target is expected to be achieved.

Representing these five attributes in an arrow diagram similar to *Figure* 10.3 can help visualize the outcome.

Figure 10.3: The arrow representation of an outcome

Alternative formats exist that can help define outcomes. Experiment with alternative formats to discover if the team creates outcomes more effectively using them. For example, the SMART criteria (introduced in the previous picture) and summarized again below define five characteristics for an outcome or goal: specific, measurable, achievable, realistic, and time-based.

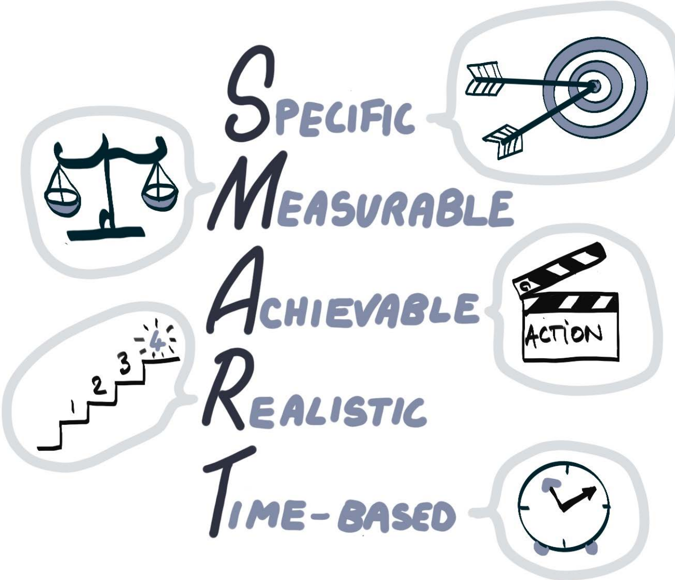


Figure 10.4: SMART outcomes

A stronger outcome statement is one that is measurable or can be broken down into a number of measures. Our approach to delivery is centered around moving to an iterative approach with regular releases of incremental value. We want to measure the outcomes as of now and then be able to measure the outcomes as we start to run experiments with feature ideas.

When starting our delivery, we should be able to regularly pause and measure the impact of the experiment or initiative to ensure that we learn from it and what we are doing is either taking us toward our target-measurable outcome or triggering a pivot or re-think.

Let's look at some examples.

Examples of Target Outcomes

To provide context for the Target Outcome examples in this section, consider the scenario of Issue Resolution Time.

During early Discovery Loop activities, a software product team uses affinity mapping to analyze customer feedback data. The analysis shows that many customers are unhappy with the time required to resolve issues with the software. The team believes that the dissatisfaction with response time is affecting customer retention.

The team uses the **Metrics-Based Process Mapping** (**MBPM**) practice, as introduced in the previous chapter, to identify significant bottlenecks that are negatively impacting response times. The exercise identifies several inefficient process components.

Other Discovery Loop activities reveal that operations stakeholders are nervous about more frequent product deployments. Frequent deployments cannot result in additional planned or unplanned work for operations personnel.

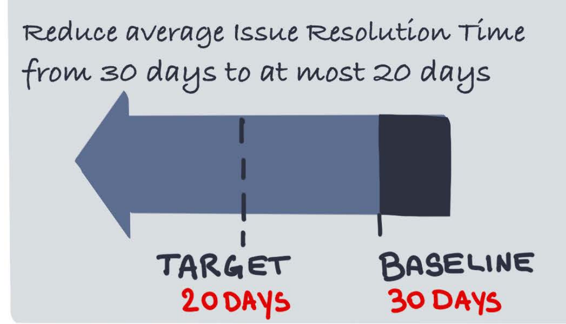
The team uses the Target Outcomes practice to:

- Discuss the results of the MBPM practice and other Discovery practice findings.
- List potential goals of the team's project.

As a result of the Target Outcomes practice, the team creates the information radiator visualizing three Target Outcomes.

In this example, the service provider previously had a service level agreement to always resolve a complex provisioning issue (including, if necessary, site visits) within 30 days. As a result of the upcoming product development, the target is to move that to 20 days. In doing this, the organization hopes to improve customer satisfaction survey scores (out of 5) from 3.2 to 3.5. They want to do this without creating any additional manual work performed by those who work on the operations.

TARGET OUTCOMES



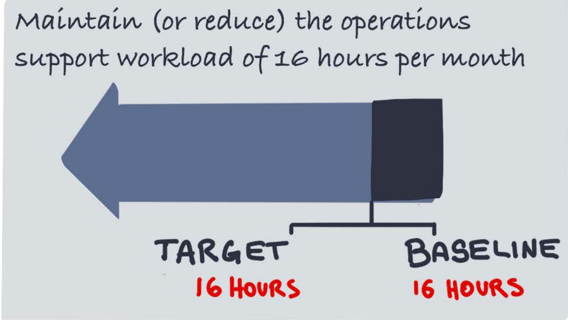


Figure 10.5: A Target Outcomes example

The Target Outcomes artifact reminds the team to:

- Focus on finding and designing solutions that reduce the resolution time for customer issues. Successful solutions reduce the average resolution time by at least 10 days and do not increase the operations personnel's workload.
- The team believes that a successful solution will also increase average customer satisfaction scores by at least 0.3.
- If the team can deliver a solution that achieves some (but not all) of the Target Outcomes, then the team uses practices in the Options Pivot to decide if a pivot is required. They may decide to do more delivery (another Delivery Loop) to try and move the measurement further toward the target. They may decide, following feedback, that the new measure is sufficient to move onto another Target Outcome. Or some reprioritization may take place. These options will be explored in detail in the next chapter.
- The team might decide to:
    - Revisit Discovery practices, and update the Target Outcomes for the next quarter based on what the team learned from the previous quarter of activity.
    - Pursue a different solution to achieve the same three Target Outcomes.

The best way to articulate and understand where a product is in relation to achieving target outcomes is to use visualization.

Visualizing Target Outcomes

A single visual showing an outcome's key attributes helps everyone clearly understand the current state and desired future state. In Mobius, these are known as outcome trackers. As ideas are planned and implemented, these visuals are updated to show progress over time. Visualization in public spaces such as team rooms or community areas helps keep the outcomes omnipresent throughout planning and design activities. This gently reminds everyone of what success looks like in a way that is accessible and clear. Showing the progress achieved so far gives teams the satisfaction that they are moving forward in a positive way. Showing lack of progress or movement in the wrong direction can trigger a quicker reaction and could even be coupled with Realtime Retrospective updates or a Stop-the-World event – practices we introduced in *Section 2*, *Establishing the Foundation*, when building our Open Culture foundation.

Figure 10.6: Visualizing Target Outcomes

Visualizing target outcomes creates alignment, shared understanding and conversation about how they can be improved and optimized.

Optimizing Target Outcomes

As with all of the artifacts we introduce from practices in this book, we are **never done** and can always improve as we learn more from subsequent delivery and (more) discovery activities. The Target Outcomes practice is intended to provide direction at the time of the practice. As we measure and learn from the outputs of our delivery cycles, we may well learn that some of the assumptions we made during the Discovery Loop are now invalid and we can rerun Discovery practices with this additional learning to produce an improved set of Target Outcomes.

Teams can refer back to the agreed Target Outcomes throughout the loops to guide their decisions. Use the Target Outcomes to prioritize work and reject unplanned work that does not contribute to outcomes.

Target Outcomes must have a time frame. Measure each Target Outcome regularly to assess progress, including at the end of the designated time period. When the time period for each outcome ends, the team evaluates the situation and decides to add new outcomes, with corresponding time periods, if needed.

As we gain more information from customers as they feed back on increments of the evolving product, it can lead to course direction changes and perhaps a need for new or revised Target Outcomes. We'll explore this in much more detail when we look at the Options Pivot coming out of Discovery (in the next chapter) and the Options Pivot coming out of Delivery in *Section 5, Deliver It*.

Target Outcomes is not a practice that can be done alone. It needs to be fed by several other practices and the result of it drives inputs to several new practices.

Chaining Target Outcomes with Other Practices

The Target Outcomes practice can be run as a standalone, but we strongly recommend running it after some of the other Discovery practices. Often stakeholders will say upfront that they categorically know what the Target Outcomes of a product or initiative should be. However, after articulating the North Star of the product or organization, running an Impact Mapping session, using a human-centered design practice such as Design Thinking, and considering a future state architecture using Event Storming or MBPM, their view may significantly change. Target outcomes that are backed by a deeper discovery of context and increased empathy with users and stakeholders allow for a far better view of what exactly is to be achieved and why.

So, we're starting to see the impact of chaining these practices together. Having all of the artifacts from the practices clearly visible to the practitioners means they can keep one eye on all of the fantastic learning achieved from their practices, for when they come to distill all the learning down into a set of target-measurable outcomes.

Another great example of a practice that should not be forgotten about is Priority Sliders. During this, we gained alignment and consensus of many stakeholders to agree on the priorities of the product or project. So, we should constantly remind everyone of what we agreed here when articulating Target Outcomes.

This is where great facilitation really takes place. A great facilitator will not be central to the conversations about context and detail but they will ask open questions about the information radiated during previous practices. They will also keep in mind what the group should be targeting and trying to achieve. They want to achieve target-measurable outcomes.

Having a well-prepared space with plenty of sticky notes and pens can help facilitate a great Target Outcomes session. This can also be achieved in a virtual environment by using a template such as the one below, which can be downloaded from the book's GitHub repository.

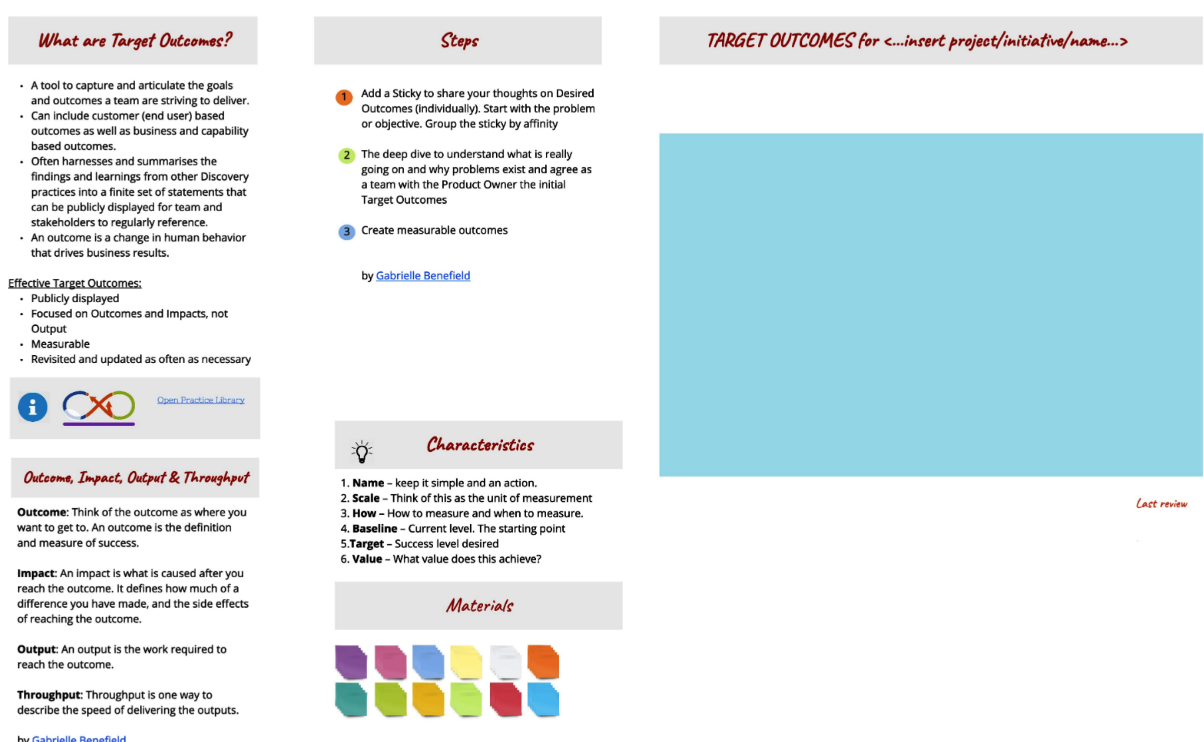


Figure 10.7: Target Outcomes canvas

Let's get back to our PetBattle team and explore the Target Outcomes that resulted from their trip around the Discovery Loop.

PetBattle Target Outcomes

The PetBattle team aligned on the following Target Outcomes:

1. PetBattle is generating revenue from an increased active user base.
2. PetBattle is always online.
3. An enthusiastic team that is excited to build and run PetBattle.

There are three distinct categories represented here. #1 – generate revenue – is clearly a business goal that is directly linked to increased user activity (an assumption we can test). Target Outcome #2 refers to the stability and availability of the product PetBattle – when the product went viral,

which crashed the system, they had very poor user experience and no chance to potentially gain revenue as PetBattle was not online for users to access! Target Outcome #3 is a team culture-based outcome – by having an engaged team who have a real sense of product ownership, they will be happier in their workplace and be more responsive and motivated to grow and improve the product in the future (another assumption we can test!).

Let's take our first Target Outcome, which is that PetBattle is generating revenue from an increased active user base. This can easily be seen as a measurable outcome. We can quantify at any given time the number of users registered in our system, the number of visitors to our site, the number of engaged users, the amount of engaging activity users are doing, and, of course, revenue secured through advertisers.

We can nicely visualize the baseline metrics we have at this point and the target metrics we want to get to from near future increments of the product.

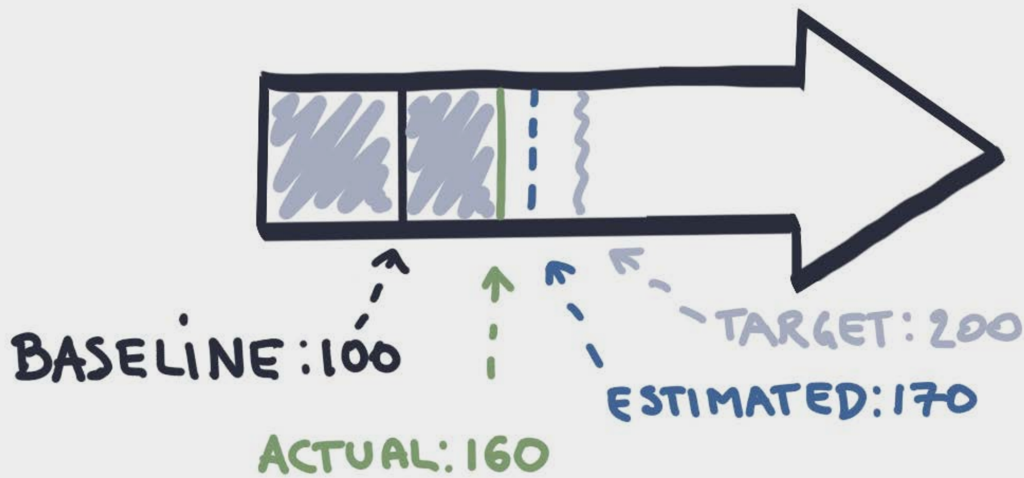


Figure 10.8: Target outcome – Increased active user base (in thousands)

Let's look at our second Target Outcome: that PetBattle is always online. From the early version of the app, there have been stability issues due to the huge popularity. We can collect a metric now on what we believe represents the uptime of the existing site and visualize how we want to improve this to the target-measurable outcome of always being online.

Figure 10.9: Target outcome – Always online

Our third Target Outcome is that we have an enthusiastic team that is excited to build and run PetBattle is focused on employee engagement and satisfaction. This can also be measured and visualized through quantifying the level of happiness and engagement of the workforce.

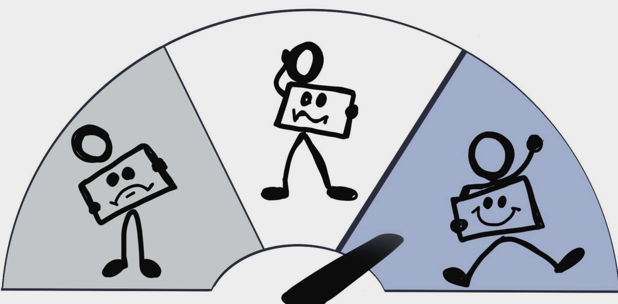


Figure 10.10: Target outcome – Increased employee engagement and satisfaction

Lots of conversation and collaboration contributes to reaching this point of consensus around Target Outcomes. When we eventually reach consensus across all stakeholder groups that these are the outcomes we will be trying to reach, we like to keep them highly visible and accessible. They should be a reference point for everything we do from this point onwards. As we start to deliver features and increments of our product, they should be taking us toward the Target Outcomes we've agreed. Capturing measurements from our products to give us real-time data out of our iterative delivery enables us to inspect, assess, adapt, and refine previous decisions made.

In the example above, we saw three very different focuses on outcomes. They were not just about the application or business but spread between the application, what the platform enables, and the performance and motivation of the team. This is a pattern we have observed with many customers.

The Balance of Three: People/Process/Technology

In our Open Innovation Labs residencies, we work in a time box of 4-12 weeks to immerse one of our customers' product teams into DevOps culture and practices using OpenShift.

We start out these engagements using practices such as Impact Mapping and human-centered design practices and aim to establish a set of Target Outcomes for the residency engagement.

We've noticed that where the outcomes are not just focused on technology or the application or the culture but a combination of all three of these, the eventual impact of the engagement is much stronger.

One of our early engagements established these three outcomes:

- A new standard for working with DevOps has been set, including the skill set of the team.
- A new application is running in production.
- The team feels great, excited, and empowered when working with this product.

One of the outcomes here is related to the technology and platform (DevOps). One is related to a revenue-generating application that we were to build during the residency. The final one is about the culture and the team.

This is a great balance of Target Outcomes for a first team in an organization that might be moving toward DevOps.

The outcomes above do lack measures and perhaps were not as SMART as they could have been.

A more recent set of outcomes with a customer in the United Arab Emirates was as follows:

- [Customer] can seamlessly deploy workloads several times a day to prove self-service capabilities.
- [Application] is seamlessly deployed as an example to prove a repeatable pattern for deploying N other applications and to decrease the time to market for changes.
- Development is a first-class citizen. Multifunctional teams are empowered using Site Reliability Engineering best practices as a Product Team.

These outcomes start to introduce the idea of something measurable. The magic word in the second outcome is decrease. There is a measurable outcome here around decreasing time to market for changes. This is something that could be measured before, during, and after the engagements.

The final example comes from a non-government organization focused on healthcare:

- Increased level of empowerment and skills such that the [application] can be confidently owned and managed.
- Create a more Open Data model for [application] for [users] to be increasingly attracted to our system.
- A clear development lifecycle on a scalable, secure, and stable platform as measured by increased security, automation, and monitoring (auditability)

Again, we see the balance of team, application, and platform in these three outcomes. All three outcomes can also be measured.

Let's now look at a simple example of how we have measured an outcome.

Target Outcomes from a Telecoms Product – Stopwatch at the Ready!

One of my favorite examples of Target Outcomes that were user-centered comes from a telecommunications company we worked with in Ireland.

We were building a new enterprise portal for corporate customers to browse and buy their company mobile phones.

The team used design thinking practices to establish empathy with employees of the top three enterprise customers. Empathy mapping these users and visualizing their existing processes using a scenario mapping practice highlighted several pain points and opportunities for increased user experience in our new solution.

The Target Outcomes were written in a format that encompassed a Who, What, and Wow – who was going to benefit from the outcomes, what was the outcome, and what was the Wow factor, or the measure.

The three outcomes that we distilled the learning of our Discovery practices were:

- An employee can search for, view, and select their desired phone in less than 30 seconds.
- An employee can place an order to upgrade their device and receive their new phone ready to use with their existing number in less than 48 hours.
- An employee can login to the portal using their company credentials, so they don't need to set up or remember another username and password.

These outcomes are very micro-focused on a particular product. The first two are very measurable and you can imagine our team had the stopwatch out in user testing against the first of them!

> I loved these outcomes because, written up on a big visible chart, it summarized for everyone walking by our team space what we were trying to achieve. We needed an app where time-constrained users could do what they needed to do quickly, they would receive their order quickly, and it would just work and they would not feel like they needed to engage with yet another company – their experience would remain aligned to their company employment.

We've now seen a few different examples of Target Outcomes set during engagements with different customers. One thing you may have noticed is that Target Outcomes can range from being very end user-centric to being more about the technology, platform, and team being used to develop the application. This leads us to explore the idea of primary outcomes (more end user-centric) and enabling outcomes (delivered by the platform and team).

Differentiating between Primary Outcomes and Enabling Outcomes

In the previous two chapters, we used several different practices to discover the Target Outcomes. Some of them are more functional-, application product-, and business-focused (the Why and Who), and some are more non-functional-focused (the How).

As we discover these different types of outcomes, we can explore relationships between them and see whether a hierarchy exists or not.

Gabrielle Benefield and *Ryan Shriver* explain[7] that *Enabling Outcomes, such as decreasing time to onboard new developers, improving test automation and reducing the time to build and deploy code, will hopefully improve primary outcomes such as improve customer experience.*

This may just be a hypothesis that needs validation through experimentation and delivery.

7 https://mobiusloop.com/kit/

Let's look at some common examples of outcomes that are a mix of primary and enabling outcomes.

- **Increasing Speed and Quality with Customers**:
    - **Improve Customer Satisfaction**: Measuring customer happiness using tools such as **Net Promoter Score** (**NPS**).
    - **Reduce Lead Time**: Elapsed time from the initial customer request or contact to delivery triggering customer happiness.
- **Increasing Quality**:
    - **Reduce Outages**: Number of operational incidents leaving customers with no service.
    - **Reduce Defects**: Number of defects in the product that impact customers.
- **Sustainability**:
    - **Improve Team Satisfaction**: A measurement of the happiness of a product team.
    - **Increase Employee Retention**: Turnover rate of employees.
- **Working**:
    - **Increase Throughput**: Total work items completed per unit of time.
    - **Improve Predictability**: Percentage of time team delivers on their commitments.
    - **Increase Agility**: Elapsed time to pivot to new priorities.
- **Efficiency**:
    - **Reduce Cycle Time**: The elapsed time to complete delivery activities such as build, test, and deployment of new features.
    - **Reduce Work in Progress (WIP)**: Number of items a team is working on at any moment.
    - **Reduce Technical Debt**: How much you're spending on reworking your codebase.

The organization's specific context will drive whether they are more focused on one or two of these and set them as their primary Target Outcomes. If the motivation is more business-focused, their primary outcome is likely to be around the performance of their products with their customer, and their customer's satisfaction.

Nearly all of the examples above can also be enabling outcomes, and we can form hypotheses of how improving these can drive improvements in more business-focused outcomes. These examples also steer our focus toward exploring software delivery techniques (such as DevOps) and the platform (such as OpenShift).

Software Delivery Metrics

In 2018, the book *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*, written by *Jez Humble, Gene Kim, and Nicole Forsgren*, was first published. We cannot recommend this book highly enough to explain how to effectively measure software delivery team performance and the science and statistics behind their extensive research.

The book addresses four metrics that can be used to assess the health of any DevOps organization. Two of them are more development-centric and measure market agility, and two are more Operations focused and measures of reliability.

1. **Lead time for change**: This is the time from code being committed to a source code repository to it being deployed to production. Shorter lead times for change is better because it enables faster feedback cycles and makes organizations better able to adjust to the marketplace.
2. **Deployment frequency**: This is how often an application is deployed to production and is an indicator of batch size. Smaller batch sizes mean deployments can be made more frequently, which also leads to more market agility.
3. **Mean time to restore**: This is how long it takes for systems to recover from failures in production. We want to reduce this which is important because we need to ensure we aren't speeding up delivery at the expense of poor customer experience from failures.
4. **Change failure rate**: This is the percentage of deployments requiring a rollback and/or fixes for a problem in production, which, again, we want to reduce as this is a secondary indicator of stability.

When we consider all four of these as enabling outcomes for a software product, we can see the strong hypothesis on how optimizing these will have a positive impact on the product's performance in the marketplace and overall customer satisfaction. Being able to have short lead times for change and a high frequency of deployments, while maintaining a low mean time to restore and low change failure rate, would suggest an overall high-performing product. That will eventually translate to a higher NPS, increased conversion rates, and even increased market share as customers tend to really like using fast-moving yet reliable software.

As this book has started to demonstrate, teams that successfully adopt the practices being introduced will see a positive effect in these software delivery metrics. The cultural practices, along with other practices such as metrics-based process mapping, identify the bottlenecks and handovers that, when removed, can speed lead times to change. The continuous integration and continuous delivery practices being used when we build pipelines allow us to increase deployments. And, with increasing levels of automation and continuous learning from practices such as retrospectives, we can keep **Mean Time to Repair** (**MTTR**) and Change Failure Rates to a minimum.

Platforms such as OpenShift provide an even further level of enablement. Without these platforms, we wouldn't be able to achieve market agility and reliability for the software products being deployed on the platform.

So, if the platform is the lowest level of enablement, are there metrics we can baseline and regularly check to ensure this enablement is being passed up to software teams and eventually to the business? Let's look at a few platform adoption metrics.

Platform Adoption Metrics

Some of the metrics you might want to capture about the platform include:

1. **Rate of Growth and Adoption:** How much change is happening in products provisioned on the platform? How many services are running?
2. **Lead Time to Onboard:** How long does it take to onboard a new developer or operator on the platform such that they have everything they need to do their job?
3. **Lead Time to Provision Projects:** How long does it take to provision a new project on the platform so it is running in production and usable?
4. **Platform Team Efficiency:** What is the number of full-time employees needed compared to the number of applications supported? Given that there should be huge economies of scale from platform adoption, we expect this ratio to substantially increase as platform adoption increases.
5. **Site Reliability Engineering metrics:** Given that the platform provides an enabler for the Site Reliability Engineering mindset, some of the metrics and associated practices can be radiated:
    - Service availability/service level objectives
    - Remaining error budget
    - Development time versus issue/support time

Again, we start to see the impact these measures can have on software delivery and operations of applications running on the platform, which, in turn, can be measured by the four software delivery metrics. And these metrics can generate a lagging effect on the quality of the software, the organization's products, and the happiness of their customer and employees. So, the value of capturing, measuring, visualizing, and sharing these metrics cannot be underestimated.

Let's look at our approach for growing this metric-centric mindset.

Continuous Metrics Inspection

We start small and focus on just one application, one team, one product. We do not want to boil the ocean by getting hundreds of metrics across an IT estate. Instead, we want to have a team that is empowered, enabled, and enthused by these metrics.

First, we need to baseline what we can by running a Target Outcomes workshop. This should include capturing the key primary outcomes – ideally two or three that the team are laser-focused on achieving.

Next, we look at what other metrics we know or can find out about existing software delivery and the platform. If we can get this information, let's make it visible. If not, let's commit to starting to capture it.

We make these outcomes and known metrics about the product, software delivery, and platform very transparent and visible. The more transparent you can be about current performance, the easier it will be to demonstrate improvement. Using Big Visible Information Radiators, there's no end to improvement. Having the information continuously radiating will enable informed decision making that is data-driven. It's all part of an ongoing process. Once an organization has invested in collecting metrics, it's time to start continually improving on them. The key here is an experimentation model that could involve process changes and ways of working, new architecture, and changes to team composition or organization structure. We will be exploring this in the next chapter as we consider our options.

Before that, we're going to create one final information radiator on this iteration of the Discovery Loop to provide a single-page summary of everything we have learned on this trip around the Discovery Loop.

Creating a Discovery Map

The Discovery practices described in this section all revolve around facilitating great conversations across the many different roles and functions across the organization. If you've used the Discovery practices we described in *Chapter 8*, *Discovering the Why and Who*, and *Chapter 9*, *Discovering the How*, you'll already have a large number of information radiators – a North Star, an Impact Map, human-centered design artifacts such as Empathy Maps, an Event Storm, Non-Functional Maps, MBPMs, Target Outcome canvases, and baseline metrics. And of course, there are many other Discovery practices in the Open Practice Library that you may have tried, and each tends to produce at least one information radiator canvas.

Within each of these artifacts is very powerful and valuable information, which is why we want to keep them all visible and accessible to anyone who is interested – be it team members or stakeholders. There will no doubt be wrong information and incomplete artifacts – that's kind of the point of this process. We only ever do just enough discovery to progress to the Options Pivot and start an iteration of delivery. By this, we mean there is a consensus across stakeholders, an alignment of shared understanding on product direction, and confidence in the delivery team that they have what they need to get started with delivery. As we learn more and more and get feedback and increase our number of measurements, we can improve and complete Discovery artifacts further (or maybe even try some new practices).

We do find it helpful to distill and harness all the information from the Discovery practices into a single summary that can be publicly displayed and used as an information radiator for those who may not have the time or interest to get into the detail. Having a single page that describes the product discovery and Target Outcomes provides a great opportunity for alignment and agreement of product direction.

There are many canvases openly available, including the Business Model canvas and the Lean canvas. The Mobius Loop has also open-sourced a Discovery Map, which is a simple graphic used to summarize:

- What have we just done a Discovery Loop on?
- Why are we doing this?
- What is the current situation?
- Who are we doing this for?
- What are the Target Outcomes and how will we measure them?
- What are the barriers and biggest challenges?

Figure 10.11: The Discovery Map by Mobius

We've used this map to summarize all learning for PetBattle.

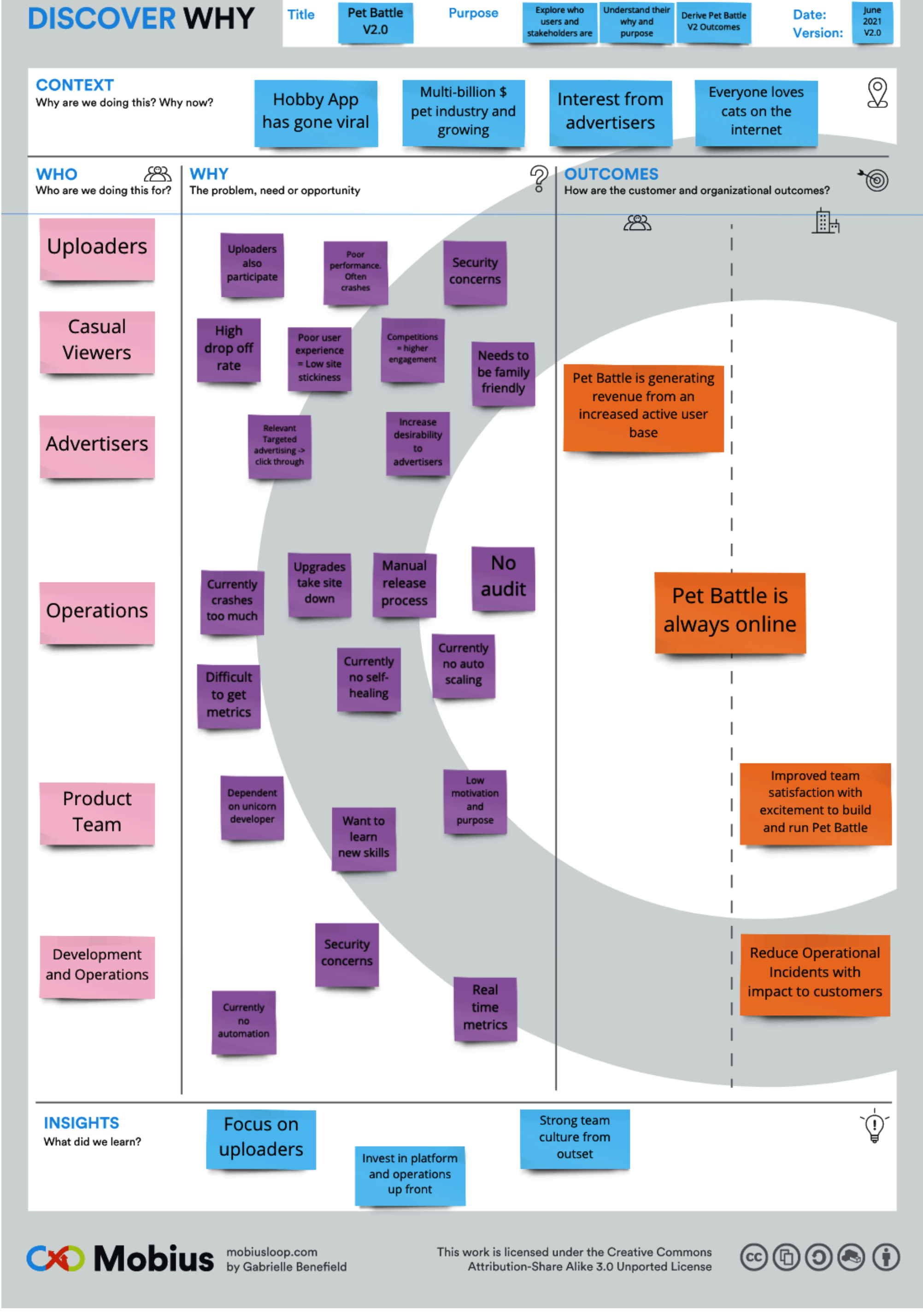


Figure 10.12: The PetBattle Discovery Map

Conclusion

In this chapter, we focused on setting Target Outcomes for the business problems and opportunities we uncovered in the previous chapters.

We explained what an outcome is and how it is different from output, and how we capture them, write them, measure them, and visualize them.

We dived deeper into measurable outcomes and metrics and explored the difference between primary outcomes and enabling outcomes. Enabling outcomes in a DevOps world is heavily focused on software delivery metrics and platform metrics.

You can read more and chat about the practice of setting Target Outcomes in the Open Practice Library at openpracticelibrary.com/practice/target-outcomes/.

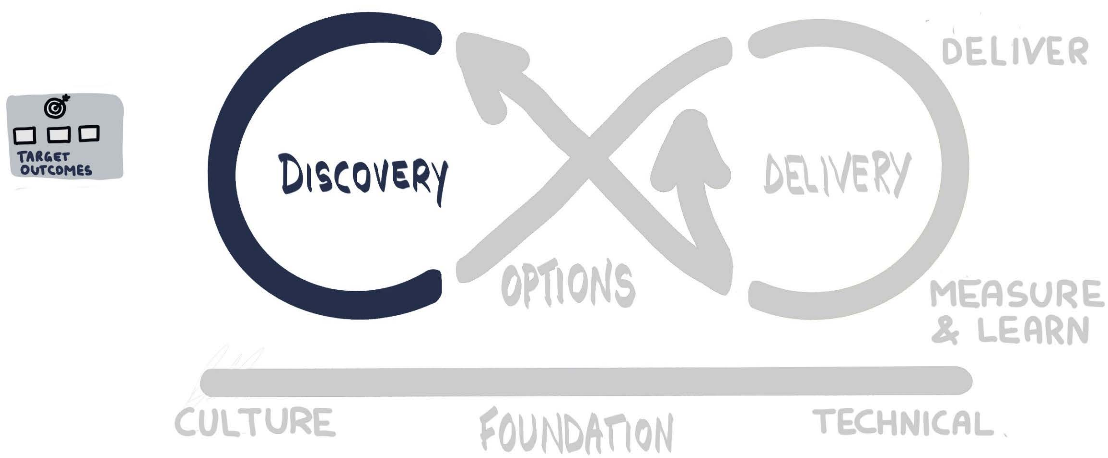


Figure 10.13: Adding a further practice to the Delivery Loop to set Target Outcomes

We wrapped up this Discovery section by summarizing all learning on the Mobius Discovery Map.

So, now we have built a foundation of open culture and open technology practices and have done some product discovery with some baseline metrics of the product, software delivery, and platform; we now need to decide what we're going to deliver first.

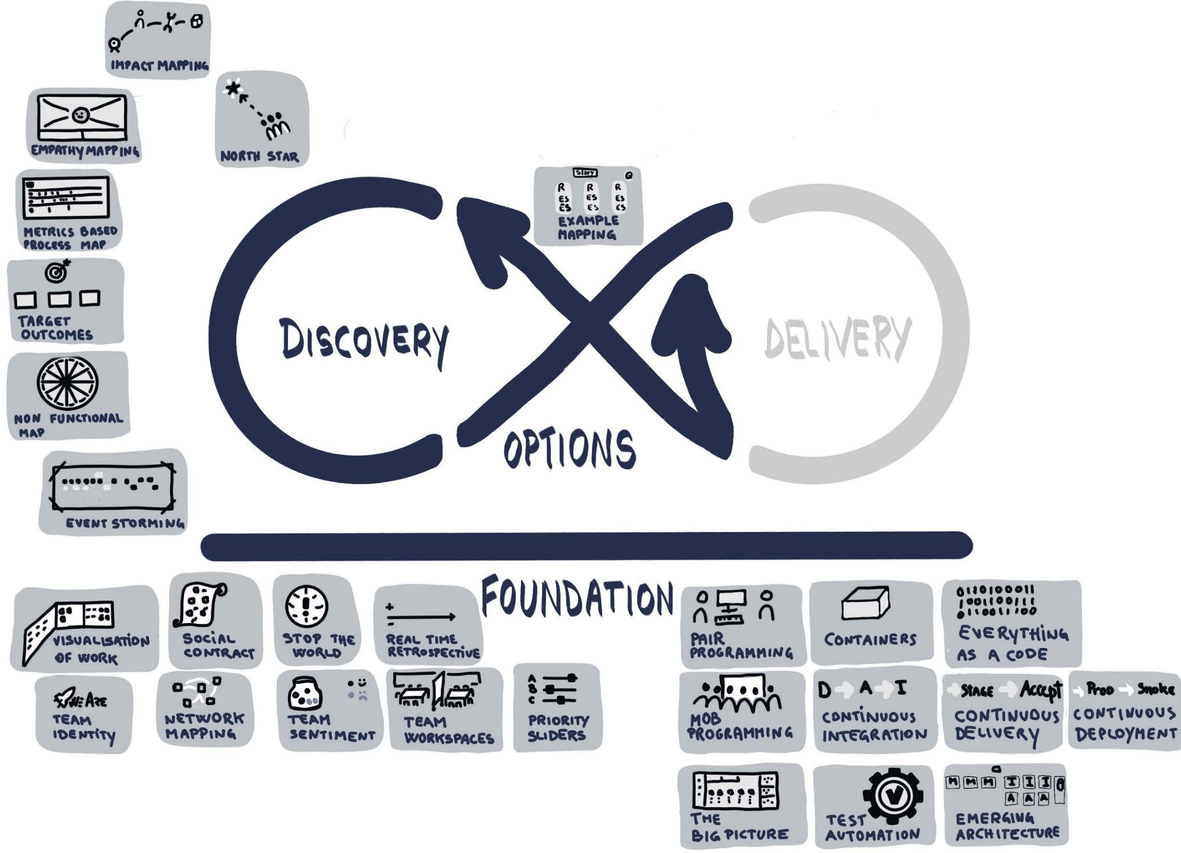


Figure 10.14: Practices used to complete a Discovery Loop on a foundation of open culture and technology

In the next chapter, we'll move to the Options Pivot and look at practices to prioritize and plan what we can deliver incrementally. We'll look at the trade-offs between feature delivery and non-functional work to improve software delivery and platform. And, we'll explore some of the advanced deployment capabilities offered by OpenShift and other platforms that will help us further decide on how to deliver the best value and products to customers.

Section 4: Prioritize It

In *Section 3*, *Discover It*, we worked our way around the Discovery Loop. We started with Why—why are we embarking on this initiative? What is our great idea? We used the North Star to help us frame this. We defined the problem and understood the context further by using the Impact Mapping practice to align on our strategic goal. Impact Mapping helped us converge on all the different actors involved that could help us achieve or impede our goal. Impact Mapping captures the measurable impacts we want to effect and the behavioral changes we would like to generate for those actors. From this, we form hypothesis statements about how the different ideas for deliverables may help achieve these impacts.

We refined this understanding further by using the human-centered design techniques and Design Thinking practices such as Empathy Mapping and Contextual Inquiry to observe and connect with our actors. We explored business processes and domain models using the Event Storming practice by generating a shared understanding of the event-driven process. Using the Event Storming notation, a microservices-based architecture started to emerge. We also discovered non-functional aspects of the design by using Non-Functional Maps and running Metrics-Based Process Mapping.

The Discovery Loop presented lots of ideas for things we can do in our delivery cycles—features we can implement; architectures that emerge as we refine and develop the solution by repeated playthroughs of the Event Storm; research that can be performed using user interface prototypes or technical spikes that test our ideas further; experiments that can be run with our users to help get an even better understanding of their motivations, pain points, and what value means to them; and processes we can put in place to gather data and optimize metrics.

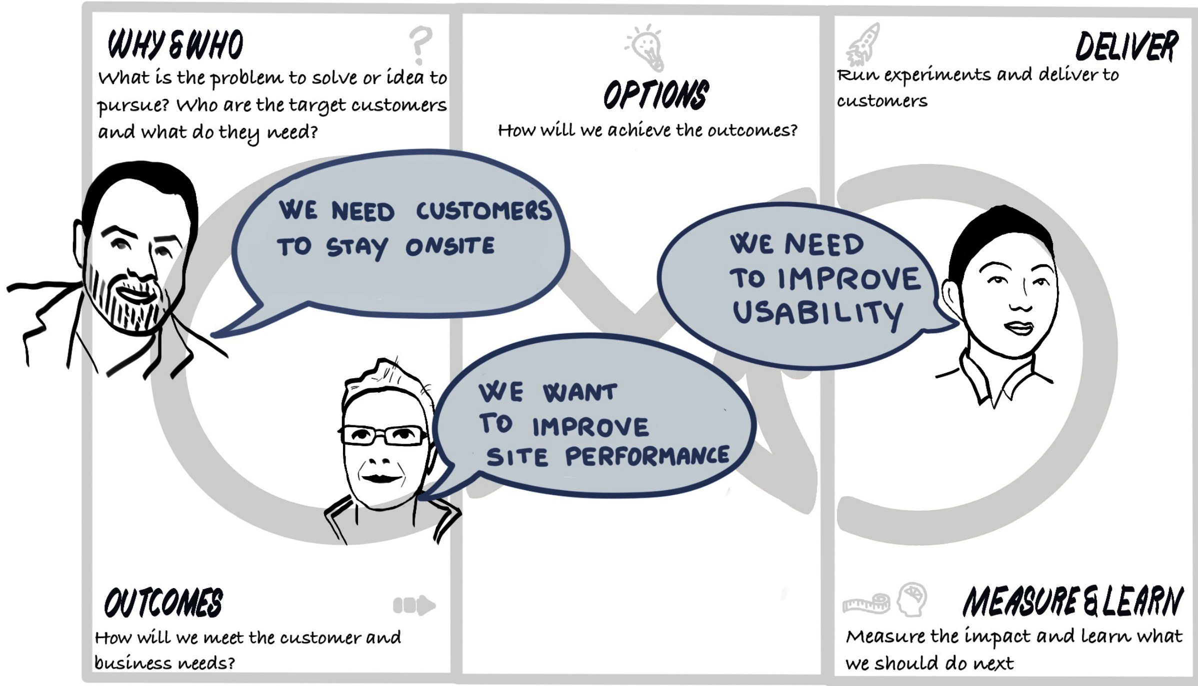


Figure 11.0.1: The Options Pivot – setting the scene

From just the first iteration of the Discovery Loop, it would be very easy to come up with hundreds of different tasks we could do from all the conversations and engagement that those practices generate. It can be a minefield visualizing all these ideas and it can take weeks, if not months, to generate tasks for a small team just from a short iteration of the Discovery Loop! So, we need to be careful to ensure we remain focused on delivering value, outcomes that matter, and that we don't get bogged down in analysis-paralysis in a world filled purely with busyness!

Before we left the Discovery Loop, we took time to translate all of this learning into measurable Target Outcomes. This started with the primary target outcomes associated with the business product, but we also took time to recognize some of the secondary targets and enabling outcomes that can help support development—especially those that can be enabled by software delivery processes and underlying platforms such as OpenShift.

With these outcomes visualized and presented using big visible Information Radiators, supporting metrics can also be baselined and radiated. We can now think about all those tasks and ideas that resulted from the Discovery Loop. But we can only do so by keeping an eye on those outcomes at all times and ensuring everything we do is directly or indirectly going to take us toward achieving them. This is where the real fun begins, because we're going to explore **how** we're going to achieve those measurable outcomes.

Mobius uses the word **options** instead of **solutions**, or the dreaded term **requirements**. Until we validate our ideas, they are simply wild guesses, so calling them solutions or saying they are required is not logical and there is no evidence to support them. Instead, we call them potential solutions, options, and we get to test them out in the Delivery Loop to prove or disprove the hypothesis that we have formed around those options. This drives us to a more data-driven approach rather than just simply guessing.

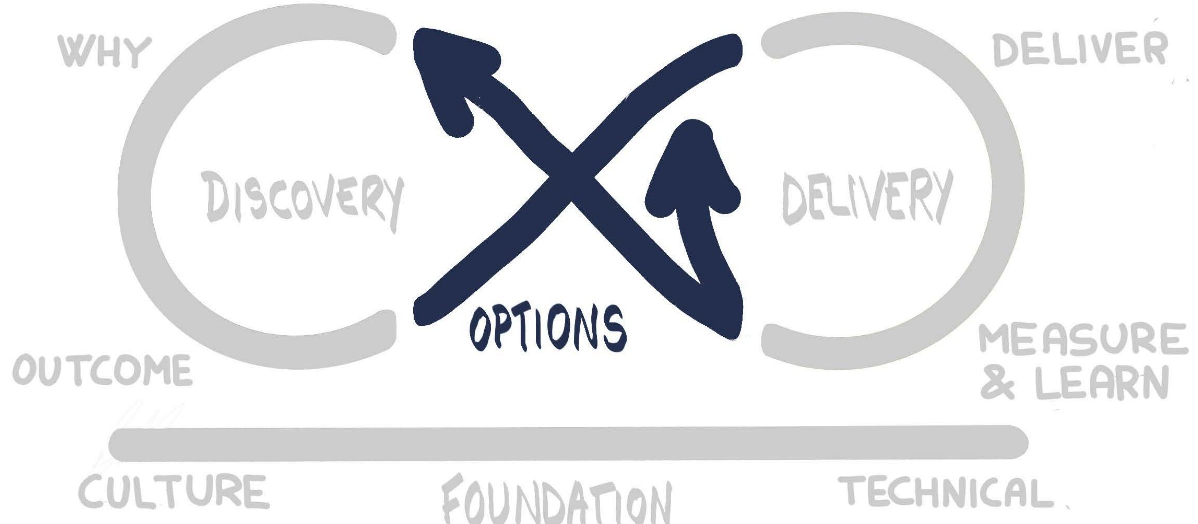


Figure 11.0.2: The Options Pivot

When we are on the **Options Pivot**, we decide which of the outcomes we are going to target next. We choose which ideas or hypotheses we need to build, test, validate, and learn from, as well as exploring how we might deliver the **options**. We also need to get a sense of priority. We never have the luxury of infinite time and resources, so prioritization is always going to be the key to achieving business value and fast learning. Learning fast is an important aspect here. We want to generate options that can validate, or invalidate, our ideas from the Discovery Loop so we can ultimately revisit and enhance them. Fast feedback is the key to connecting the Discovery artifacts with a validated prototype.

Chapter 11, *The Options Pivot*, will focus on the practices we use before we begin a Delivery Loop. We will return to the Options Pivot again after the Delivery Loop in *Section 7*, *Improve It*, *Sustain It*, when we take the learnings and measurements that have resulted from the latest Delivery Loop iteration and decide what to do next given these findings.

11

The Options Pivot

During the Discovery Loop, we started to come up with lots of ideas for implementation. The Impact Map gave us deliverables that formed hypothesis statements. The human-centered design and Empathy Mapping practices gave us ideas directly from the user. The Event Storm gave us standalone features (triggered by commands) that can be implemented using standalone microservices (codifying the aggregate). The Metrics-Based Process Map and Non-Functional Map gave us ideas on how we can speed up the development cycle and improve security, maintainability, operability, scalability, auditability, traceability, reusability, and just about anything else that ends with ability!

The next step after the Discovery Loop is the Options Pivot, where all the information from these practices that we've used gets boiled down to a list of options for actions to take and decisions to make on what to deliver next.

The Options Pivot is the heart of the Mobius Loop. On the left-hand side of it is where we absorb all the learning and Target Outcomes we aligned on in the Discovery Loop. We generate further ideas. We refine ideas on what to deliver next and then choose the options to work on. Later in the book, in *Chapter 17*, *Improve It*, we'll look at the right-hand side of the Options Pivot. This is where we adapt our approach based on the measurements and learnings from a completed iteration of the Delivery Loop. We decide whether to do more Discovery, more Delivery, or Pivot completely. We refine what to discover or deliver next.

Remember, we are working in fully autonomous, cross-functional teams. We don't have separate testing teams for performance or usability, so we can't assume work items associated with these functions will be dealt with on the other side of the wall! This makes our job difficult as we have to weigh up the relative values of different options. We have to decide between new feature development, urgent bug fixes, platform improvements to speed up development, usability improvements, enhancements to security, and many other aspects that will make our product better.

In this chapter, we're going to do the following:

1. Visualize all the work we might do using the **User Story Mapping** practice.
2. Organize all our work into small, thin slices of value using the **Value Slicing** practice so that we can continuously deliver value.
3. Start the **Design of Experiments** practice to test hypotheses that emerged during the Discovery Loop.
4. Prioritize by exploring different practices that help us in the Options Pivot, including **Impact and Effort Prioritization**, **How/Now/Wow Prioritization**, and the **Design Sprint**.
5. Form the initial **Product Backlog** with traceability to all preceding practices.
6. Set up **Product Backlog Refinement** to happen on an ongoing basis.
7. Apply **Economic Prioritization** to Product Backlog items.
8. Explain the importance of the **Product Owner** role in achieving all of the above.
9. Explore how experiments can be supported by some of the **advanced deployment capabilities** enabled by the OpenShift platform and how we can plan to use these to ensure we maximize learning from our Delivery Loops.

Let's start with one of our favorite visualization practices to radiate and plan lots of small incremental releases by slicing value.

Value Slicing

We are approaching the part of the Mobius mental model where we will start delivering increments of our solution. They will vary from running short prototypes and technical experiments or spikes, to conducting defined user research, to implementing features that have resulted from Event Storming and other Discovery practices.

An iteration of the Delivery Loop is not prescribed in length. If you are using a popular iterative agile delivery framework such as Scrum, an iteration of the Delivery Loop translates well to one sprint (a fixed time-box between one and four weeks). If you are using a more continuous delivery approach such as Kanban to enable an ongoing flow of value, each Delivery Loop may simply represent the processing of one Product Backlog item and delivering it into the product. You may even be using a non-agile delivery methodology such as Waterfall whereby the Delivery Loop is more singular and slower to move around. The Mobius Loop is agnostic to the delivery approach. But what is consistent regardless of the delivery approach is the idea that we seek to deliver high-value work sooner, establish important learning more quickly, and work in small batch sizes of delivery effort so we can measure and learn the impact to inform our next set of decisions.

To help us break down all our work items and ensure they are grouped to a level that will form small increments of value, we use popular visualization and planning practices.

Simple path mapping techniques break the work down by mapping back from the Target Outcomes to the least number of steps needed to deliver it. There are many other practices, such as journey mapping, story mapping, future state mapping, service blueprints, and more. Mobius is less concerned with the how, as long as you focus on finding the simplest way to deliver the outcomes. This technique we have found works very effectively is called Value Slicing.

Let's look at how we approach Value Slicing.

First, we note all of the standalone work ideas that have been generated by the Discovery practices. Our focus here is now on Outputs (and not Outcomes) as we want to group all of our deliverables together and form an incremental release strategy that delivers the outcomes. A starting point is to copy each of the following from existing artifacts:

- Deliverables captured on the Impact Map
- Commands captured on the Event Storm
- Ideas and feedback captured on Empathy Maps
- Non-functional work needed to support decisions made on the Non-Functional Map
- Ideas and non-functional features captured during discussion of the **Metrics-Based Process Map** (**MBPM**)
- All the other features and ideas that have come up during any other Discovery Loop practices you may have used and the many conversations that occurred

Here are a couple of tips we've picked up from our experience. First, don't simply move sticky notes from one artifact to this new space. You should keep the Impact Map, Event Storms, Empathy Maps, MBPMs, and other artifacts as standalone artifacts, fully intact in the original form. They will be very useful when we return to them after doing some Delivery Loops.

Second, copy word-for-word the items you're picking up from those practices. As we'll see in the coming chapters, we will really benefit when we can trace work items through the Discovery Loop, Options Pivot, and Delivery Loop, so keeping language consistent will help with this. Some teams even invest in a key or coding system to show this traceability from the outset.

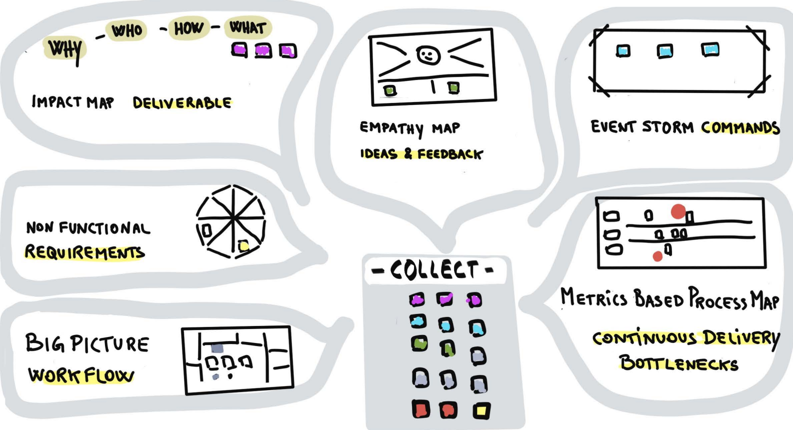


Figure 11.1: Collecting information and ideas from Discovery Loop practices

To start with, simply spread all the items across a large work surface. There's something very satisfying about standing back and seeing all the possible work we know of in front of us. It can be amazing to see just how much has been ideated from those few practices. It can also be a bit chaotic and daunting. This is why we need to start organizing the work.

If you're working virtually with people distributed, having a Canvas such as the following one (and available for download from the book's GitHub repository) may be helpful:

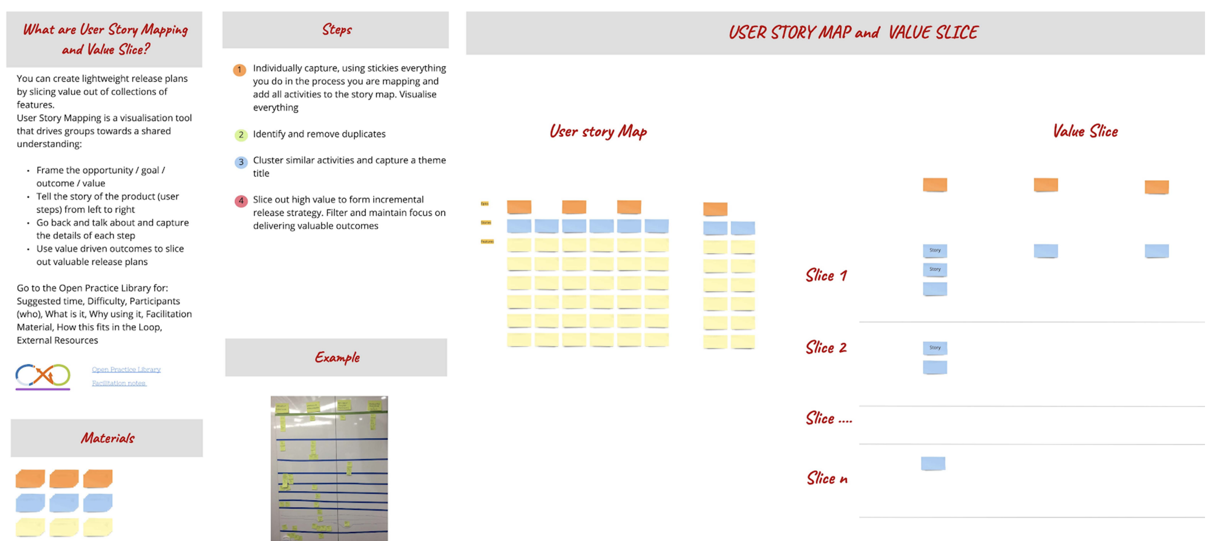


Figure 11.2: User Story and Value Slice Map template

Next, remove any duplicates. For example, you may have identified a deliverable on your Impact Map and the same feature has ended up in your Event Storm. Your user interviews may also have found similar feature ideas captured on Empathy Maps. Where there are identical features, remove the duplicate. If the idea can be broken down into smaller standalone ideas, refactor and re-write your sticky notes to have these multiple ideas. The more the better in this practice!

The next step is to categorize each of the items into some kind of common theme and give that theme a title. We're looking for something that brings all of the items together. If you were to put each item into a bucket, what would the label on the bucket be? A top tip is to start with the Target Outcomes that were derived from the Discovery Loop and set them as the headings to categorize each item under. The reason we do this is that we want to work with an outcome-driven mindset. We have agreed on some Target Outcomes so, really, every work item we are considering should be taking us to one or more of those outcomes. If we pick any one of the items and can't easily see an outcome it will help achieve, we should be questioning the value of doing that thing at all. (There are cases where such items that don't map to outcomes are still important, so if this does happen, just give them their own pile.)

We should end up with all items in a neat, straight column directly beneath the Target Outcome they are categorized under.

If we have a good, well-thought-out set of Primary Outcomes and Enabling Outcomes, it should be a very positive exercise mapping all of the features, experiments, research ideas, and so on to an outcome. This exercise should be collaborative and include all members of the cross-functional team. Developers, operators, designers, Product Owners, business SMEs, and so on will all have been involved and provided input to the preceding Discovery Loop practices. They should remain included during the Options Pivot to ensure their ideas and initiatives are understood and included on the map.

The resulting visualization of work should include functional features and non-functional initiatives. All of the work that can take place on the platform to enable faster and safer development and quicker release of product features should be shown. If we stand back at the end of the exercise, we should start to see our delivery loops starting to emerge.

Figure 11.3: Clustering tasks under Target Outcomes

The next step is to prioritize all tasks and items on the board. This is never easy but nearly always needed. If you have worked on a project where time has not been an issue and it's been obvious that the team will have all the time, they need to confidently deliver everything asked of them, you are in a unique position! That has never happened to us and there has always been a need to prioritize work and choose what not to do! This can start with the Product Owner deciding his or her perspective on priority. However, as we progress through this chapter, we'll look at a few practices and tools that you can bring out to help with prioritization in a collaborative environment and drive consensus. Executing those practices can then be reflected on this value map we're creating.

We like to attempt to prioritize each column. So, take each Target Outcome with all of the features and other items that we believe will achieve them and prioritize them. The most important and compelling items should be at the top. These are the items that need to be prioritized above anything else if you are to achieve the outcome. The lesser understood or "nice to have" items should be further down the column.

The final stage is to slice value out of the value map. Using some sticky tape (ideally colored, such as painters' tape), we ask the person who holds overall responsibility for prioritizing work and articulating value (usually this is the **Product Owner** for a team using Scrum) to slice horizontally what they see as a slice of value for the whole product. This means looking at the most important items for each theme and combining them with some of the other highly important items from other themes.

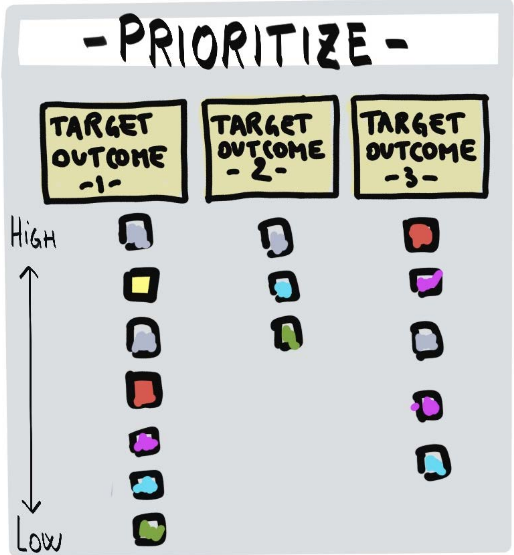


Figure 11.4: Prioritizing work within clusters

At this point, our Product Owner has a huge amount of power. They can prioritize within a given outcome. They can prioritize a whole outcome and move everything down or up. They can combine items together from different outcomes to form proposed releases. They can slice one, two, three, or fifty slices of value – each one containing one, two, or more items. Most importantly, they can facilitate conversations with all stakeholders and team members to arrive at a consensus on this two-dimensional Value Slice Map.

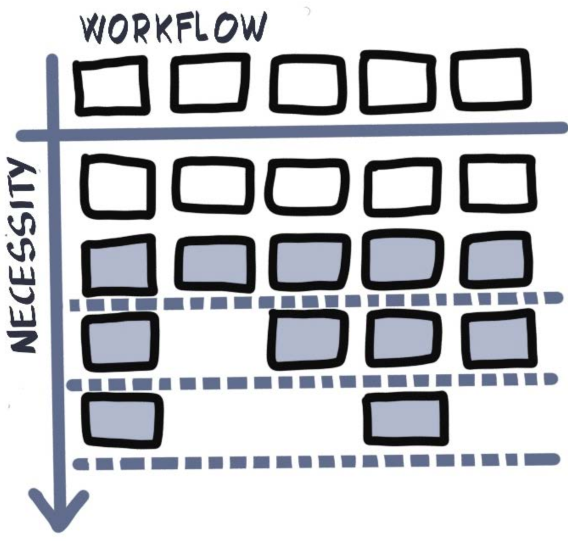


Figure 11.5: Two-dimensional Value Slice Map

During many years of using these practices, we've picked up a few facilitation tips to help explain them correctly. The first involves how you might visualize and plan two valuable activities.

The Beer and the Curry

In 2017, I led an engagement with a global oil company. Toward the end of the first week, the team was tired. It had been a busy, intensive week. We'd formed our team and built our foundation of culture. We'd run several practices of the Discovery Loop, including user Empathy Mapping and Event Storming, which involved lots of standing, lots of thinking, and lots of conversation.

On Thursday afternoon, I was facilitating User Story Mapping and Value Slicing based on all the items that had been captured on the Event Storm and Empathy Maps. This practice was new to the team. After we had gone through the first few steps by putting all the work on the wall and organizing it against the outcomes, I talked about the need to slice and prioritize.

I started by saying, *Obviously, we'd like to do all this work*, after which one of the senior stakeholders interrupted and said, YES! *We need to do all this work*. I could sense there was some discomfort among stakeholders, as if I was doing a typical consultants' effort on locking down scope when the stakeholders wanted everything built. Perhaps my leading choice of words could have been better.

Figure 11.6: Explaining Value Slicing

But I wasn't trying to decide what was in and out of scope. My whole agile mindset is based on flexible scope, the ability to adapt and change scope as we learn more, and always ensuring we're delivering the next most valuable and important work.

To explain my mindset, my thoughts fast-forwarded to a team social we had planned for later that day. It had been a long week and we had planned to go for a few drinks and a curry – again boosting our cultural foundation further by allowing the team to relax and get to know each other a bit better.

I was looking forward to having a beer and having a curry after that beer. In fact, I was really looking forward to that beer. I felt we'd really earned it that week and it was going to be great to raise a glass and say cheers with my new team! But that didn't mean that the curry wasn't important. Nor did it mean that the curry was not going to happen. We were going to have a beer first followed by a curry. That was how we'd prioritized the evening. We hadn't de-scoped anything nor were we planning to. The beer was in my top slice of value. The curry was in my second slice of value.

The team felt more relaxed understanding we were not de-scoping any work at all using this practice but simply organizing by value. The team also felt very relaxed and enjoyed both a beer and a curry!

We've also learned a few simple tricks that can help set up the Value Slicing practice to work effectively.

One to Few to Many Slices of Value – Continuous Delivery

I've learned various tricks over the years of facilitating this exercise.

One of the first times I ran it, we had organized all the work into columns linked to Target Outcomes and we progressed to the slicing part of the practice. I placed one slice of tape on the wall and asked the Product Owner and stakeholders to move the sticky notes they deemed the most valuable above that line of tape and the less valuable ones beneath that line.

As I observed the team working through this process, I realized that the single line of tape had generated a misleading point of this practice. There was a reluctance to put anything beneath the line because there was a perception that this meant **out of scope**. I explained this was not the case and what I was trying to do was slice out the **Minimal Viable Product** or **MVP**. MVP defines the minimum number of features that could form the product that could be released to users to learn from and build upon. In reality, many stakeholders see defining the MVP as something negative as it's where they lose all the great innovative featuresthat they may want but are not collectively deemed important. I actually try to avoid using the term MVP, as it is often greeted with some negative emotion.

I learned from this facilitation that one slice should never be used as we are not defining things as in or out of scope and we are not defining just the MVP.

Working with another customer in Finland, I took this learning and adapted my facilitation approach. With all the items that had been captured from the Discovery Loop on the map, I produced three slices of tape. Hopefully now the Product Owner and stakeholders would not fall into the in-scope/out-of-scope trap. However, now there was a new misunderstanding! For this particular engagement, which was an immersive four-week Open Innovation Labs residency focused on improved operations, we had planned three one-week sprints. By coincidence, I had produced three slices of tape for Value Slicing. So, the stakeholders and Product Owner assumed that whatever we put in the first slice would form the scope for Sprint 1, the second slice would be Sprint 2, and the third slice would be Sprint 3.

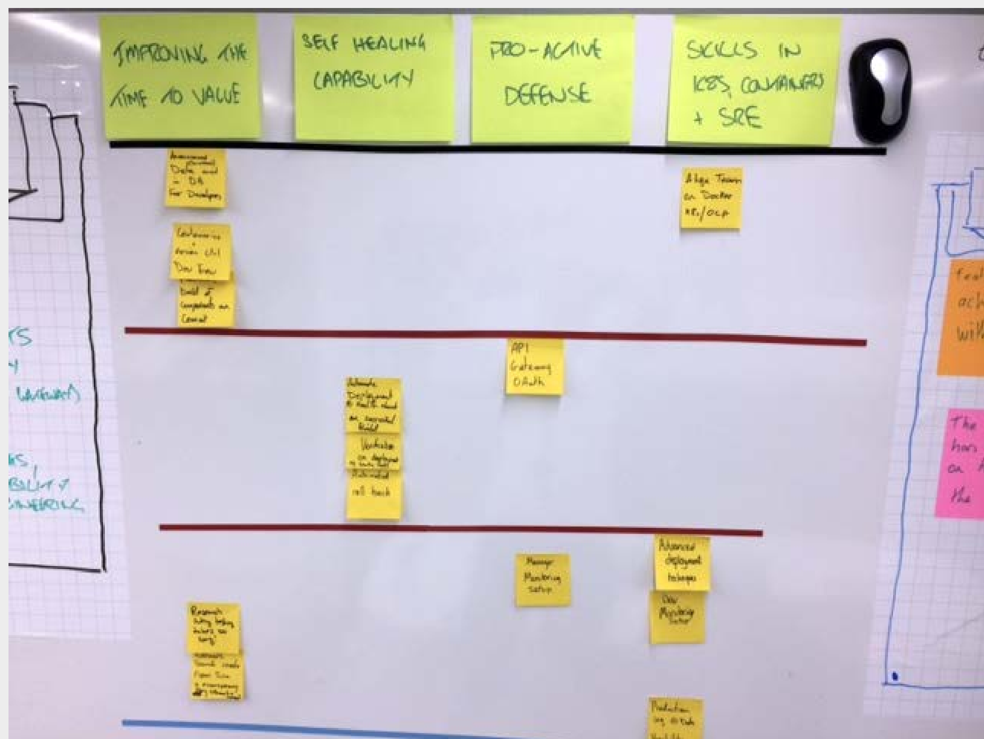


Figure 11.7: Value Slicing of the items captured from the Discovery Loop

I explained that this was not the case. We do not yet know how long it will take the team to deliver each item in each slice. We will use other practices in the Delivery Loop to help us understand that. We could end up delivering more than one slice in one sprint. Or, it may take more than one sprint to deliver one slice. We just don't know yet.

Since then, I have tweaked my facilitation further. When making the slices available, I now produce lots of them – at least 10, sometimes more than 20. I also make the roll of tape accessible and tell the Product Owner to use as many slices as they would like – the more the better, in fact! I've found Value Slice Maps now often have many more slices.

A Product Owner from a UK defensecompany once remarked to me that you could argue that each item on the Value Slice board could be its own slice of value. I celebrated with a massive smile when I heard this. Yes! When we reach that mindset and approach, we truly are reaching the goal of continuous delivery.

Figure 11.8: Value Slicing with many slices

Visualizing and slicing increments of value has evolved from the amazing thinking and work produced by Jeff Patton in his book *User Story Mapping*[1] published in 2008. User Story Mapping is an effective practice for creating lightweight release plans that can drive iterative and incremental delivery practices. We highly recommend reading Patton's book and trying out the exercise he describes in his fifth chapter about visualizing and slicing out the value of something very simple, like everything you do in the morning to get up, get ready, and travel to work. We use this exercise in our enablement workshops and find it really brings the practice to life well.

Let's look at how the PetBattle team approached Value Slicing.

1 https://www.jpattonassociates.com/user-story-mapping/

PetBattle – Slicing Value towards Continuous Delivery

The PetBattle team reviewed all of the artifacts they produced during their first Discovery Loop.

The Impact Map identified "Increasing Site Engagement" for Uploaders as the place they wanted to invest in running experiments and building initial features. The Empathy Map of Mary, their user, added further support to building tournament services and a live leaderboard. The team Event Stormed the idea of Mary entering the daily tournament and winning a prize to break down the event flow to identify commands, read models, some UI ideas, and aggregates. The Metrics-Based Process Map identified some bottlenecks in the existing PetBattle deployment steps, mainly due to a lack of automation. Finally, the team brainstormed all the non-functional considerations they had.

They copied all of the features that had resulted from these onto fresh sticky notes and spread them across the wall.

Then it was time to consider the headings for their Value Slicing Map. The team recalled that they distilled all of the Discovery Loop information and learning into three primary Target Outcomes:

- PetBattle is generating revenue from an increased active user base.
- PetBattle is always online.
- Improved team satisfaction with excitement to build and run PetBattle.

They also identified an additional Enabling Outcome:

- Reduce Operational Incidents with impact to customers.

These four outcomes formed the backbone of the PetBattle Value Slice Map.

Figure 11.9: Target Outcomes backbone

As the team explored these four outcomes further, they thought it might help to break them down a bit further to help with shared understanding with stakeholders. The Impact Map had driven focus on four outcomes:

- Increased participation rate of the casual viewer
- Increased uploads
- Increased site engagement of the uploaders
- Increased number of sponsored competitions

Collectively, these would all help with the first primary outcome where PetBattle would be generating revenue from an increased active user base. So, these were added to the Value Slice Map:

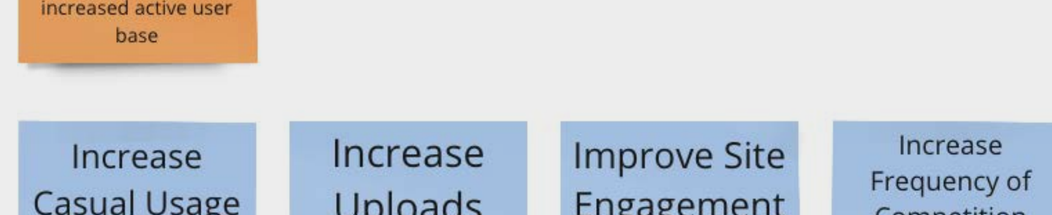


Figure 11.10: First Target Outcome broken down

The second primary outcome was that PetBattle would be online.

The team reflected on the sections of their Non-Functional Map and recognized three outcomes that would help achieve this:

- Improve Site Reliability
- Improve Maintainability and Supportability
- Increase Auditability and Observability

Figure 11.11: Second Target Outcome broken down

As the team discussed the third primary outcome, Improved team satisfaction with excitement to build and run PetBattle, their conversations were all about achieving great testability. Having a foundation of technical practices that would allow them to automate different levels of tests and also user-test, utilizing advanced deployment techniques, would make them very happy. They also reflected on some of the ideas they came up with when they were forming as a team and building their foundation of culture – socials, having breakfast and lunches together, and starting a book club were just a few ideas that would help improve team culture. So, they added these important headings:

Figure 11.12: Third Target Outcome broken down

Finally, they had their Enabling Outcome, whereby reducing operational incidents with impact to customers would help drive all the other outcomes. This could also be broken into three areas:

- Reduce Security Risks
- Improve Reusability
- Enhance Performance and Scalability

Figure 11.13: Fourth Target Outcome broken down

So, they had a huge collection of outputs spread over one wall and an organized set of outcomes as headings on another wall:

Figure 11.14: All Target Outcomes at two levels

It was time to connect the outputs to the outcomes by forming columns beneath each outcome.

They started with the first primary outcome. The outputs sourced here were mainly commands from the Event Storm, supported by focused impacts on the Impact Map and high motivations captured on the Empathy Map.

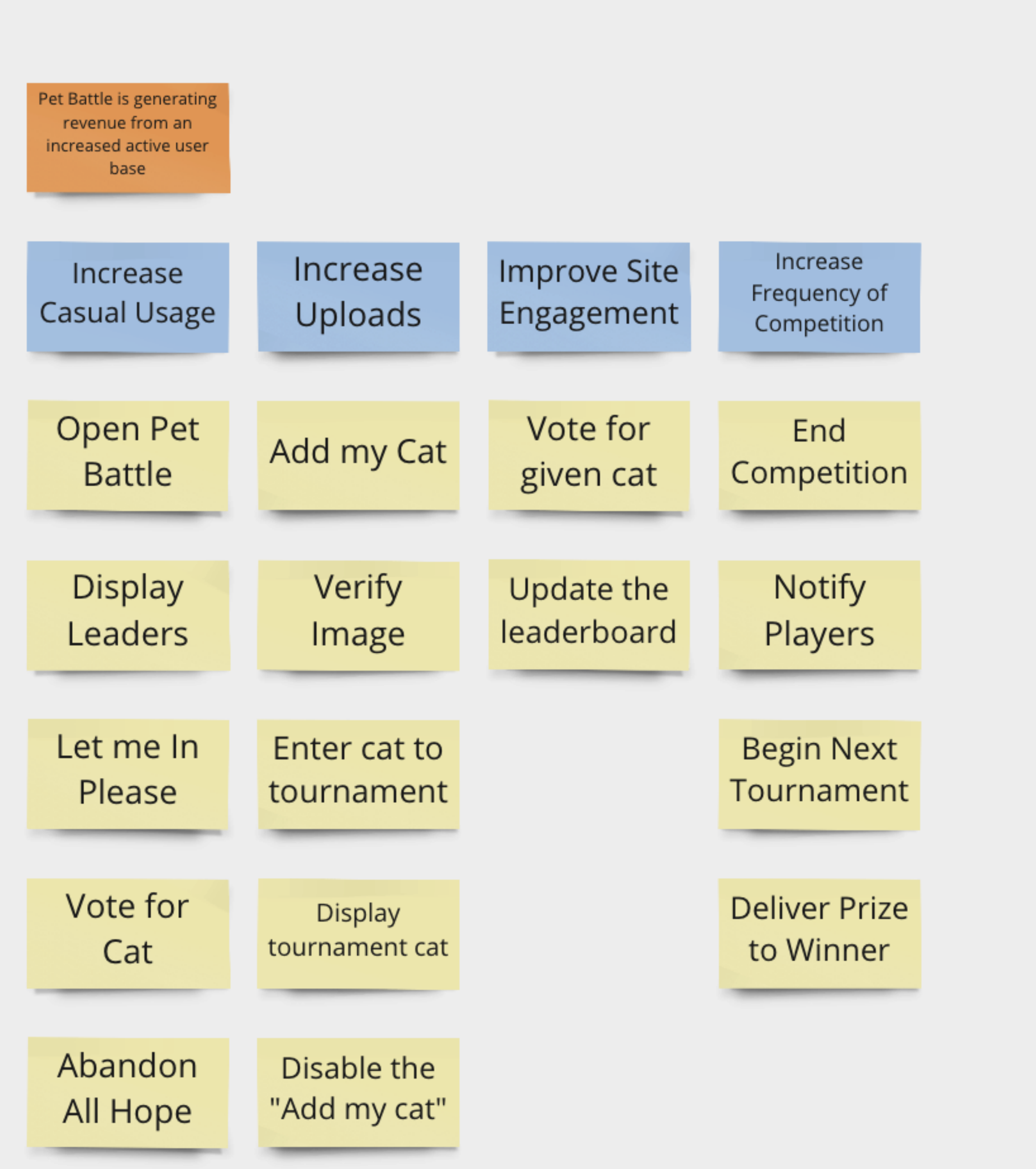


Figure 11.15: Outputs to deliver first Target Outcome

The outputs moved under the second and fourth outcomes were sourced from the MBPM and Non-Functional Map. This was also true for the third outcome, which also included some of the ideas captured during the early social contract and real-time retrospective that was started when building the cultural foundation.

The team ended up with a UserStory Map that showed the initial journey through PetBattle as well as the journey the team would go on to deliver and support it:

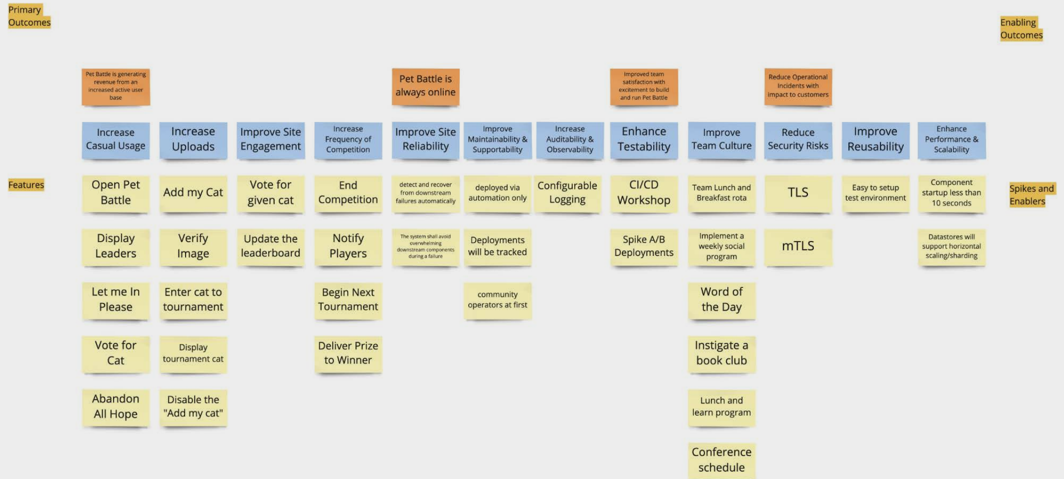


Figure 11.16: PetBattle User Story Map

This is the first information radiator that shows functional features of an application, work to build, operate, and improve the platform and generate an enthusiastic, high-performing team.

The final step is to start slicing valuable increments of the whole engagement. Working with Product Ownership, the team was keen to ensure all outcomes were being tackled early in some minimal form and they would continue to improve every outcome as they delivered more slices.

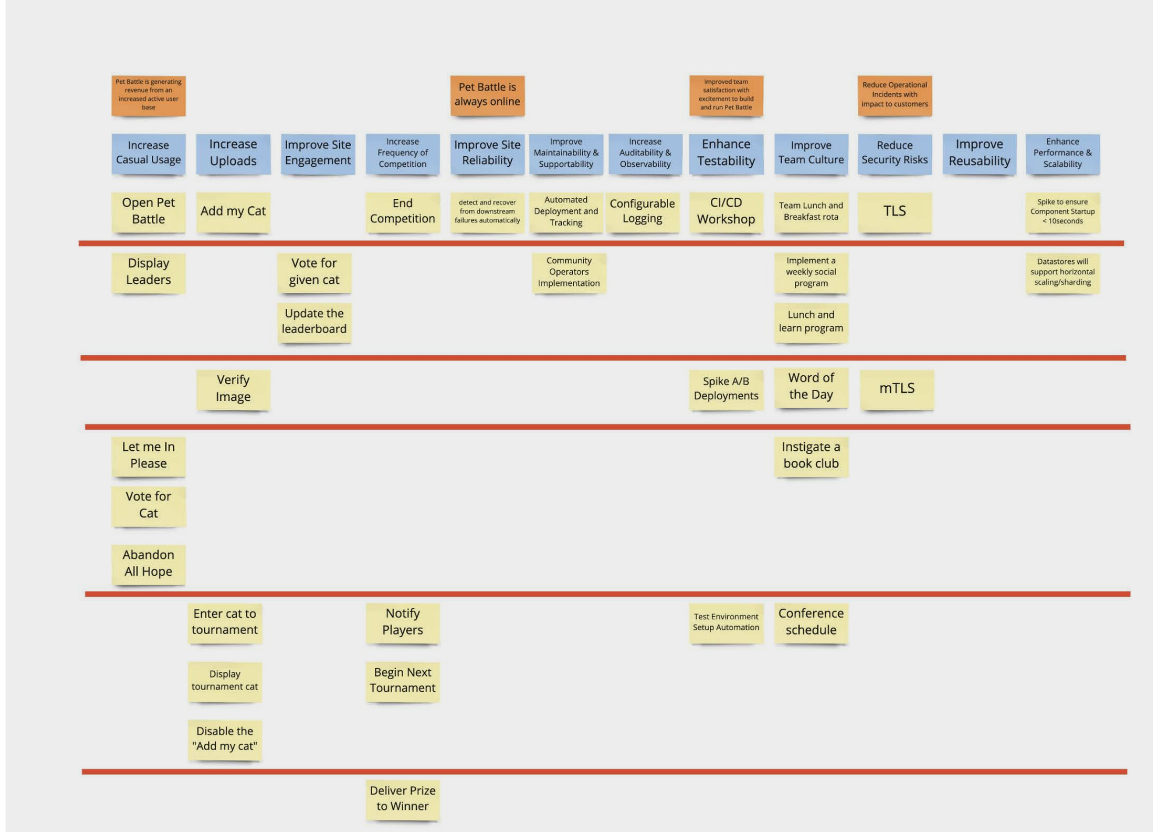


Figure 11.17: Value Slicing the PetBattle User Map

Looking at the top slice of value brought a feeling of excitement. The team could see the first items they were going to do to make the PetBattle vision a reality!

There is a growing set of interesting links, conversations, and further information on these practices in the Open Practice Library at openpracticelibrary.com/practice/user-story-mapping. Take a look and, if you have a story or experience to share, you can help improve the practice further.

Now we've seen the powerful User Story Mapping and Value Slicing technique, we're going to explore a few other practices that will help make this even more successful and collaborative. We often find that people have two challenges with User Story Mapping. First, they don't know how to get everything onto the User Story Map in the first place. Second, the approach to prioritization and slicing out value can be difficult for some and can also lack collaboration.

Let's look at the first challenge first.

When we introduced the User Story Mapping practice, we said we start by copying all of the outputs, the deliverables, the features, and the ideas that had surfaced from the practices used on the Discovery Loop. That sounds very simple and straightforward. In fact, it really just calls for a human copy-and-paste function to replicate all the deliverables captured on the Impact Map, all the commands captured on the Event Storm, and all the ideas and non-functional work captured during discussion of the MBPM.

But is that enough? Are we shutting off the potential for increased innovation by simply relying on the inspiration that happened a few days ago? A slightly different approach is to not just think of User Story Mapping and Value Slicing to be about delivering features. We can try moving to a more experimental mindset where, during the Options Pivot, we really want to design experiments we can run during the Delivery Loop.

Design of Experiments

All our ideas for new products, services, features, and indeed any changes we can introduce to make things better (more growth, increased revenue, enhanced experience, and so on) start off as a hypothesis or an assumption. In a traditional approach to planning, a team may place bets on which experiment to run based on some form of return on investment-style analysis, while making further assumptions in the process.

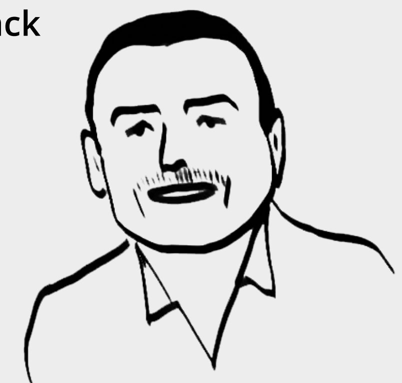
Design of Experiments is an alternative to this approach, in which we try to validate as many of the important ideas/hypotheses/assumptions we are making as early as possible. Some of those objects of the experiments we may want to keep **open** until we get some real-world proof, which can be done through some of the advanced deployment capability (such as A/B Testing) that we'll explore later in this chapter.

Design of Experiments is a practice we use to turn ideas, hypotheses, or assumptions into concrete, well-defined sets of experiments that can be carried out in order to achieve validation or invalidation – that is, provide us with valuable learning.

Design of Experiments is a fail-safe way to advance a solution and learn fast. It can provide a quick way to evolve a product, helps drive innovation in existing as well as new products, and enables autonomous teams to deliver on leadership intent by placing small bets.

You may need more than one experiment for each item (idea, hypothesis, assumption). An experiment usually only changes a small part of the product or service in order to understand how this change could influence our Target Outcomes. The number of experiments is really defined based on what you want to learn and how many distinctive changes you will be introducing.

Qualitative versus Quantitative Feedback



Working on a Labs residency with a road and travel insurance provider in 2018 provided us with an opportunity to design experiments on an early prototype of rebuilding the mobile application with an improved user experience and increased conversation.

We wanted to measure the current application versus some ideas brainstormed with business stakeholders for improved experience, so we designed an experiment for our test users. We advised them to behave as if this was a real-life situation, stop where they would normally stop, read how they would normally read, and so on, when navigating through the application.

The role of this application was to guide the user through the car insurance booking process. To complete this process, they needed their license plate number, social security number, and residential zip code.

Each user was guided to follow a URL (ideally on mobile) for the deployment of the application running on OpenShift. They were instructed to select a car and try to compare and buy what seemed to be the best insurance for the user. The experiment ended at the point where they had purchased insurance. (Note – each user was told that this was a test and that the payment details provided to them were for a test credit card and no monies would be moved.)

The A/B Test meant different approaches for displaying the page could be used with different users so we could test different prototypes in user interviews.

Quantitative data from the experiment showed the duration of the journey through the application, drop-off rates, and completion rates.

Qualitative data from the associated users highlighted pain points in the user experience, where there remained some confusion, and validated some of the positive experiences.

The qualitative and quantitative feedback combined provided confirmation of which approach was the most suitable. This meant the product team could confidently code the "best" approach as validated by data.

This process, from end to end, took one week.

The format of the experiment documentation is really as important as the content. It is the content that tells you how well you have designed the experiment; for example, does the experimental design allow for too many opportunities where the outcome may be ambiguous?

Good experiments need the following minimum details to be successful:

- **Hypothesis**: Formulated as a sentence, often expressing an assumption.
- **Current condition**: What is the situation now (as measurable as possible)?
- **Target condition**: What are we trying to achieve (as measurable as possible)?
- **Obstacles**: What could prevent us from achieving the target condition? What could cause interference or noise?
- **Pass**: How can we define a positive pass? If the target condition may not always be achieved, then what do we consider a significant enough change to conclude the experiment is confirming the hypothesis, that is, passing with a positive outcome?
- **Measures**: How can we measure the progress?
- **Learning**: Always capture outcomes and learning, which should ideally lead to more experiments of higher order.

Once described, the experiments can be implemented, tracked, and measured in order to analyze the outcomes. In an ideal world, an experiment will have binary success/failure criteria, but most often we need to analyze data using statistical methods to find out if there is a significant correlation between the change introduced with the experiment and the change in the Target Outcome.

> **NOTE**
>
> Successful experiments are not experiments that have proven our assumption is correct. Successful experiments are those that provide valid and reliable data that shows a statistically significant conclusion.

Design of Experiments is nothing without Discovery or Delivery, which is the main reason for combining this practice with others. Experiments are sourced from the information collected during practices on the Discovery Loop. Hypotheses are formed during Impact Mapping. Assumptions are noted during Event Storming, Empathy Mapping, and other human-centered design practices. Ideas are captured in all Discovery practices.

Experiments need to be prioritized as we can only do so much in the time we have. Combining this practice with the various prioritization matrices, such as **Impact-Effort Prioritization** or **How-Now-Wow Prioritization**, or even economic prioritization practices such as **Weighted-Shortest-Job-First**, helps a lot. We are about to explore each of these in the next part of this chapter.

Experiments are often realized first through rapid prototyping, which requires user research and user testing, which we do on the Delivery Loop. This combination provides for fast learning even before a single line of code is written.

Experiments can be run in production as well. In fact, tests in production are the ultimate form of validation of ideas/hypotheses/assumptions as the validation is supported by real data and real customer actions and behavior. The A/B Testing practice provides a very valuable combination.

Often, you may have a set of experiments go through a sequence of Rapid Prototyping/Prototyping with User Research, and then a subset of successful experiments would be carried forward to production to pass through A/B Testing. There are other mechanisms of controlling deployments that will enable measuring and learning from real customer behavior – we'll introduce all of these later in this chapter.

You can read and discuss this practice further at openpracticelibrary.com/practice/design-of-experiments.

Designed experiments should end up on a User Story Map and Value Slice Map so that they can be prioritized against all other work.

Let's look at a couple of other tools that can help with the prioritization discussions, starting with the Impact and Effort Prioritization Matrix.

Impact and Effort Prioritization Matrix

The Impact and Effort Prioritization Matrix is a decision-making/prioritization practice for the selection of ideas (such as functional feature ideas, performance ideas, other non-functional ideas, platform growth ideas, and so on).

This practice opens up product development for the whole team (which really understands effort) and connects them to stakeholders (who really understand impact). Developing new products goes hand in hand with the generation of ideas, hypotheses, and their testing/validation. Unfortunately, it is mostly impossible to test and evaluate all the ideas and hypotheses we can come up with. This requires us to filter and prioritize which of them to work on.

This matrix is simple, easy to understand, and very visual, and can include the whole team in the process of transparent selection of ideas and hypotheses to work on first. It also helps Product Owners and Product Managers build the product roadmaps and Product Backlogs and explains priorities to stakeholders.

This practice is very powerful in helping to identify direction and ideas for pivoting purely from the visualization.

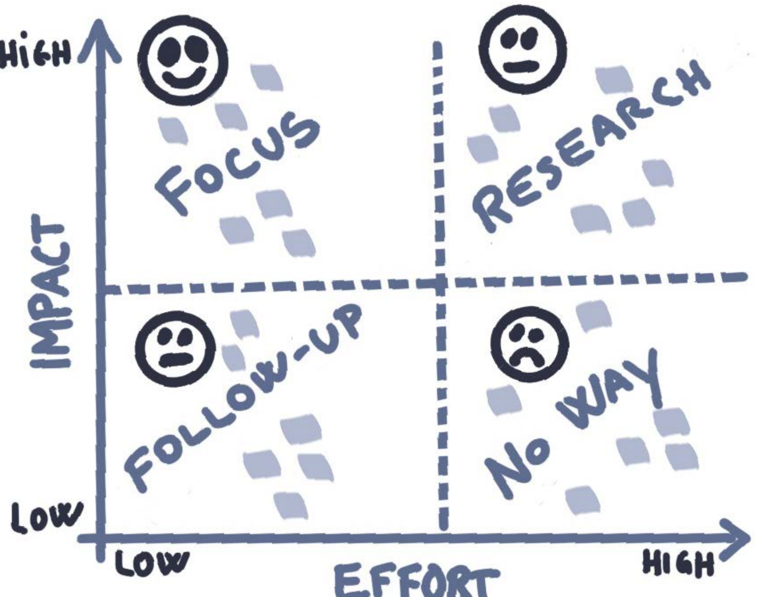


Figure 11.18: Two-by-two matrix comparing Impact versus Effort

Four separate groups of ideas have emerged from this practice:

- **The Best Ideas to Focus on**: High Impact / Low Effort – These should be considered in the higher slices of the Value Slice Map.
- **Research Required**: High Impact / High Effort – These should be considered in the higher slices of the Value Slice Map but, in later Product Backlog Refinement practices, may be deemed lower priority given it will take longer to realize the value.
- **Follow Up**: Low Impact / Low Effort – These should be low down on the Value Slice Map and either followed up if time permits or removed completely.
- **No Way – Bad Ideas**: Low Impact / High Effort – These should be low down on the Value Slice Map or removed completely.

The sources of ideas and hypotheses are all the Discovery practices, such as Event Storming, Impact Mapping, and Empathy Mapping. While we perform those aforementioned practices, often ideas may emerge for possible improvements or new hypotheses may form.

Before adding any of them as items to the Product Backlog, these ideas and hypotheses would typically need some research, analysis, and further elaboration.

Once prioritized, these ideas and hypothesis may lead to:

- New features being added through the User Story Map and Value Slice board
- Complete new features being broken down into smaller features or User Stories and refactored on the Value Slice board
- User Research
- Design of Experiments
- Technical Spikes and UI Prototypes

The Impact and Effort Prioritization matrix has its own Open Practice Library page at openpracticelibrary.com/practice/impact-effort-prioritization-matrix/ – a great place to continue the learning and discussion about this prioritization practice.

A slightly different perspective on prioritization is achieved using the How-Now-Wow Prioritization practice. Whereas the previous practice is used to filter out and prioritize the very high-impacting features, this practice is used to identify and prioritize the quick wins and base features needed for a product.

How-Now-Wow Prioritization

How-Now-Wow is an idea selection tool that is often combined with Brainstorming, **How-Might-We**[2] (**HMW**), and Design of Experiments. It compares and plots ideas on a 2x2 matrix by comparing the idea's difficulty to implement with its novelty/originality.

Similar to the Impact and Effort Prioritization Matrix, How-Now-Wow Prioritization is simple, easy to understand, and very visual, and can include the whole team in the process of transparent selection of ideas/hypotheses to work on first.

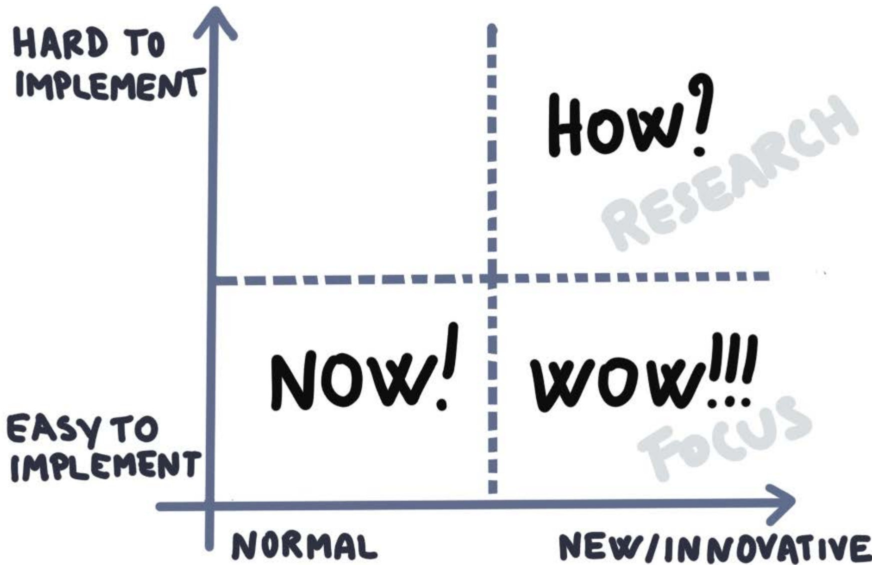


Figure 11.19: How-Now-Wow Prioritization map

Again, the sources of ideas and hypotheses are all the Discovery practices, such as Event Storming, Impact Mapping, HMW, and Empathy Mapping. When we perform those aforementioned practices, often ideas will emerge for possible improvements or new hypotheses may form.

We can plot each of these on the How-Now-Wow matrix by assessing each item and considering how easy or difficult it is to implement (using team members to collaborate and align on this) and how new and innovative the feature is.

2 https://openpracticelibrary.com/practice/hmw/

Three separate groups of ideas have emerged from this practice. There are three we're particularly interested in:

1. **Now Ideas**: Easy to implement and considered normal ideas for the product. These should be considered in the higher slices of the Value Slice Map and are ideas we would expect to deliver.
2. **Wow Ideas**: Easy to implement and considered highly innovative or new for the product. These should be considered as potential ideas for the higher slices of the Value Slice Map and would be particularly valuable if innovation and market differentiation were deemed high-priority focus areas. If the Priority Sliders practice has already been used, it may provide some direction here.
3. **How Ideas**: Hard to implement and considered highly innovative or new for the product. These would benefit from further research to understand the implementation difficulty and potential impact further. Design of Experiments, prototyping, and further User Research will help validate whether this innovation is something that would be well received. Technical Spikes and research will help establish confidence and potentially easier solutions to implement.
4. **Other ideas**: Hard to implement and not particularly innovative or new for the product. We're not interested in these ideas at all.

Once placed on the How-Now-Wow matrix, these ideas and hypotheses may lead to:

- New features being added through the User Story Map and Value Slice board
- Complete new features being broken down into smaller features or User Stories and refactored on the Value Slice board
- User Research
- Design of Experiments
- Technical Spikes and UI Prototypes

For more information on the How-Now-Wow Prioritization practice and to start a conversation about how to best use it, have a look at openpracticelibrary.com/practice/how-now-wow-prioritization-matrix.

Our primary motivation behind using practices such as Impact and Effort Prioritization and How-Now-Wow Prioritization is to facilitate conversation. Practices that get business folks talking to each other and aligning on an approach with some shared understandings are great. Practices that get techie folks collaborating and gaining a common understanding of the implementation approach and complexity are also great. Practices that get business people and techie people all collaborating to reach a consensus and a common view of both the business context and implementation approach are amazing. These two practices are examples of those.

Figure 11.20: Different stakeholders collaborating to gain a better understanding of the product

Both practices highlighted some features to do more research on. Practices categorized in the How quadrant of the How-Now-Wow matrix will benefit from additional research. Practices categorized in the High Effort / High Impact quadrant of the Impact and Effort Prioritization matrix will benefit from additional research.

Many of the human-centered design practices outlined in *Chapter 8*, *Discovering the Why and Who*, will help with this research. This includes Empathy Mapping, qualitative user research, conceptual design, prototyping, and interaction design. If the feature area is of very high importance, it may be valuable to invest in a specific practice that will really further the understanding of the feature – the Design Sprint.

The Design Sprint

The Design Sprint has become a popular practice to support product research. It is a five-day customer-centric process for rapidly solving a key challenge, creating new products, or improving existing ones. Design Sprints enable you to:

- Clarify the problem at hand and identify the needs of potential users.
- Explore solutions through brainstorming and sketching exercises.
- Distill your ideas into one or two solutions that you can test.
- Prototype your solution and bring it to life.
- Test the prototype with people who would use it.

The process phases include Understand, Define, Sketch, Decide, Prototype, and Validate.

The aim is to fast-forward into the future to see your finished product and customer reactions, before making any expensive commitments. It is a simple and cheap way to validate major assumptions and the big question(s) and point to the different options to explore further through delivery. This set of practices reduces risks when bringing a new product, service, or feature to the market. Design Sprints are the fastest way to find out if a product or project is worth undertaking, if a feature is worth the effort, or if your value proposition is really valid. For the latter, you should also consider running a Research Sprint.[3] It compresses work into one week and most importantly tests the design idea and provides real user feedback in a rapid fashion.

By now, there are many different variations of the Design Sprint format. You may come across the Google Ventures variation – the Design Sprint 2.0 – which is the agenda shown below. The best thing to do is to try different variations and judge which one works for what context.

3 https://library.gv.com/the-gv-research-sprint-a-4-day-process-for-answering-important-startup-questions-97279b532b25

Monday	Tuesday	Wednesday	Thursday	Friday
Intro	Decide on Solution	Prototype Creation	Testing with End Users	Showcase (demonstrate process and findings)
How Might we Map	Storyboard Design	User Test Preparation	Analyze Results and Prepare Findings	
Lightning Demos	Decider Vote	Interview Room Setup		
Taking Notes	Storyboard			
Crazy 8's	Final Walk-through			
Long-Term Goals	User Test Recruiting			
Sprint Questions				
Lightning Demos				
Drawing Ideas				
Start User Test Recruiting				
Develop Concepts				

Table 11.1: A five-day Design Sprint

Effectively, we are using the same Mobius Loop mental model as used throughout this book but micro-focused on a particular option in order to refine understanding and conduct further research about its value. That improved understanding of relative value then filters back into the overall Mobius Loop that classified this as an option worthy of a Design Sprint.

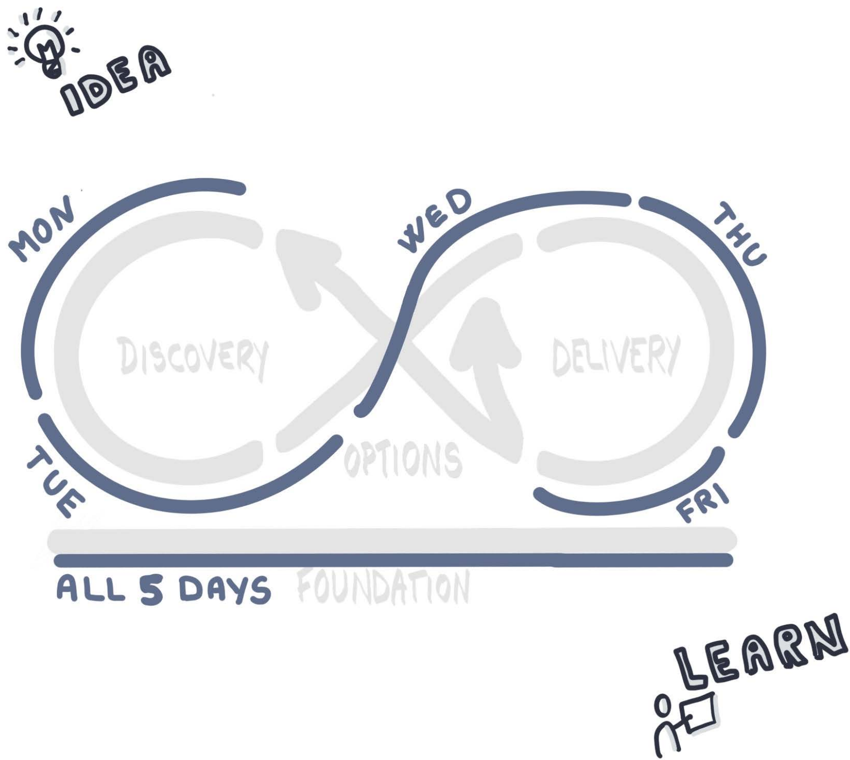


Figure 11.21: The Design Sprint – quick trip round the Mobius Loop

A Design Sprint will really help refine the shared understanding of the value a feature or group of features may offer in a product. They may be functional user features. They may also be non-functional features that are more focused on improving the platform and improving the development experience. The same agenda as above can apply where "users" are developers or operators and the Design Sprint is focused on researching some potential work that will improve their development or operations experience.

This practice will help elaborate and refine the information that is pushed through the User Story Map and Value Slicing Map. We see it as a practice on the Options Pivot because it will help decide whether or not to proceed with the delivery of the associated features.

Read more about the practice, add your own experiences, or raise any questions you might have at openpracticelibrary.com/practice/design-sprint/.

The User Story Mapping practice helps us visualize our work into a story with a clear backbone. Value Slicing allows us to form incremental release plans that can be delivered in iterations. The Impact and Effort Prioritization and How-Now-Wow Prioritization practices help provide alternate perspectives to help with the Value Slicing. The Design Sprint allows us to dive deeper into a specific feature area to research it further, so we can prioritize with increased confidence.

All of these practices (and many others you'll find in the Open Practice Library) are homing in on us being able to produce an initial **Product Backlog** – a single, one-dimensional list of stuff we're going to take into the Delivery Loop.

Let's now look at how we translate the information from our Value Slices into a Product Backlog and how we can continue to prioritize it.

Forming the Initial Product Backlog

Here's some good news. Forming the Product Backlog is really, really easy. If you've run some Discovery Loop practices and then run User Story Mapping and Value Slicing of the resulting learning, all the hard thinking, collaboration, and alignment has been done.

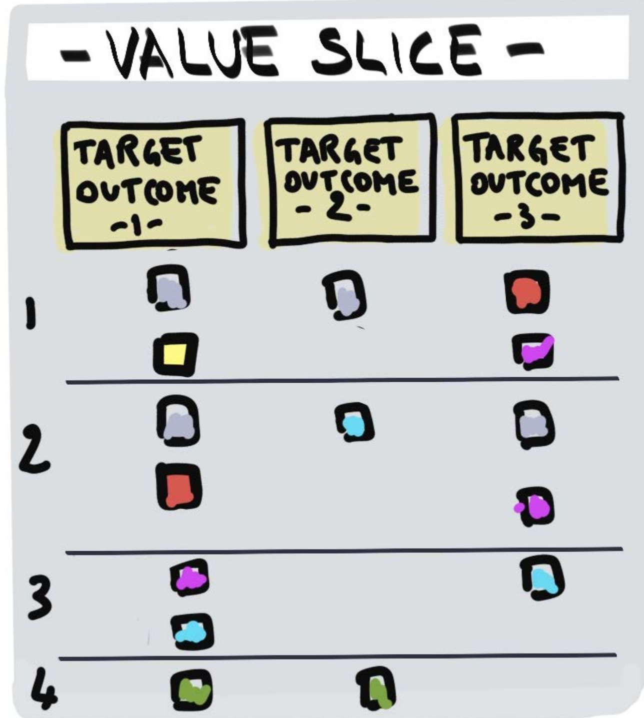


Figure 11.22: Value Slices to drive the initial Product Backlog

This has been a great blend of practices and, if there was strong collaboration and a sense of alignment throughout (which is facilitated by having a strong foundation of open culture), the Value Slice Map should represent a combined, shared view of work and how it can be incrementally released.

To create the Product Backlog, we simply copy each sticky note in the top slice from left to right and place them in a single column.

Figure 11.23: The initial Product Backlog

The sticky note on the left of the top slice will be copied and placed as the item at the top of the Product Backlog. The sticky note to the right of it on the top slice will be the second item on the Product Backlog. Once we've copied all items in the top slice, we move to the second slice of value and, again, copy each of the items from left to right onto the Product Backlog.

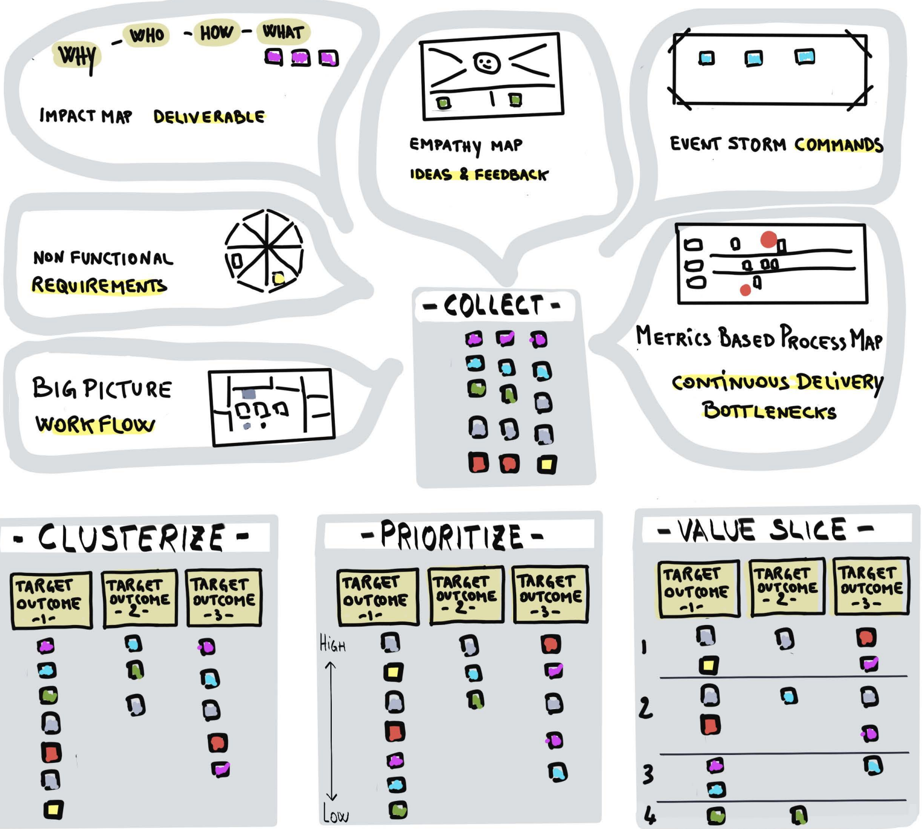


Figure 11.24: Summary of the Value Slicing practice

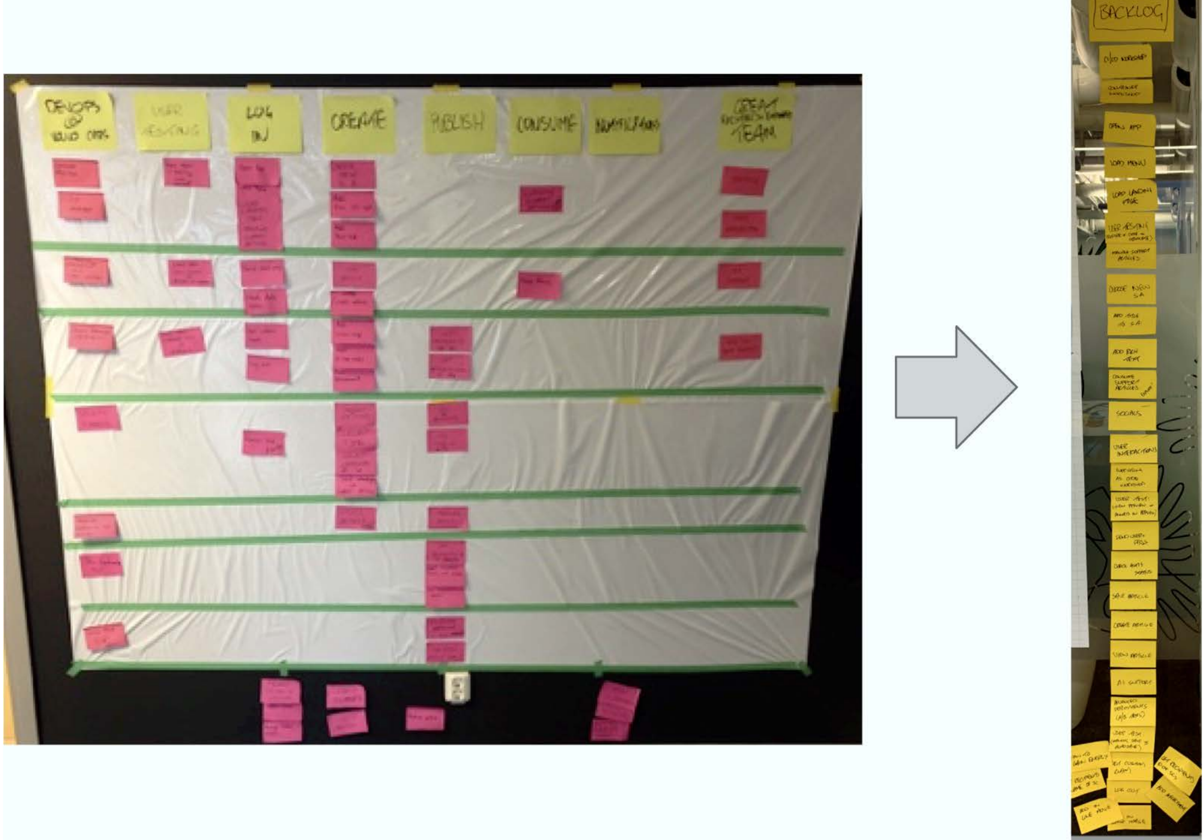


Figure 11.25: Creating a Product Backlog from the Value Slicing Canvas

We end up with a single column of Product Backlog items that have been sourced and prioritized through a collection of robust practices. That traceability is important because we can trace back to the Discovery Loop practice that generated the idea and the value it is intended to deliver.

Let's look at that traceability in action with our PetBattle organization.

PetBattle — Tracing Value through Discovery and Delivery Practices

We saw earlier in this chapter how slices of value were created from the User Story Map. We also saw how the User Story Map was built up entirely from the learning captured through Discovery Loop practices.

Translating this to a Product Backlog is easy.

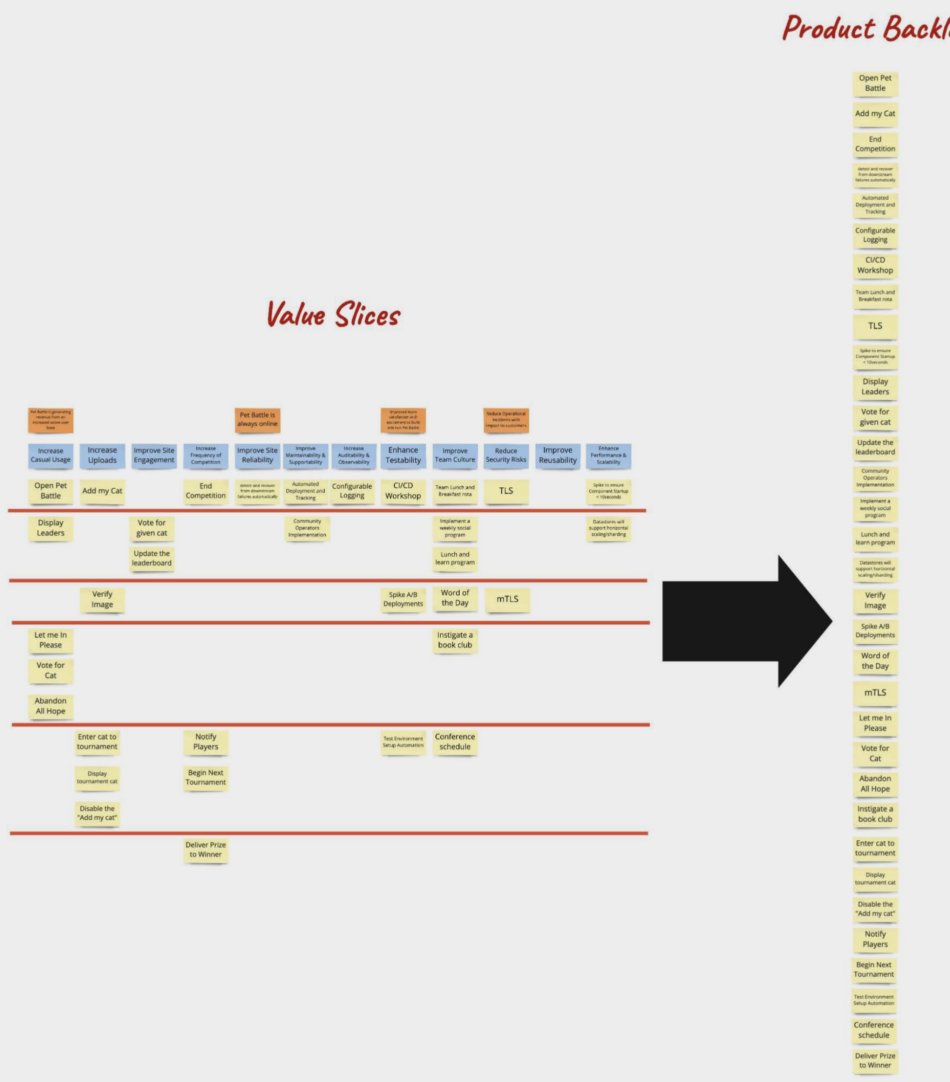


Figure 11.26: Transforming Value Slices into a Product Backlog

This is the beginning of the life of the PetBattle Product Backlog. It will remain living, breathing, and always ready for updates as long as the PetBattle product is in operation.

In fact, the team immediately sees some early prioritization needed and recommends moving the CI/CD workshop and team lunch/breakfast items to the top. They all agreed there was no point writing any code or building any features until they had CI/CD in place and a fed and watered team!

The Product Backlog is a living, breathing artifact. It should never be static. It should never be done. It is a tool that is always available for teams and stakeholders to reference and, in collaboration with Product Owners, a place to add ideas, elaborate on existing ideas, and continue to prioritize work items relative to each other.

From this moment onward, we will start and continue the practice of Product Backlog Refinement.

Product Backlog Refinement

The Product Backlog Refinement practice sits at the heart of the Mobius Loop. We use it coming out of a Discovery Loop. We use it coming out of a Delivery Loop. We use it all the time.

It is perhaps the one practice that sits on the Mobius Loop that does not have a suggested or directed time-box to execute it in. There is also not a suggested or directed number of times you would run it or when you would run it. Product Backlog Refinement should occur as often or as little as needed to get to a backlog that stakeholders and team members have confidence in. You can even use the Confidence Voting practice (as introduced in *Chapter 4*, *Open Culture*) to measure this.

There is also no defined agenda for a Product Backlog Refinement session and it can be attended by a variety of different people, such as development and operational team members, business stakeholders, and leadership. The activities that take place in Product Backlog Refinement include:

- Talking through and refining the collective shared understanding of an item on the backlog, its value to users, and the implementation needs that need to be satisfied
- Re-writing and refining the title of a Product Backlog item to better reflect the collective understanding
- Writing acceptance criteria for a specific item on the Product Backlog
- Doing some relative estimation of the effort required to deliver a feature from the backlog to satisfy the acceptance criteria
- Splitting an item on the backlog into two (or more) smaller items
- Grouping items together into a more standalone item that will deliver a stronger unit of value
- Capturing new ideas and feedback on the Product Backlog
- Prioritizing and re-ordering items on the Product Backlog

All of the artifacts we've generated on the Discovery Loop and Options Pivot are useful to look at, collaborate on, and refine further when performing Product Backlog Refinement. They too are all living, breathing artifacts and, often, conversations during Product Backlog Refinement trigger further updates to these. So, for example, we may add a new deliverable to our Impact Map and connect it to an impact and actor to test with. We may elaborate on some details on the Event Storm as we start to consider the implementation details of an associated backlog item. As new items are captured from Product Backlog Refinement, the Impact and Effort Prioritization Matrix, How-Now-Wow Prioritization Matrix, and Value Slice board artifacts are all available to relatively plot the new item against existing items. In *Chapter 17*, *Improve It*, we'll return to the Options Pivot following an iteration of the Delivery Loop and look at how the measurements and learning captured from delivery can drive further Product Backlog Refinement.

Arguably one of the most important aspects of Product Backlog Refinement is prioritization and, in particular, prioritizing what is toward the top of the Product Backlog. This is what the team will **pull** from when planning their next iteration of the Delivery Loop. So, it's important that the items at the very top of the backlog truly reflect what is most valuable and help generate the outcomes that matter.

For more details on Product Backlog Refinement and to converse with the community, take a look at the Open Practice Library page at openpracticelibrary.com/practice/backlog-refinement.

We've already seen a few tools that help with initial Product Backlog generation and giving the first set of priorities. Let's look at a few more that will help with ongoing Product Backlog prioritization.

Prioritization

Throughout this chapter, we've used the terms **features** and **Product Backlog items** to explain the different units of work that we capture through Discovery and prioritize and decide which to work on first in the Options Pivot. An important clarification that's needed is that this does not just mean functional features. We are not just deciding which shiny new feature the end users are going to get next. We need to balance customer value against risk mitigation; we need to balance functional against non-functional work. We do that by balancing research, experimentation, and implementation.

Value versus Risk

When we prioritize Product Backlog items, we are relatively assessing all options available to us. That does include new features we're going to implement. It also includes defects and problems in production that need to be fixed. It includes non-functional improvements to the architecture to make future development and operations simpler and stronger. It includes the experiments we might want to execute or further research we want to do in the form of a user interface prototype or a technical spike. It's really anything that will consume the time of some of the cross-functional product team members.

When we prioritize, we need to think about the relative value delivering the item will bring as compared to the risk it might mitigate through acquiring additional learning and confidence. There are different kinds of risk, including:

- **Business Risk:** Are we building the right thing?
- **Technical Risk:** Will this thing work on the platform and will it scale?
- **Cost and Schedule Risk:** Will we deliver within the right time-box to meet market needs? Will we be able to meet any cost constraints?

Running Technical Spikes and proving some of the non-functional aspects of the platform early can provide the knowledge and confidence value, which can be equally, if not more, important than customer value achieved from delivering functional features. In fact, this non-functional work helps us achieve the Enabling Outcomes outlined in *Chapter 10, Setting Outcomes*, whereas the functional implementations are more focused on achieving the primary outcomes.

Let's look at an economic prioritization model that can help us quantify risk, knowledge value, and customer value. It can be used by a Product Owner in collaboration with wider groups of team members and stakeholders and presented to the wider organization.

Cost of Delay and WSJF

Weighted Shortest Job First (**WSJF**) is an economic prioritization model. It is a very popular practice in the **Scaled Agile Framework** (**SAFe**), but works very well as a standalone practice with any size of product or organization.

Many organizations struggle to prioritize risk mitigation action, learning initiatives, or delivery value using any scientific approach. Instead, prioritization comes down to meetings in a room where the **Loudest Voice Dominates** (**LVD**) and/or decisions are made by the **Highest Paid Person's Opinion** (**HIPPO**).

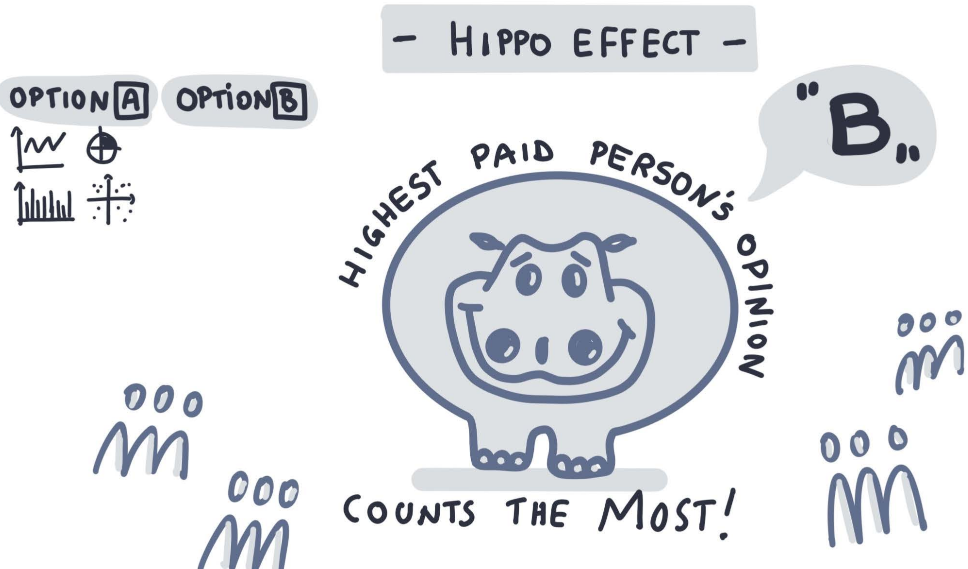


Figure 11.27: Decisions are made based on the HIPPO

So, what is WSJF? It is based on Don Reinertsen's research on the Cost of Delay and the subject of his book *The Principles of Product Development Flow – Second Generation Lean Product Development*. Reinertsen famously said, *If you quantify one thing, quantify the cost of delay*. Josh Arnold explains how the Cost of *Delay is calculated by assessing the impact of not having something when you need it. As a typical example this might be the cost incurred while waiting to deliver a solution that improves efficiency. It is the opportunity cost between having the same thing now, or getting it later.*[4]

4 Source: Mark Richards, SAFe City Simulation Version 2.0

The core thinking behind the Cost of Delay is value foregone over time. For every day we don't have an item in the market, what is it costing the organization? If the value of the item is a cost-saving initiative, how much money is the organization not saving by not implementing this feature? If the value of the item is revenue-related, what is the additional revenue they're missing out on by not implementing it?

The Cost of Delay can be sensitive to time. There are seasonal influences – for example, shipping in retail can be very time-sensitive around, say, the holiday season. Changes may be needed for legislation and compliance. The cost can be very high if something is not delivered by a certain date when new legislation kicks in. The Cost of Delay will be nothing in advance of this date and very high after this date.

There are three primary components that contribute to the Cost of Delay:

- **Direct business value** either to the customer and/or the organization. This reflects preferences users might have that will drive up their customer satisfaction. It will also reflect relative financial reward or cost reduction that the item is expected to drive.
- **Time criticality** to implementing the solution now or at a later date. This incorporates any seasonal or regulation factors that might drive time criticality, as well as whether customers are likely to wait for solutions or if there is a compelling need for it now.
- **Risk reduction and opportunity enablement** is the indirect business value this might bring to the organization. It considers the hidden benefits this might bring in the future as well as reducing the risk profile.

Using Cost of Delay to prioritize work in agile backlogs will result in items being prioritized by value and sensitivity to time. It also allows us to have a lens on direct business value (such as new functional feature development) and indirect business value (such as non-functional improvements to the OpenShift platform).

Cost of Delay =

Business Value + Timing Value + Risk Reduction/Opportunity Enablement Value

WSJF adds a further dimension to this by considering the cost of implementation. Reinertsen said *it is critical to remember that we block a resource whenever we service a job. The benefit of giving immediate service to any job is its cost-of-delay savings, and the cost is the amount of time (duration) we block the resources. Both cost and benefit must enter into an economically correct sequencing.*[5]

Weighted Shortest Job First (WSJF) = Cost of Delay (COD) / Duration

5 The Principles of Product Development Flow – Second Generation Lean Product Development by Donald G Reinertsen

What unit do we use for the three components in Cost of Delay and Duration? It's arbitrary. The actual numbers are meaningless by themselves. The agile practice we use to support COD and WSJF is Relative Estimation,[6] whereby we are relatively assessing the magnitude of business value, timing value, and risk reduction/opportunity enablement for each item on the Product Backlog relative to each other item. This allows us to prioritize the Product Backlog according to WSJF.

We've now introduced several practices on this first trip to the Options Pivot that help us generate more ideas from discovery, refine them, prioritize them, and, ultimately, decide which options we're going to take into a Delivery Loop next. But who makes this decision? The term **we** has been used a lot in this chapter, emphasizing the importance of collaboration. But what happens when we don't have a consensus? Who gets the final say? This is where the importance of great production ownership comes in.

PetBattle – Prioritizing using WSJF

The PetBattle team gathered to conduct a WSJF session on all of the features and work they'd captured on the Value Slice board.

They chose to use this practice in addition to the Impact and Effort Prioritization and Value Slicing because they felt they needed a way to quantify both risk reduction and timeliness of upcoming work. As they had several Enabling Outcomes that are more non-functional based, the team felt that the Cost of Delay and WSJF would allow them to correctly articulate the value of this work relative to functional features.

For each item, the team would spend one minute talking about their understanding of the feature. Each team member would then write down four values – business value, time criticality, risk and opportunity enablement, and duration. The first three values would be given a rating of 1 to 10. The duration used a modified Fibonacci sequence[7] and could either be 1, 2, 3, 5, 8, or 13.

6 https://openpracticelibrary.com/practice/relative-estimation/

7 https://www.mountaingoatsoftware.com/blog/why-the-fibonacci-sequence-works-well-for-estimating

The team members would reveal individual scores to each other and a conversation would follow to converge and align on the team's assessment for each score.

This resulted in a Cost of Delay value and a WSJF value for each item.

	Business Value	Timing Value	Risk Reduction/ Opp Enablement	Cost of Delay	Duration	WSJF
Display Leaders	8	5	0	13	5	2.60
Vote for Cat	10	10	0	20	3	6.67
Open Pet Battle	10	10	0	20	3	6.67
Let me in please	10	10	0	20	5	4.00
Verify Image	2	6	0	8	8	1.00
Add my Cat	10	10	0	20	3	6.67
Display Tournment Cat	9	7	0	16	5	3.20
Disable "Add my Cat"	4	7	0	11	2	5.50
Notify Players	7	8	0	15	2	7.50
Vote for Given Cat	10	10	0	20	2	10.00
Enter Cat into Tournament	10	8	0	18	3	6.00
Deliver Prize to Winner	2	3	2	7	1	7.00
Begin Next Tournament	5	5	0	10	1	10.00
Update the Leaderboard	7	8	0	15	2	7.50
End Comptition	5	2	0	7	1	7.00
Detect and recover from downstream failures a	3	4	8	15	5	3.00
Automated Deployment and Tracking	8	8	10	26	5	5.20
Community Operators	3	4	10	17	5	3.40
Configurable Logging	1	4	8	13	3	4.33
A/B Deployments	8	5	4	17	5	3.40
CI/CD	5	5	9	19	3	6.33
Word of the Day Fun	1	2	3	6	1	6.00
Book Club	7	6	7	20	2	10.00
Weekly Social Program	6	5	8	19	2	9.50
Team Lunch and Breakfast	6	5	8	19	2	9.50
TLS	2	4	10	16	5	3.20
Ensure component startup performance	3	4	7	14	3	4.67
Data stores support horizonatal scaling/sharing	3	4	7	14	5	2.80

Table 11.2: Calculating WSJF for PetBattle work

Some of the conclusions drawn out during the conversation included:

- Most of the functional features were of high business value and of high time criticality to launch the minimal version of this application.
- The non-functional work driven by Enabling Outcomes was deemed of high value for risk reduction and of lower (but some) business value.
- Some non-functional work was clearly more time-critical than others, such as having automated deployment.
- Most work was relatively similar in size with the exception of the Verify Image feature, which has some uncertainty around the solution.

The team agreed this had been a useful Product Backlog Refinement exercise and would help with overall prioritization.

The previous sections on forming the product backlog, refining it and prioritization are all key responsibilities of Product Ownership which we will now explore further.

Product Ownership

Everything in this chapter is about Product Ownership. Everything in the previous chapters about Discovery is Product Ownership. Prioritizing early efforts to build a foundation of open culture, open leadership, and open technology practices requires strong Product Ownership from the outset.

There are whole books and training courses written about Product Ownership, Product Owners, and Product Managers. Much of our thinking has been inspired by the amazing work of Henrik Kniberg. If you have not seen his 15-minute video on YouTube entitled *Product Ownership in a Nutshell*,[8] please put this book down, go and get a cup of tea, and watch the video now. Maybe even watch it two or three times. We, the four authors of this book, reckon we've collectively seen this video over 500 times now!

8 https://www.youtube.com/watch?v=502ILHjX9EE

Some say it is the best 15-minute video on the internet, which is quite an accolade! It packs in so many important philosophies around Product Ownership in such a short amount of time. We tend to show this video during our DevOps Culture and Practice Enablement sessions, when we start a new engagement with a new team, or simply to kick-start a conversation with a stakeholder on what agile is really about.

The resulting graphic is well worth printing out and framing on the wall!

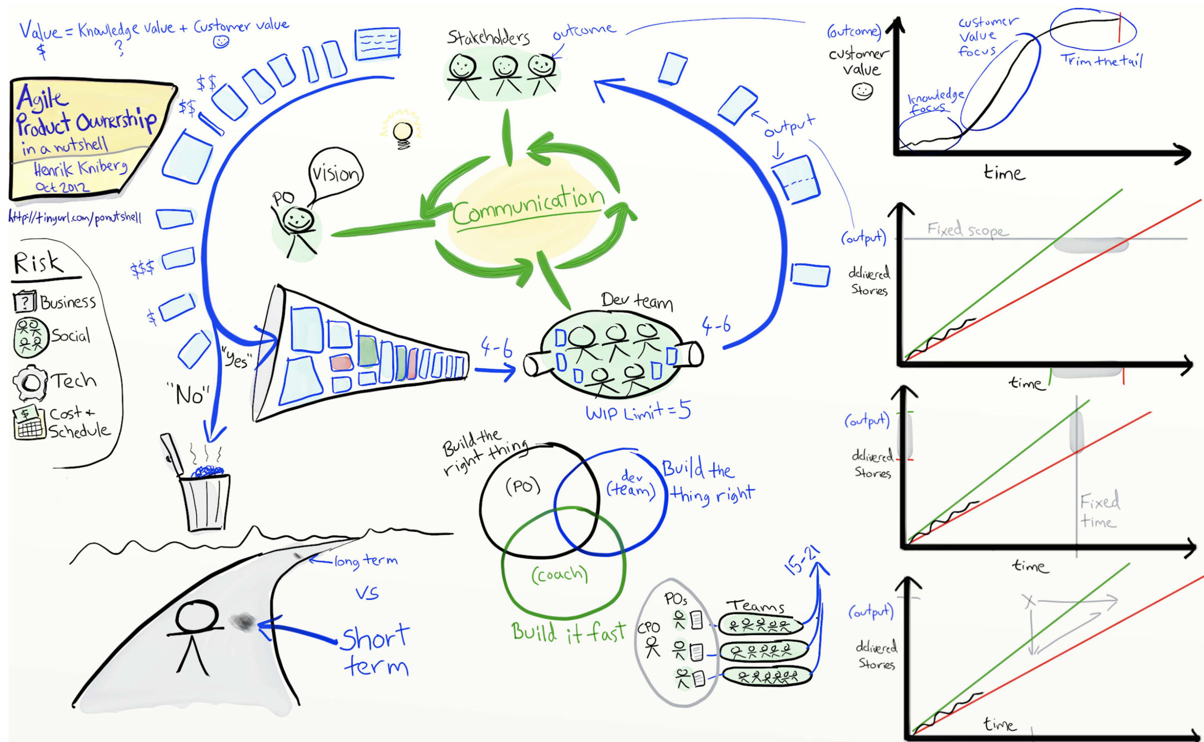


Figure 11.28: Agile Product Ownership in a Nutshell, courtesy of Henrik Kniberg

During our time working on hundreds of different engagements, we've seen some examples of amazing Product Ownership. We've also seen some really bad examples.

Let's look at some of the patterns that we've observed, starting with Product Owners.

Experimenting with Different Product Owners

Working with a UK retail organization in the mid-2010s, we were using the Scrum framework to deliver a four-tier IBM WebSphere Commerce and MobileFirst platform.

This was the first time this organization had attempted an agile delivery and it was complex due to multiple Scrum teams, one Waterfall team, multiple suppliers, multiple time zones, multiple technology vendors including some immature technology, multiple business units, and so on.

The first Product Owner assigned to the engagement was a contractor who had no prior history or engagement with the organization. We quickly learned that he had no empowerment to make decisions as he constantly had to assemble meetings with stakeholders any time a Scrum team needed clarification or asked to do some Product Backlog refinement. Product Owners need to be empowered.

The organization did adapt and re-staffed the Product Owner role to be a very senior business-focused person who was a direct report of the CIO. This was better as he certainly was empowered to make fast decisions and prioritization choices. The challenge this time was getting access to him regularly. He promised us one hour per week. With a Product Backlog that needed a lot of refinement and a lot of clarification to development teams, this was nowhere near enough. Product Owners need to be available to their team(s).

There was a further adaptation and the Product Owner role was given to a long-standing and highly respected technical architect who was dedicated to the project. This worked really well as he had a great relationship with both technology and business stakeholders, knew the organization's strategy and priorities really well, and had a great personality that people felt very comfortable collaborating with. Product Owners need to understand the business.

Over time, the need for direct access to this Product Owner diminished. It is a pattern I've noticed working with several organizations where Product Ownership has been particularly strong. Great Product Owners democratize Product Ownership and provide a direct connection between teams and stakeholders. Product Owners should see their current role as one to self-destruct and not be needed in the long term.

Next, let's look at how great Product Owners have approached their first iterations and what they've prioritized to form their first iteration goals.

Patterns of Early Sprints and the Walking Skeleton

In the next chapter, we're going to switch to the Delivery Loop and talk about the practices we use to plan, deliver, showcase, and learn from iterations of the delivery loop.

Thinking specifically about the first time a team does an iteration of delivery for a new product, I've noted a consistent pattern in what great teams have as their first delivery goal. I've worked on several engagements where the goal has been virtually identical. This may be my coaching and I have influenced this, but there is always a sense of alignment and confidence that this is the best way to de-risk, learn, and set up for faster delivery in subsequent iterations.

An example first iteration goal from a European automotive organization was:

Set up our workspace, provide a walking skeleton connecting the frontend to API to database underpinned by CI/CD.

Several other engagements have had almost identical goals, and the pattern is strong because:

- The team wants to set up their workspace. That may be their physical workspace with lots of information radiators and collaboration space. It may be a virtual workspace with digital tooling. It may be a development environment using code-ready workspaces and being familiar with all tools to be used.
- The plan to build a walking skeleton. This is a thin slice of the whole architecture delivered in one iteration. There won't be any fancy frontend or complex backend processing. They will prove full-stack development and that the cross-functional team representing all parts of the logical architecture can deliver working software together. It's a walking skeleton because it is a fully working product. It just doesn't do very much yet!
- Their work will be underpinned by continuous integration and continuous delivery. This green-from-go practice means they are set up for success when it comes to automating builds, tests, and deployments. If they prove this and learn this for a thin slice, it will become increasingly valuable as we start to put all the flesh and organs into the walking skeleton!

The final part of this chapter shifts the focus from what we're deciding to deliver next to how we're going to measure and learn from our experiments and the features we deliver. The OpenShift platform enables our teams to consider several advanced deployment capabilities.

Advanced Deployment Considerations

Earlier in this chapter, we explained the practice of **Design of Experiments** and how we intend to take an experimental mindset to our development, especially where assumptions and hypotheses have been formed in our Discovery Loop practices.

The OpenShift platform enables several different deployment strategies that support the implementation of experiments. When we are on the Options Pivot, we should consider these strategies and which (if any) we should plan with the delivery of the associated Product Backlog item. The advanced deployment strategies we can consider include:

- A/B Testing
- Blue/Green Deployments
- Canary Releases
- Dark Launches
- Feature Toggling

We introduce these concepts here as, from an options planning perspective, this is where we need to be aware of them. We'll return to specific implementation details in *Section 6*, *Build It*, *Run It*, *Own It*, and we'll explore how we use the resulting metrics in *Section 7*, *Improve It*, *Sustain It*.

A/B Testing

This is a randomized experiment in which we compare and evaluate the performance of different versions of a product in pairs. Both product versions are available in production (live) and randomly provided to different users. Data is collected about the traffic, interaction, time spent, and other relevant metrics, which will be used to judge the effectiveness of the two different versions based on the change in user behavior. The test determines which version is performing better in terms of the Target Outcomes you have started with.

A/B Testing is simple to apply, fast to execute, and often conclusions can be made simply by comparing the conversion/activity data between the two versions. It can be limiting as the two versions should not differ too much and more significant changes in the product may require a large number of A/B Tests to be performed. This is one of the practices that allows you to *tune the engine*, as described in *The Lean Startup*[9] by *Eric Ries*.

9 http://theleanstartup.com/

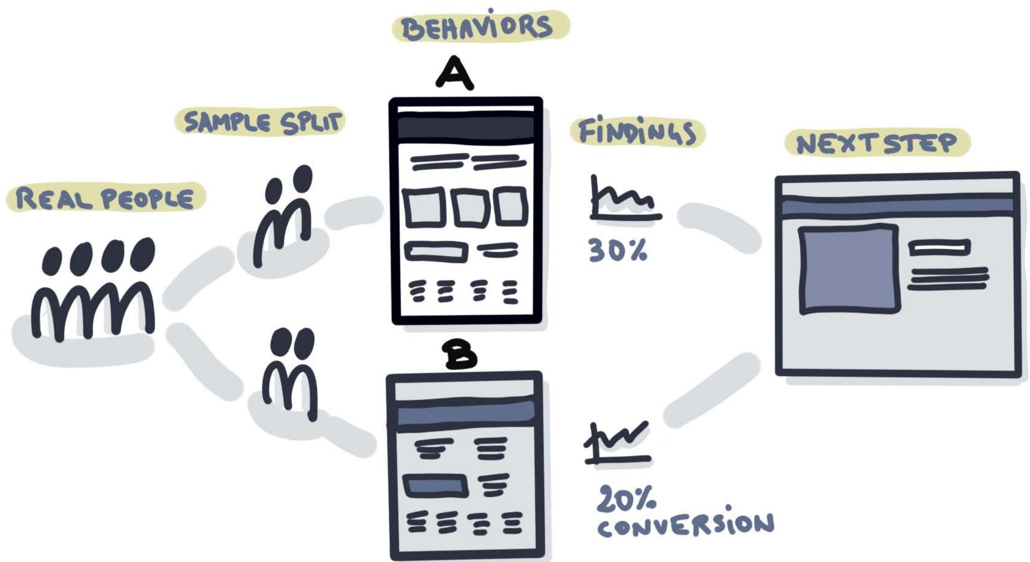


Figure 11.29: A/B Testing

For more information on this practice and to discuss it with community members or contribute your own improvement to it, please look at openpracticelibrary.com/practice/split-testing-a-b-testing/.

Blue/Green Deployments

Blue/Green Deployment is a technique in software development that relies on two productive environments being available to the team. One of them, let's call it **green**, is operational and takes load from the reverse proxy (load balancer/router). The other environment, let's call it **blue**, is a copy upgraded to a new version. It is disconnected from the load balancing while this upgrade is completed.

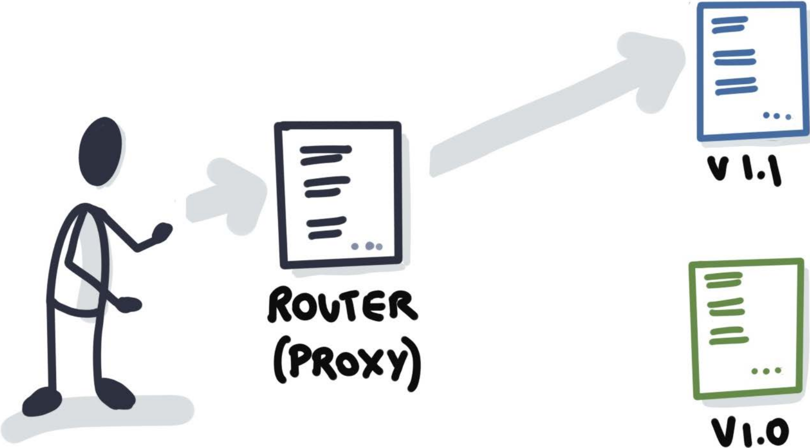


Figure 11.30: Blue/Green Deployments

The team can perform all required tasks for an upgrade of the product version on the **blue** environment without the rush of downtime. Once the **blue** environment is ready and has passed all tests and checks, the team simply redirects the reverse proxy (load balancer/router) from the **green** environment to the **blue** environment.

If everything works fine with the **blue** environment, the now outdated **green** can be prepared to be recycled to serve as the **blue** for the next release. If things go bad, the team can switch back to a stable environment instantly using the reverse proxy/load balancer/router.

This is a feedback loop practice that allows the team to get prompt feedback from the real-life use of their changes. It enables continuous delivery and provides safety for performing complex releases. It removes the time pressure and reduces the downtime to practically zero. This is beneficial for both technical teams and end users who will not notice glitches or unavailability of the service/product, provided that the new version is performing at par. In case of adverse effects, it allows the teams to have an instant roll-back alternative and limit the negative impact on customers.

To explore this practice further, visit the Open Practice Library page at openpracticelibrary.com/practice/blue-green-deployments/.

Canary Releases

In software development, this is a form of continuous delivery in which only a small number of the real users of a product will be exposed to the new version of the product. The team monitors for regressions, performance issues, and other adverse effects and can easily move users back to the working old version if issues are spotted.

The term comes from the use of caged birds in coal mines to discover the buildup of dangerous gases early on. The gases would kill the bird long before they became life-threatening for the miners. As with the canary in the mine, this release practice provides an early warning mechanism for avoiding bigger issues.

The canary release provides continuous delivery teams with safety by enabling them to perform a phased rollout, gradually increasing the number of users on a new version of a product. While rolling out the new version, the team will be closely monitoring the performance of the platform, trying to understand the impacts of the new version, and assessing the risks of adverse effects such as regressions, performance issues, and even downtime. This approach allows the team to **roll back** the release as soon as such adverse effects are observed without the majority of the customers being impacted even for a limited amount of time.

Canary Release is similar to A/B Testing in the sense that it is only exposing a part of the population to the new feature, but unlike A/B Testing, the new feature can and is typically a completely new feature and not just a small tweak of an existing one. The purpose is different too. A/B Testing looks to improve the product performance in terms of getting business outcomes, while the Canary Release is focused entirely on technical performance.

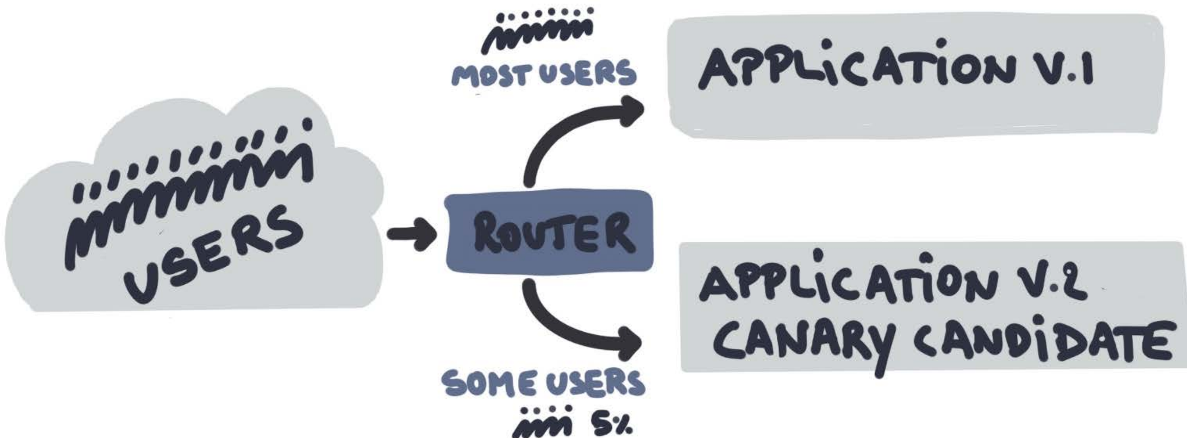


Figure 11.31: Canary deployments

You can read more about this practice, contribute improvements, or have a discussion with the wider community at openpracticelibrary.com/practice/canary-release.

Dark Launches

Dark Launches are another continuous delivery practice that release new features to a subset of end users and then captures their behaviors and feedback. They enable the team to understand the real-life impact of these new features, which may be unexpected for users in the sense that no users asked for them. It is one of the last steps for validating a product/market fit for new features. Rather than launching the features to your entire group of users at once, this method allows you to test the waters to make sure your application works as planned before you go live.

Dark Launches provide safety by limiting the impact of new features to only a subset of the users. They allow the team to build a better understanding of the impact created by the new feature and the ways the users would interact with it. Often novel ways of interaction can surface, ways that were not initially envisioned by the team. This can be both positive and negative, and the limited availability allows the team to draw conclusions from the real-life use and decide if the feature will be made widely available, further developed, or discontinued.

Figure 11.32: Dark Launches

The Dark Launches practice has its own Open Practice Library page at openpracticelibrary.com/practice/dark-launches/, so head there for further information, to start a conversation, or to improve the practice.

Feature Flags

Feature Flags (also known as Feature Bits/Toggles/Flipping/Controls) are an engineering practice that can be used to change your software's functionality without changing and re-deploying your code. They allow specific features of an application to be turned on and off for testing and maintenance purposes.

In software, a flag is *one or more bits used to store binary values*. So, it's a Boolean that can either be true or false. A flag can be checked with an if statement. A feature in software is a bit of functionality that delivers some kind of value. In its simplest form, a Feature Flag (or Toggle) is just an `if` statement surrounding a bit of functionality in your software.

Feature Toggles are a foundational engineering practice and provide a great way to manage the behavior of the product in order to perform experiments or safeguard performance when releasing fresh new features.

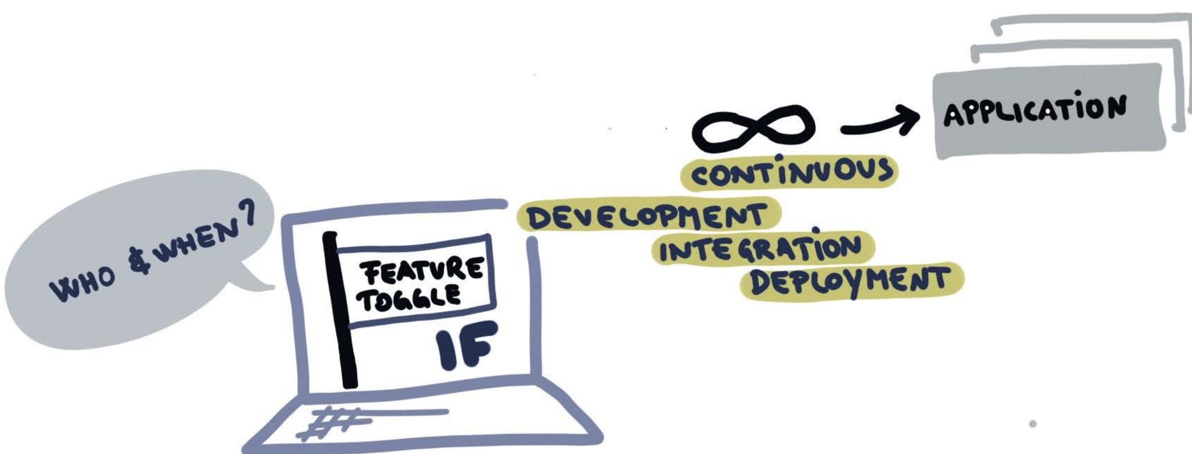


Figure 11.33: Feature Flags

This practice has a page in the Open Practice Library at openpracticelibrary.com/practice/feature-toggles, so have a look there for more information or to engage with the community on it.

Having introduced these advanced deployment considerations and how they support the Design of Experiments, let's look at how our PetBattle team might use them.

PetBattle – Tech Spikes, Prototypes, Experiments, and Feature Implementations

The team gathered by the new Product Backlog, which was, floor to ceiling, a huge column of sticky notes! In fact, they didn't all fit in a single column so the items toward the bottom fanned out to produce what looked like a big funnel. It was nice and ordered at the top but the bottom would need some organizing. But that was OK as the team reckoned there was enough work on the backlog to keep them busy during the next few weeks, so they could refine the Product Backlog as they went along.

Having attended a workshop on advanced deployment considerations, they decided to do some refinement on the functional features on the Product Backlog. They produced three columns on their whiteboard with the headings Research, Experiment, and Implement. They time-boxed the discussion for 45 minutes. There were 15 features to talk through, so on

average, 3 minutes per item. Their goal was to put each feature in one of the columns with a short note on what their approach to the implementation was.

- **Open PetBattle:** This was easy. Anyone using the app would need to open it. IMPLEMENT.
- **Display Leaders:** Lots of questions about what and how to display. How many leaders? Should we add pagination or scroll? They decided some RESEARCH was needed – perhaps a UI prototype with some user testing.
- **Let me in please:** The team had to go back to the Event Storm to remind themselves what this was about! Again, it was a simple feature of letting the user in to see Pets uploaded. IMPLEMENT.
- **Vote for Cat:** This triggered some conversation. Do they vote up or down? Or do they just give a vote (or nothing at all)? The team was divided and had heard differing views from user interviews. They decided to EXPERIMENT with an A/B Test.
- **Add my Cat:** Not much research or experimentation needed. A standard uploading tool was needed. Just IMPLEMENT.
- **Verify Image:** This sounded a bit trickier. There were merging AI/ML patterns available. It needed some technical RESEARCH and probably a Technical Spike.
- **Enter cat into tournament:** Not much ambiguity here. IMPLEMENT.
- **Display Tournament Cat:** It wasn't clear if this was going to be well received or not. The team thought they could EXPERIMENT with a feature toggle and then it's easy enough to turn off.
- **Disable "Add my Cat":** Some users have more than one cat and will want to add more than one. Let's EXPERIMENT with a Dark Launch of this feature to a small subset of users.
- **Vote for given cat:** Once the team got the results from the A/B Test, they could EXPERIMENT further and launch as a Canary Test.
- **Update the Leaderboard:** IMPLEMENT
- **End Competition:** IMPLEMENT
- **Notify Players:** Not clear how this would happen – SMS? Email? Other mechanisms? The team decided to do some user RESEARCH.

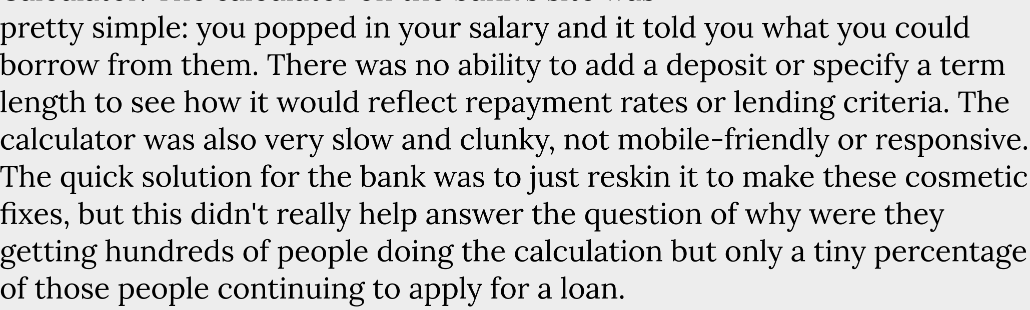
- **Deliver Prize to Winner:** The prize strategy was still under consideration, so more RESEARCH would be needed.
- **Begin Next Tournament:** This could either happen immediately or the next day. Perhaps another EXPERIMENT A/B Test would see what drop-off the team gets by waiting.

The team finished in 40 minutes. It was a great discussion and they felt ready to do their first iteration planning with these decisions.

Let's look at another real-world experience to see just how simple yet effective this experimental mindset can be.

Reframing the Question – How Much Can I Borrow or How Much House Can I Afford?

It can be very easy to fall into the trap of recreating what you already have but with a shiny new frontend, but ultimately that may not solve problems that are more fundamental. While working for a bank, a small team of us were tasked with trying to find out why there was such a large drop-off in mortgage applications once people had used an online Mortgage Calculator. The calculator on the bank's site was pretty simple: you popped in your salary and it told you what you could borrow from them. There was no ability to add a deposit or specify a term length to see how it would reflect repayment rates or lending criteria. The calculator was also very slow and clunky, not mobile-friendly or responsive. The quick solution for the bank was to just reskin it to make these cosmetic fixes, but this didn't really help answer the question of why were they getting hundreds of people doing the calculation but only a tiny percentage of those people continuing to apply for a loan.



The team was very small, just one designer, two engineers, a business analyst, and a Product Owner. As a small co-located team, buried in the heart of the bank, we were able to move fast! We interviewed people who had recently purchased mortgages with the bank to get insight into their motivations for using the tool. We did a load of research by going into the bank branches and asking people open-ended questions while they used the existing tool. We collated this information along with how they were accessing the calculator and if they were to complete an application, what device they would use,that is, their phone or their laptop.

Through this research we stumbled upon an interesting fact – people were not interested in *How much could I borrow* but *How much house can I afford.* This simple difference might seem inconsequential but it massively affected how we rebuilt the bank's online mortgage calculator. It meant people wanted to be able to tailor their calculation to see how their rates and lending criteria could be affected by, for example, having more income. Or, if they were to continue to save for another year and have more of a deposit saved, could they get a better rate? This flip meant people were using the tool to not see if they could afford a given home but how much of a home could they afford and by when.

It would have been very simple for us to just recreate the bank's existing calculator with a new skin that ran on a mobile – but this would not have addressed the core problem. By reframing the question, we were now in a position to create a simple calculator tailored to the needs of the bank's first-time buyers.

All these advanced deployment considerations provide powerful tools for use in Options planning and how we can conduct research, experimentation, and implementation.

Research, Experiment, Implement

This chapter has highlighted that, when considering our options, it's not just about prioritizing features and implementing them. We need to balance feature implementation with ongoing research and experimentation.

A great way to summarize and visualize all of this is by using the **Mobius Loop's Options Map**.

Creating an Options Map

We concluded *Section 3, Discover It*, with a Discovery Map – a single information radiator that summarized the iteration of the Discovery Loop.

We will conclude *Section 4, Prioritize It*, with an Options Map – this is another open-source artifact available in the Mobius Kit under Creative Commons that you can use to summarize all the learnings and decisions taken during your journey in the Options Pivot.

This map should slot neatly next to your Discovery Map and is used to summarize:

- What have we just focused on in the Options Pivot?
- Which outcomes are we targeting?
- What ideas or hypotheses do we need to validate?
- How will we deliver the options?
- Which options do we want to work on first?

When we return to the Options Pivot after an iteration of the Delivery Loop, we'll complete the final section of this map:

- What did we learn?

Figure 11.34: The Options Map

Now let's look at PetBattle's Options Map.

PetBattle – The Options Map

There is a lot of detail captured in this map, so please feel free to explore the online version available in the book's GitHub repository.

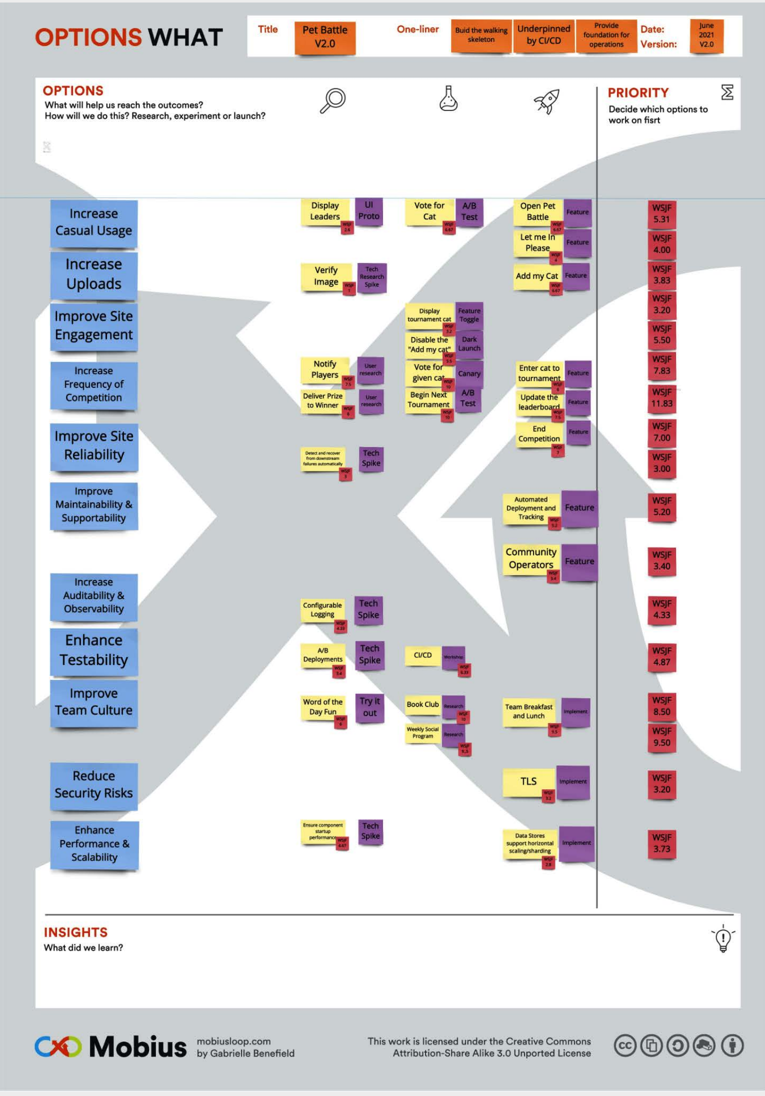


Figure 11.35: The PetBattle Options Map

The Options Map provides clarity and direction as to how the product priorities to help reach outcomes. It helps form our delivery strategy.

Conclusion

In this chapter, we focused on how we are going to deliver the outcomes set in the previous section.

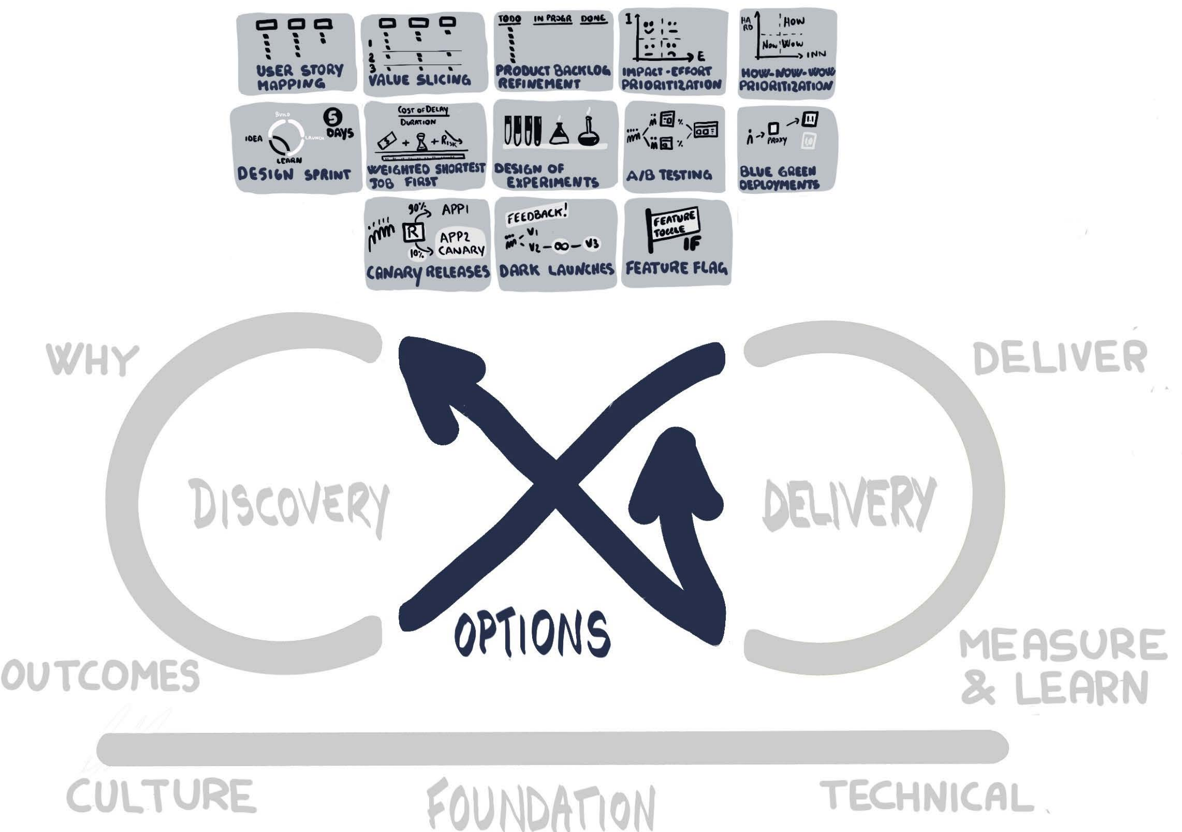


Figure 11.36: Adding practices to navigate us through Options

We explored the User Story Mapping and Value Slicing practices and how we take all of the information captured in Discovery practices and push it through these tools. We also showed how using some helpful practices to look at the same information with slightly different lenses –Impact versus Effort Prioritization and How/Now/Wow Prioritization – can help improve Value Slicing. Where proposed feature areas would benefit from a deeper dive to understand the value, we recommended the Design Sprint as an option.

We showed how these practices drive the initial Product Backlog prioritized by value and how this produces a living, breathing artifact that will be subject to continuous Product Backlog Refinement as we gather more learning, feedback, and metrics for our delivery. The economic prioritization model WSJF, which is based on Cost of Delay, provides a repeatable and quantifiable tool to drive this. It's one of many prioritization tools that can help the Product Ownership function work smoothly and effectively.

Finally, we looked at the advanced deployment considerations that should be taken when designing experiments and how platforms such as OpenShift enable powerful evidence-based testing to be conducted in production with users. A/B Testing, Blue/Green Deployments, Canary Releases, Dark Launches, and Feature Flags were all introduced from a business perspective. We will return to the implementation details of these in *Section 6, Build It, Run It, Own It* and explore how we interpret the measures from them in *Section 7, Improve It, Sustain It.*

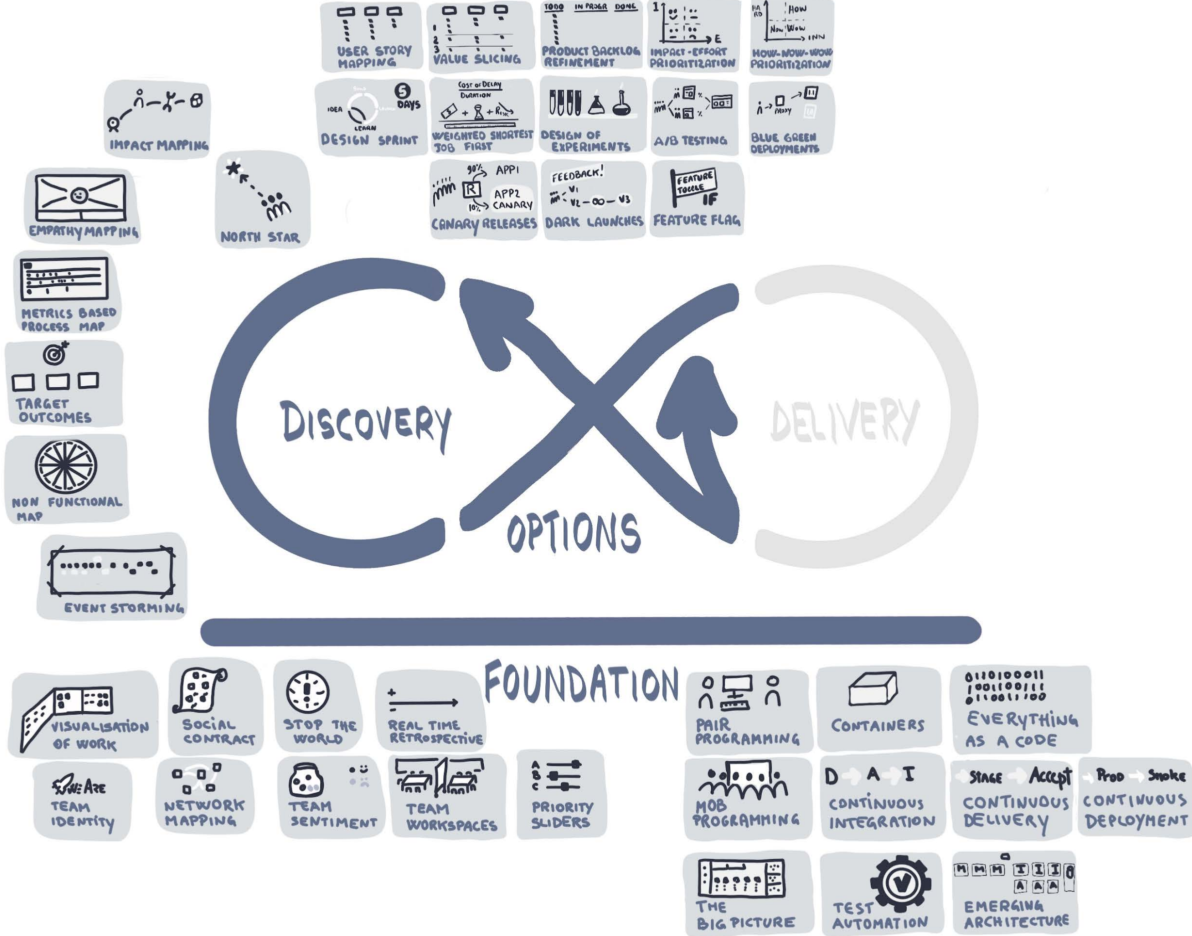


Figure 11.37: Practices used to complete a Discovery Loop and Options Pivot on a foundation of culture and technology

In the next chapter, we will shift to the Delivery Loop. We'll look at agile delivery and where and when it is applicable according to levels of complexity and simplicity. We'll also look at Waterfall and the relative merits and where it might be appropriate. We'll explore different agile frameworks out there and how all of them relate to the Open Practice Library and Mobius Loop. We'll explore the importance of visualization and of capturing measurements and learning during our iterations of the Delivery Loop.

Section 5: Deliver It

In *Section 4*, *Prioritize It*, we moved through the Options Pivot for the first time. We took all of the learning and ideas that can come out of the practices used on the Discovery Loop and applied more practices to organize them according to their value. We used prioritization techniques to help us slice value and considered how to plan and design experiments that could be run by making use of advanced deployment capabilities offered by the OpenShift platform.

As outlined in the *Mobius Outcome Delivery QuickStart Guide* by Gabrielle Benefield and Ryan Shriver (freely available to download from mobiusloop.com), Mobius encourages product development through the use of continuous discovery, validation, and learning. Building products this way requires a strong foundation of culture and technical practices that enable an ongoing experimental fast release and fast feedback approach. Indeed, ongoing research and experiments are just as much part of product development as feature delivery.

The key differences between research, experimentation, and implementation are the investments in time and money required to steer a team toward building the right thing that is aligned with the target outcomes. Mobius encourages the idea of delivering maximum value from the least amount of investment.

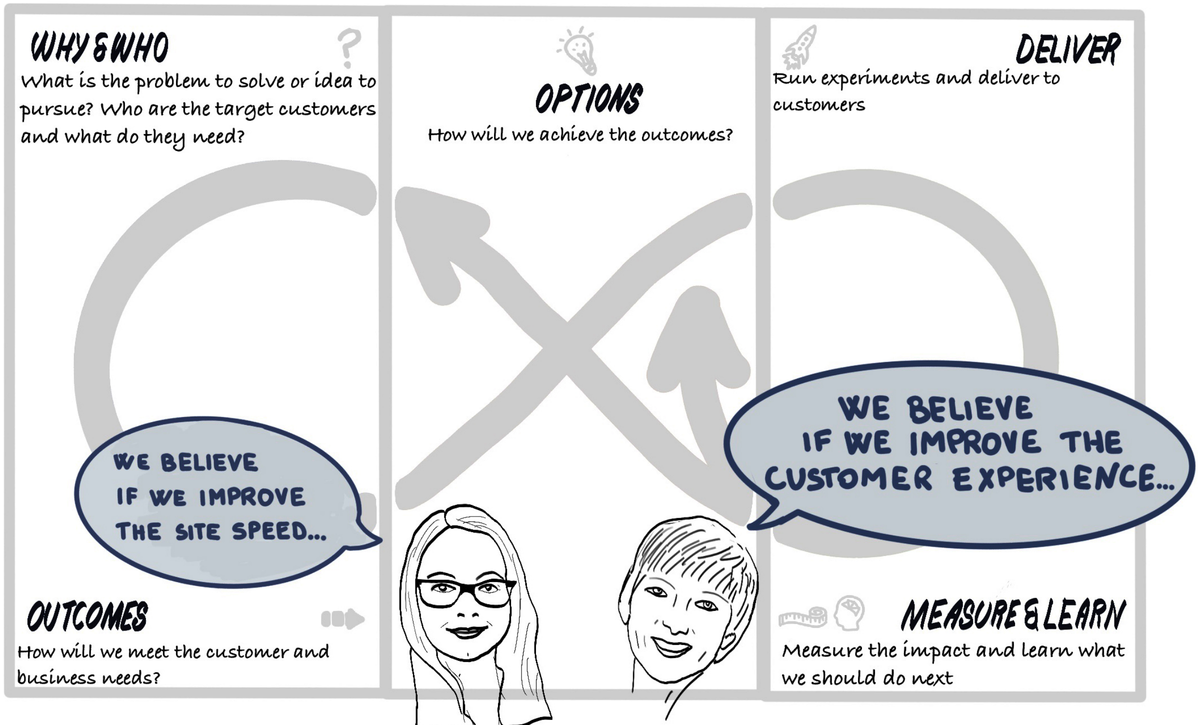


Figure 12.0.1: The Delivery Loop – setting the scene

As we move onto the Delivery Loop, we're going to carry out research, execute experiments, and implement features. How are we going to do this? Are we going to use a framework like Scrum, or **Scaled Agile Framework** (**SAFe**), or **Large-Scale Scrum Framework** (**LeSS**), or Lean? Are we going to follow a prescriptive approach or methodology? No, we are not! As has been the case throughout this book—indeed, as is the mindset behind Mobius and the Open Practice Library—we are going to offer a toolbox of practices that will help you work your way around the Delivery Loop. We will not say there is one right way to do things because it does not exist. Our approach is to adapt based on context, learning, and experience. We will look at some of the popular practices that we have found to be very effective and share some stories and tips behind them.

Which practices you choose is up to you but, in nearly all cases, we strongly recommend using practices that allow fast delivery, rapid feedback, and continuous learning throughout.

To this effect, this section is divided into two chapters. *Chapter 12*, *Doing Delivery*, is all about Doing Delivery—the difference between Waterfall and Agile approaches and popular Agile practices to incrementally and iteratively deliver. *Chapter 13*, *Measure and Learn*, is all about measuring metrics and learning about what they tell us. We can use the many different flavors of metrics, applied to the small incremental deliveries, to continually improve.

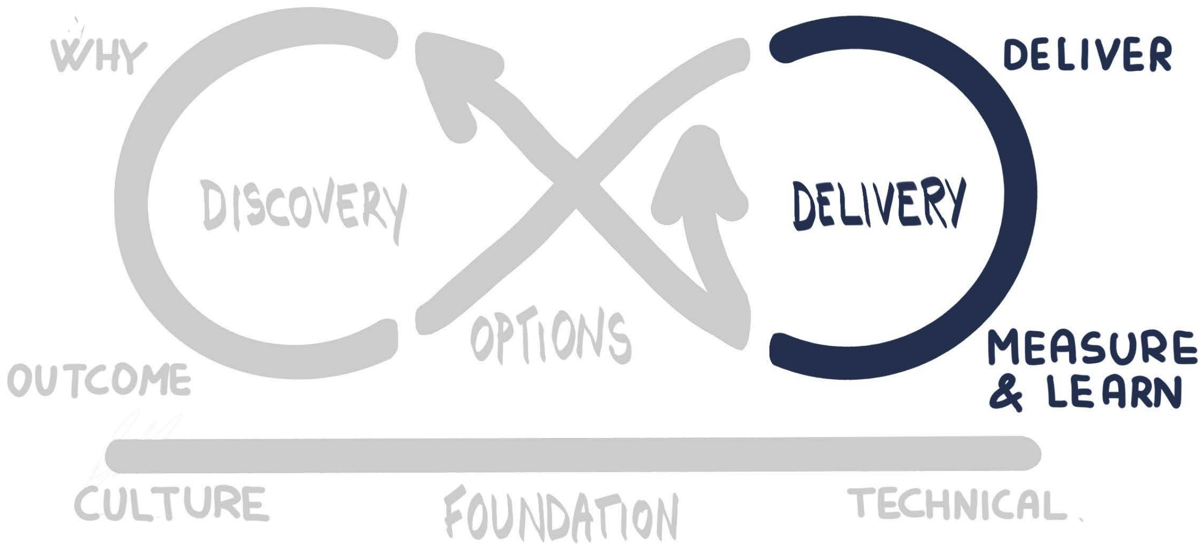


Figure 12.0.2: The Delivery Loop

12

Doing Delivery

Ultimately, code has no value until it is delivered to production and its associated functionalities are in use. It's time to connect all our learning to date to delivery.

We are well into the second half of this book and other than setting up a technical foundation, as we did in *Section 3*, *Discover It*. we haven't written any application code, nor have we delivered anything yet. What we have done is go through a tremendous set of practices to help discover, set outcomes, assess and prioritize options, and decide what we are going to deliver next. Now, we have reached the point where we will write some code, deliver features, execute experiments, and learn from further research.

In this chapter, we are going to look at different delivery approaches. Your immediate expectation might be that this chapter is going to be all about **Agile** and that using Agile methodologies is now the only way to deliver working software. However, in addition to taking a look at Agile, where it came from, and different ways we can use it, we will highlight that it is not the only option. There is still a place for more traditional approaches to delivery, such as **Waterfall**. In this chapter, we will:

- Explain the Waterfall and Agile methodologies.
- Look at the difference between complicated, complex, and clear systems and how we can use them to drive the applicability of Agile.
- Explore how the Mobius Loop is framework-agnostic and, through different perspectives, speed, and frequency of iterations, can be used to explain delivery of every type.
- Share some tools and tips about getting **ready** for Delivery Iterations.

- Describe the practices with examples we often use to help with Delivery Iterations; including, for example, Iteration (or Sprint) Planning, Daily Stand-Ups, Showcases, and Retrospectives.
- See how, as teams mature and continuously improve, they can adapt their working practices and approaches.
- Share some tools and tips about getting working software features **done** with a focus on both functional and non-functional acceptance.
- See, for the first time, how all of our practices blend together to give us a continuous delivery engine.

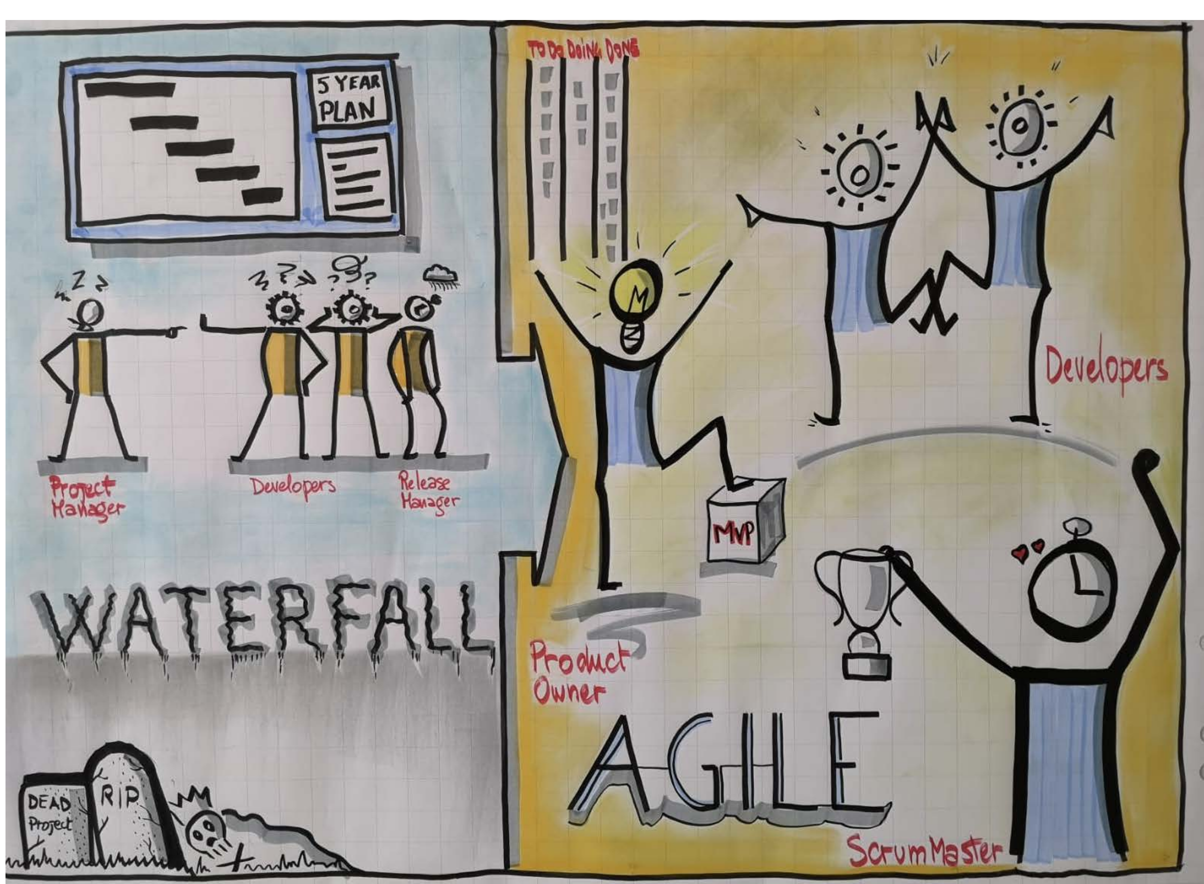


Figure 12.1: Waterfall and Agile teams

Let's start by having a look at those two phenomena—Waterfall and Agile. As depicted in *Figure 12.1*, Waterfall is often associated with Gantt charts, dependencies, and hand-offs and hand-overs from one part of the organization to another. Agile takes a different mindset and is more cross-functional. Some may say Waterfall is old and dying.
We'll see later in this chapter that there remain valid use cases for taking a Waterfall approach.

Waterfall

Waterfall projects are broken down into linear steps; each step relies on the previous one being completed before starting the next. The first step is to gather user requirements, followed by designing and planning, and then finally software development. Once testing and deployment are done, we enter the maintenance and operation mode.

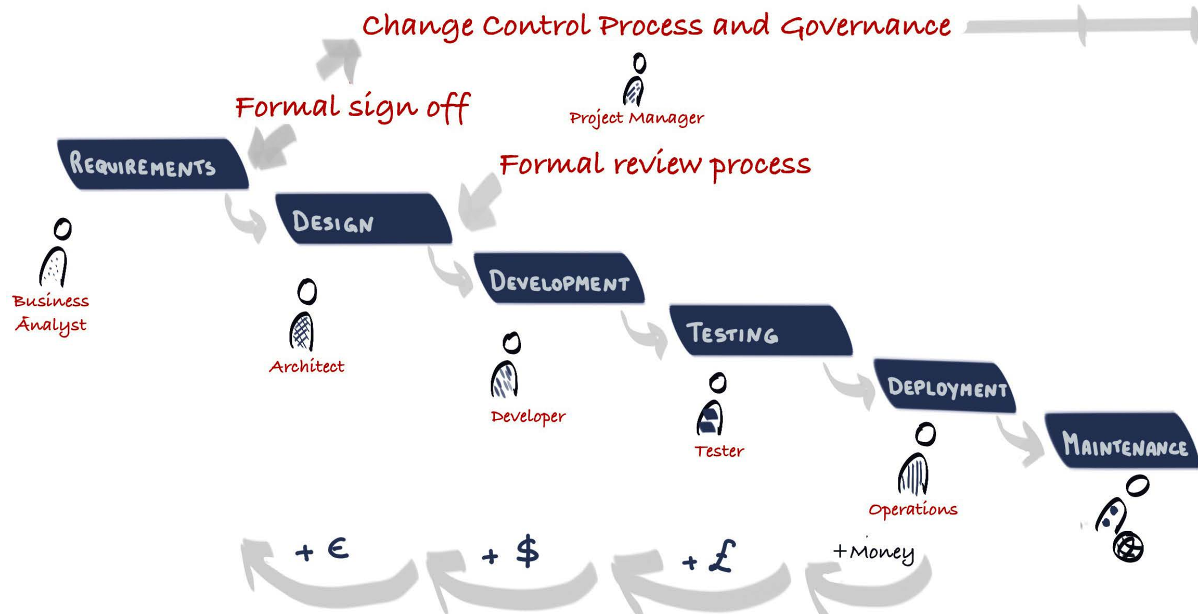


Figure 12.2: Waterfall process

This type of delivery methodology has a long history outside of software development, especially in traditional engineering. When building a bridge or a factory, the *project plan* lays out the requirement of all the people, resources, tasks, and timelines for the overall project of work. The Waterfall approach can be described as a *plan-driven* engineering process where success is measured according to how well development is able to keep up with the plan.

By following a plan, individual variations are minimized. In the industrial world, this helps make both delivery and cost predictable. Because the traditional Waterfall model does not allow for looping, requirements and design are gathered in vast documents to begin planning. Changing and understanding the documents becomes very tricky for teams because there are no mechanisms for getting a shared understanding other than reading all the documents over again. Have you ever participated in a book review session? How many different points of view do you get when a group of people reads the same book? Coupling this with the likeliness of an organization to raise expensive change requests and making changes to the documentation can quickly show how disastrous this approach can be for a company. So how does this method fare in the world of software?

Often, we find that in the software realm the requirements do not match what the end users expect. This is because as we develop the software, questions and misunderstandings about the original requirements and design become hard to address. As design and business requirement teams are not heavily involved in the development phase, there is no opportunity to refresh the designs or ideas.

Delivering customer and business value through working software takes a lot longer using Waterfall—as working software is only delivered toward the end of a Waterfall process. Think of how much time is lost while you're waiting for your great application to arrive. What if it never arrives? Or worse, arrives too late and you've missed your market?

Changing requirements during a Waterfall execution is often left to another Waterfall phase. If changes are urgent, then they may be handled through some form of change request mechanism. However, this is really a form of loop—leading us to a more iterative approach. In Waterfall, these loops can still be very time-consuming and costly. If only there was a quicker way!

Winston Walker Royce was an American computer scientist and director at Lockheed Software Technology Center in Austin, Texas. He was a pioneer in the field of software development, known for his 1970 paper from which the idea of the single-pass Waterfall model for software development was mistakenly drawn. That's right, Waterfall was misunderstood and mistaken!

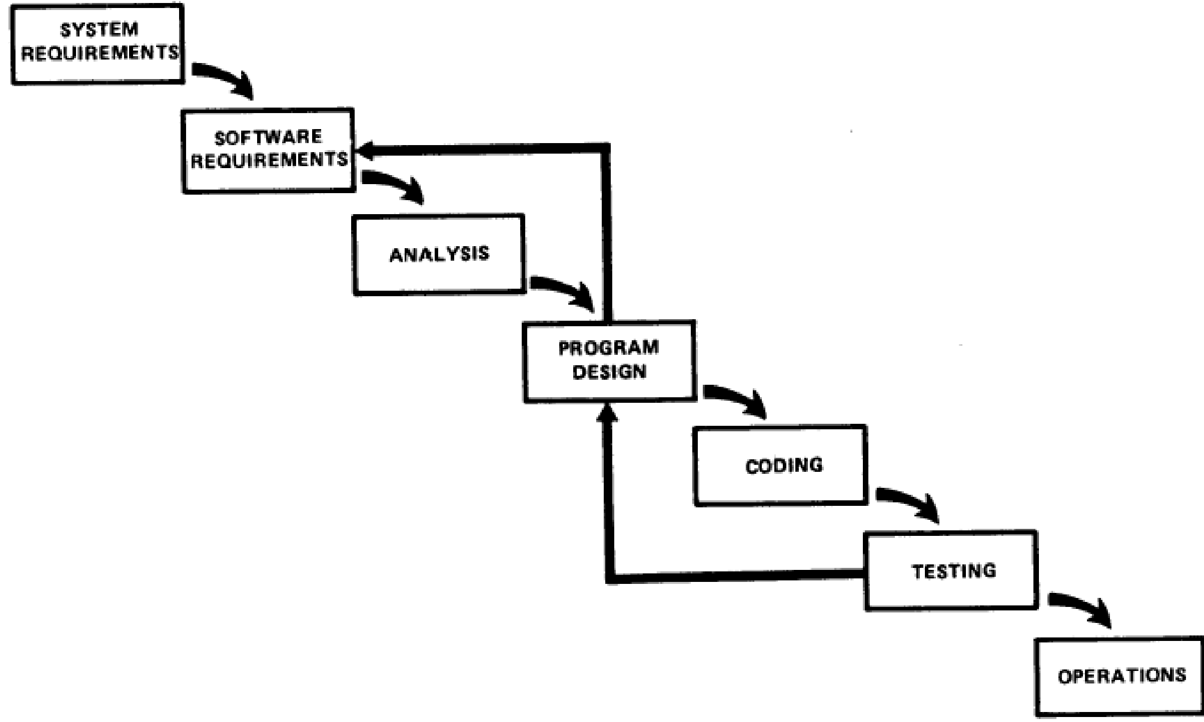


Figure 12.3: Waterfall was not designed as a single pass!

Barry Boehm wrote in 1987: "*Royce's 1970 paper is generally considered to be the paper which defined the stagewise Waterfall model of the software process. But it is surprising to see that both the earlier Benington and Hosier papers had good approximations to the Waterfall model, and that Royce's paper already incorporates prototyping as an essential step compatible with the Waterfall model.*"[1]

Royce demonstrated that while the development of large software systems required a more thorough approach, there was an inherent risk in the single-pass sequential approach. He proposed an iterative approach and advocated that projects should pass through this at least twice. But many Waterfall projects started and have continued to follow just a single-pass flow, left to right, with just a single iteration, which was not intended or recommended by Royce—Waterfall plans were never supposed to be executed just once.

The Waterfall model did help in the innovation of other approaches and the focus on iteration was corrected in Agile models.

1 Barry W. Boehm (1987). "Software Process Management: Lessons Learned from History" in ICSE '87 Proceedings of the 9th international conference on Software Engineering pp 296-298

The Birth of Agile

Agile development methods have been talked about and practiced for over two decades now. The *Agile Manifesto* came about when a bunch of men, all prominent in software development, got together at a ski retreat in Utah to do some critical analysis on why IT projects were getting such a bad name. They looked back at the previous 10 years of software delivery throughout the '90s and concluded IT projects were taking too long to execute, coming over budget, and often not delivering value to end users. So these men sat down at the end of their trip and wrote the *Manifesto for Agile Software Development*.

We know what you're thinking: this manifesto thing must have been a massive document! A huge book filled with detailed instructions on how to write software as well as how to manage requirements and costs. In fact, it was much simpler than this—so simple you could almost fit it in a tweet!

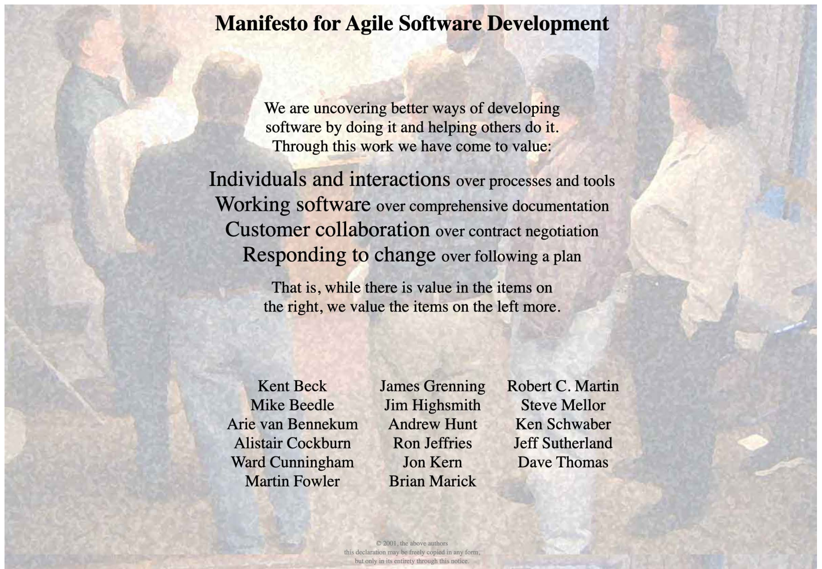


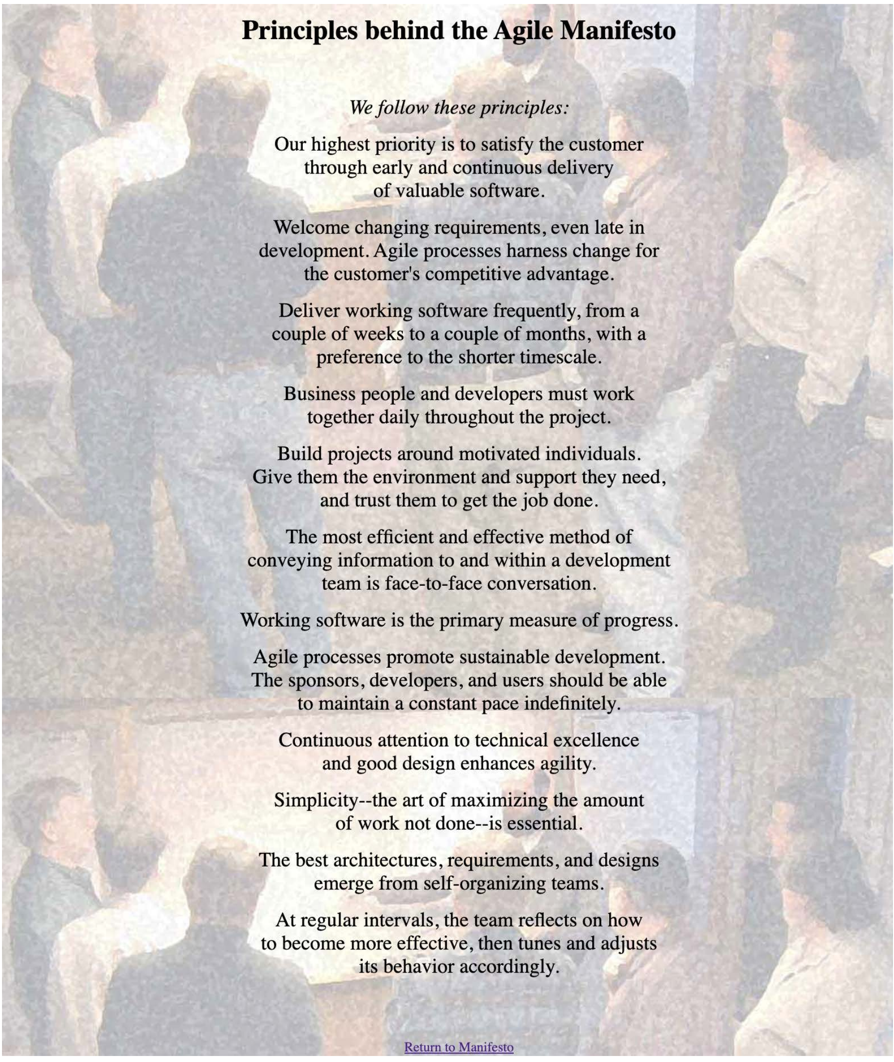
Figure 12.4: The manifesto for Agile software development

The Agile Manifesto[2] is a simple set of values, highlighting the importance of some values over others.

Working software over comprehensive documentation—This sounds great, I don't have to write any documentation is what some are probably thinking. But they would be wrong. The values are not saying things on the right are not important, but that there is higher value in the items on the left. This is the bit people tend to forget when quoting the manifesto.

2 https://agilemanifesto.org/

In addition to this set of values, this group came up with a set of 12 principles[3] to support them.



Principles behind the Agile Manifesto

We follow these principles:

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity--the art of maximizing the amount of work not done--is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Return to Manifesto

Figure 12.5: Principles behind the Agile Manifesto

These principles provide more detail about the kind of behaviors they foresaw as *being agile*. Read together, the principles provide an incredibly useful resource when having conversations about what it means to be *agile*. Agile these days has really become an overused buzzword and is frequently injected into sentences. Examples include—"*We're going to use Agile to deliver this project*" or "*We're going to install Agile.*" Reading through and having a conversation about the 12 statements can provide clarity and alignment on what we mean by Agile.

3 https://agilemanifesto.org/principles

The Agile Manifesto contrasts with the plan-driven Waterfall approach in that it:

- Is more *adaptive*, rather than *prescriptive*.
- Is more *people-oriented*, rather than *process-oriented.*

At the beginning of the industrial age, people working on the factory floor were described by Frederick Taylor, an American mechanical engineer widely known for his methods to improve industrial efficiency, as being *lazy*, *venal*, and *stupid*. A factory was built and designed by architects and engineers who did not work on the factory floor. One of the tenets of Agile is that the people and teams who do the work decide on how it gets done.

An important part of Agile is its focus on cross-functional teams delivering working software. The designers and business stakeholders are continuously involved in this process. Unlike Waterfall, when questions arise about the end user experience, these can be researched and answered based on the most up-to-date information available to the team.

Agile methodologies focus on simultaneous workflows that form a baseline to help us control change. Plans are constantly revised to reflect the learnings during a product's development. Success is based on the value delivered by the working software.

Agile is focused around products that are directly connected to the end user as opposed to effort and output that may not be directly connected. The team comes together to drive successful delivery where processes and tools are used to optimize product delivery.

One of the most noticeable traits of Agile is its breaking down of project delivery into smaller iterative phases. This works really well for software products that require regular revisions and testing. Because the work is done incrementally, teams can easily adjust what is being worked on. They can pivot to work on high-value features and deliver those to customers early and fast.

Some critics of Agile methods quite rightly point out that Agile projects also fail. However, it is not methodologies that fail, it is teams and organizations that fail. Often the reasons can be clearly articulated as the team taking on too much *technical debt*—adding new features or changing existing software is just too hard to do quickly because the code base is a mess, or the architecture is just plain wrong.

Fundamentally, Agile is an ideology or a philosophy—a way of thinking. It is articulated by the four value statements in the Agile Manifesto and defined further by the twelve principles. Agile is instantiated and brought to life by a series of practices, many of which are in the Open Practice Library. Practices have been grouped together to form popular Agile frameworks such as Scrum, Kanban, and **SAFe** (**Scaled Agile Framework**). All of these make up the Agile ecosystem.

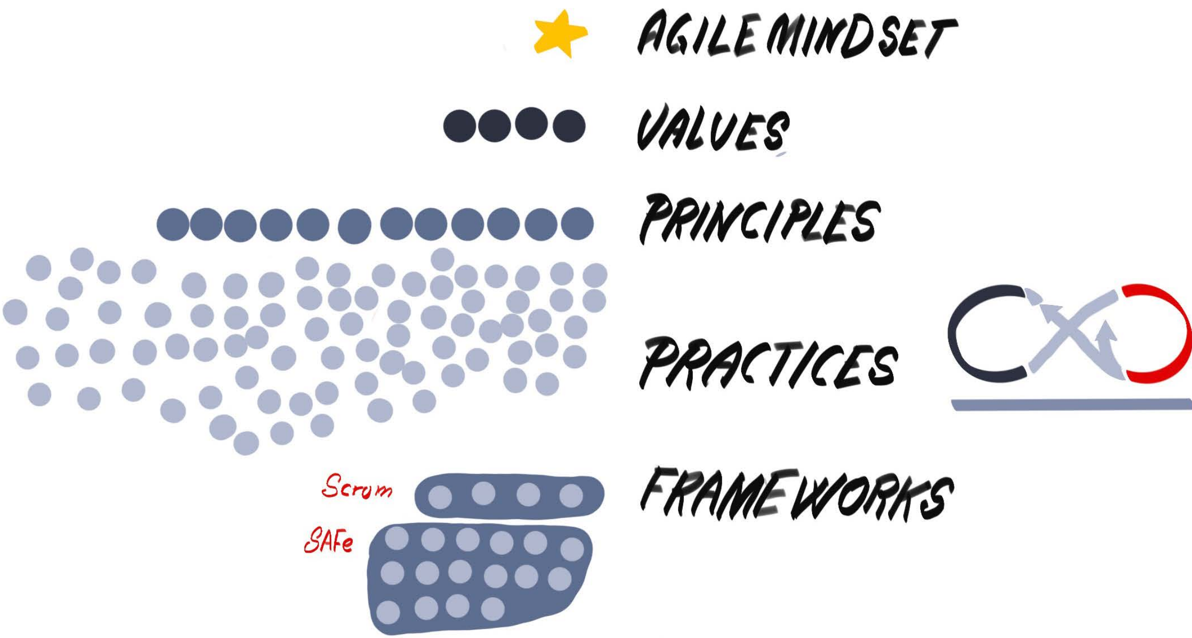


Figure 12.6: Decomposing the Agile mindset into values, principles, practices, and frameworks

Before we get into the detail of Agile, let's consider how OpenShift complements Agile and promotes business agility.

How Does OpenShift Help?

Technology is one of the foundational pillars that makes Agile productive and responsive to change. OpenShift is the collaborative platform that allows team members with diverse sets of skills to come together to securely write, test, deliver, and operate their application software. The platform enhances team dynamics, as team members can quickly and easily work in a homogenous environment—they are not forced to work in isolation. The ability to run simultaneous workflow streams in separate OpenShift projects fits extremely well with Agile delivery. Different clusters can be used to isolate production from non-production workloads. While each environment may be sized differently, they are identical in their software makeup. This drives up quality by eliminating version and configuration drift across environments.

As we mentioned in the introduction, we still see a place for less Agile approaches to delivery (including the use of Waterfall). Let us now look at the characteristics of different projects that can determine what approach is best.

Decision-Making Contexts

In 2020, Red Hat produced an eBook entitled *Transformation takes practice*.[4] This was written in response to a question asked time and again by business leaders: Why are so many digital transformation efforts failing? In the eBook, Mike Walker, Global Director of Red Hat Open Innovation Labs explains: "*In complex sociotechnical systems, it is a group of people, not individuals or managers, who can create innovative change. These groups must tune the system through a perpetual cycle of probing, sensing, and responding to outcomes*."

To explore that cycle of **probing, sensing, and responding** to outcomes, let's introduce a very helpful framework that compares this approach to alternative approaches used in different systems.

The Cynefin Framework

The **Cynefin framework** was created in 1999 by Dave Snowden when he worked for IBM Global Services. *Cynefin* is the Welsh word for *habitat*, and the framework offers five decision-making contexts or *domains*:

- *Clear* (known until 2014 as *simple*, but more recently renamed *clear* by Snowden)
- *Complicated*
- *Complex*
- *Chaotic*
- *Disorder*

4 https://www.redhat.com/en/resources/transformation-takes-practice-open-practice-library-ebook

The framework is intended to help managers identify how they perceive situations and make sense of their own and other people's behavior. The framework draws on research into systems theory,[5] complexity theory,[6] network theory,[7] and learning theories.[8]

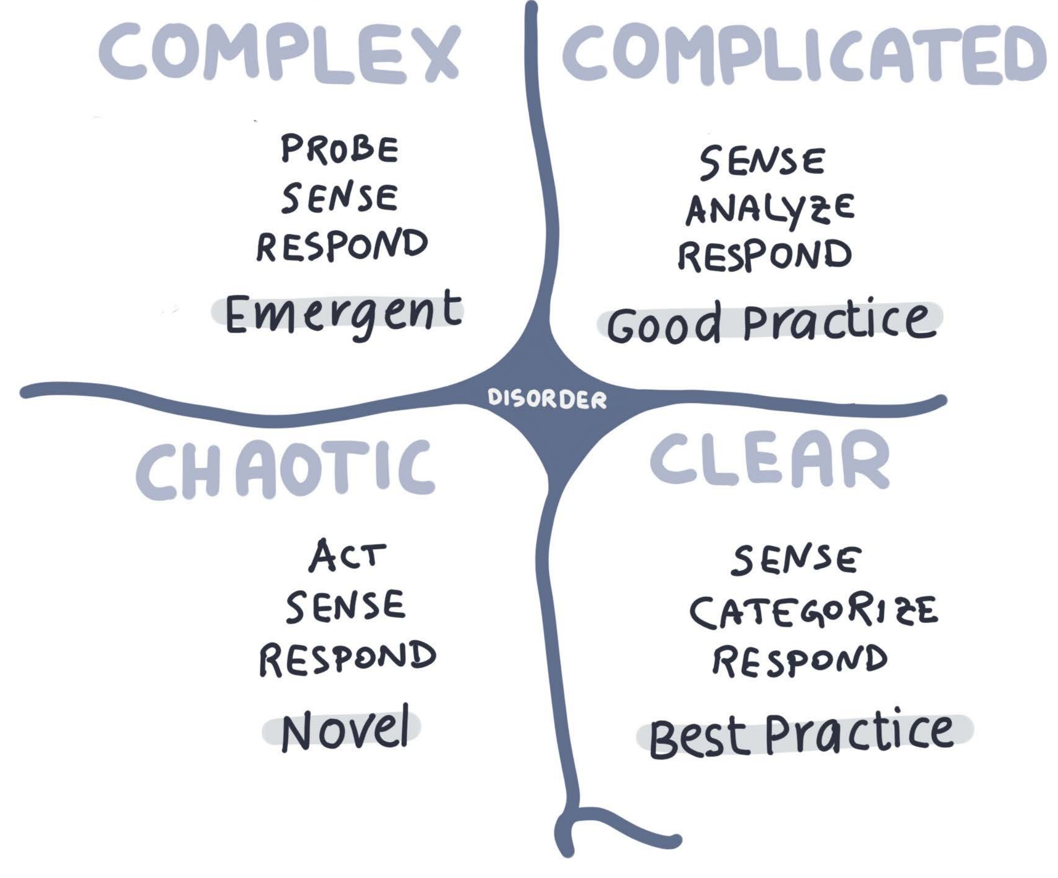


Figure 12.7: The Cynefin framework

The *clear*, or the *obvious* domain represents the *known knowns*. This means that there are rules in place (or best practice), the situation is stable, and the relationship between cause and effect is clear—if you do X, Y is expected. This is the domain of legal structures, standard operating procedures, and practices that are proven to work.

5 https://en.wikipedia.org/wiki/Systems_theory

6 https://en.wikipedia.org/wiki/Complexity_theory_and_organizations

7 https://en.wikipedia.org/wiki/Network_theory

8 https://en.wikipedia.org/wiki/Learning_theory_(education)

The *complicated* domain consists of the *known unknowns*. The relationship between cause and effect requires analysis or expertise where there is a range of right answers. The framework recommends *sense–analyze–respond*, that is, first assess the facts, then conduct analysis and use that to apply the appropriate good operating practice.[9]

The *complex* domain represents the *unknown unknowns*. Cause and effect can only be deduced in retrospect, and there are no right answers. "*Instructive patterns ... can emerge*," write Snowden and Boone, "*if the leader conducts experiments that are safe to fail*." Cynefin[10] calls this process *probe–sense–respond*.

In the *chaotic* domain, cause and effect are unclear. Events in this domain are "*too confusing to wait for a knowledge-based response*," writes Patrick Lambe. "*Action—any action—is the first and only way to respond appropriately*."[11] In this context, managers must *act–sense–respond*, *act* to establish order, *sense* where stability lies, and *respond* to turn the chaotic into the complex.

Most of the work we do with our customers sits in the complex and complicated domains. To differentiate between these two domains further, let's look at a helpful example used in the *Transformation takes practice* eBook about the difference between a Ferrari and a rainforest.

The Ferrari and the Rainforest

A recent piece of research by Harvard Business Review[12] shows that 80% of business leaders say their digital transformation efforts are ineffective. For organizations pursuing digital transformation, one key step is recognizing the difference between complicated and complex systems. Think of a Ferrari. It is made of thousands of components, but their sum and how each interacts with the other to create the whole are fundamentally knowable. Diagnosing and repairing a malfunctioning Ferrari requires you to sense, analyze, and respond to feedback and learning. A Ferrari is a complicated system.

9 https://hbr.org/2007/11/a-leaders-framework-for-decision-making

10 https://hbr.org/2007/11/a-leaders-framework-for-decision-making

11 Lambe, Patrick (2007). Organising Knowledge: Taxonomies, Knowledge and Organisational Effectiveness. Oxford: Chandos Publishing, 136.

12 https://www.redhat.com/cms/managed-files/cm-digital-transformation-harvard-business-review-analyst-paper-f22537-202003-en_0.pdf

Now think of a rainforest. It is a dynamic ecosystem with billions of interacting organisms and elements. We understand those dynamics at some level, but they are essentially unpredictable, with cause and effect only evident in hindsight. The problems threatening rainforests would be best addressed through probing, sensing, and responding. In other words, it requires continuous cycles of hypotheses, experimentation, and measurement to arrive at desired outcomes. A rainforest is a complex system.

Today's large organizations behave more like rainforests, not Ferraris. It is a complex sociotechnical system with many dynamics—both human and technical—at play, that resist quantification. This distinction is important because many enterprise leaders assume organizations are complicated systems in which analysis is the key to arriving at the best solution. But in complex systems, probing and sensing, or hypothesis and experimentation, in rapid and iterative cycles, are more effective in identifying the best solutions and ideas.

What has all this got to do with the delivery of applications such as the case study used throughout this book, PetBattle? There is a lot of terminology attached to delivery methodologies. When do we use Scrum versus Kanban? Is a Burndown chart just a reverse Gantt chart? How can we show value being delivered when using Waterfall? Ultimately, each method or approach is trying to achieve a similar goal: the completion of a feature set in any given product that can be released to our end users.

Some good starting guidelines when trying to understand which methodology to use include the following:

- Waterfall works best for projects that are completed in a linear fashion, where it is clear to *sense-categorize-respond* as best practice. So, following a standard operating procedure to install and test a software product will be a one-time-only piece of work and will not benefit from agility or iteration. There will be no need to go back to prior phases of work. This is often seen as a traditional way of working.
- An iterative Agile framework such as Scrum is concerned with getting work done by completing and showing small increments of work and inspecting and adapting to feedback. In other words, we are taking the *probe-sense-respond* approach articulated in the complex domain of Cynefin. Other Agile frameworks such as Kanban are primarily concerned with continuous delivery. The highest priority item is done next, and a fixed chunk of work is pulled off the backlog into a smaller stream of work. This also supports the *probe-sense-respond* mindset.

Let's get back to our visualization and mental model that we have been using throughout the book using the Mobius Loop.

When Does a Mobius Loop Mindset Make Sense?

One of the reasons we love the Mobius Loop is because we can use it to explain any project, program, or work we've ever been involved in.

As it is framework-agnostic, it does not demand Scrum or Kanban. It does not stipulate how many iterations a team or organization should go through, around one or both of the loops. It complies with any of the four established domains of Cynefin.

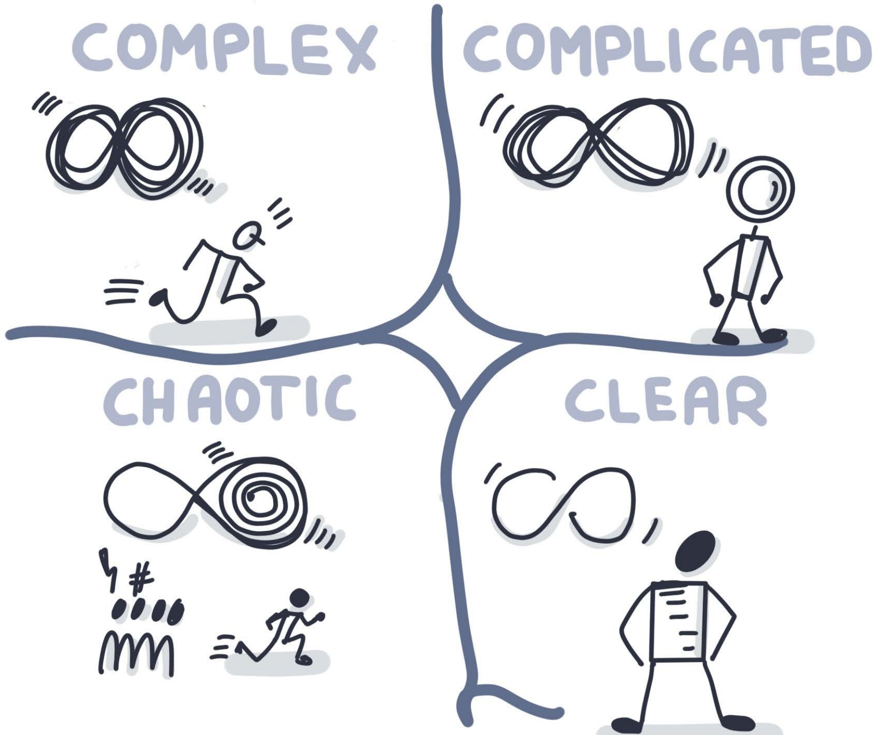


Figure 12.8: The Cynefin framework and Mobius Loop

Projects in the **complex** domain fit well with an Agile approach. Visualized on the Mobius Loop, this means starting on the Discovery Loop and establishing the first set of Target Outcomes. The Options Pivot involves refining what to discover or deliver next and deciding on which options to deliver. The Delivery Loop creates a design and build for a small increment, runs that increment in production to collect data, and takes measurements before learning and reflecting on the outcomes achieved. There is then an adaption point to either go through another loop of Discovery, do another loop of Delivery, or return to the Options Pivot to reprioritize and pivot. This cycle will continue indefinitely as long as there are product delivery and/or operations ongoing. There will be a growing desire, through continuous improvement, to speed up the iterations of Discovery and Delivery.

Projects in the **complicated** domain fit well with an **Agile** approach. Visualized on the Mobius Loop, this means starting on the Discovery Loop and establishing Target Outcomes. This may be slower and more detailed, given projects in the complicated domain benefit from sense and analysis before a response. A similar pattern will be experienced for complex projects through Options into Delivery and then either a return to Discovery, Delivery, or Options.

Finally, projects in the **chaos** domain need to act quickly and move out of that domain as quickly as possible. That may mean starting on the Delivery Loop with the urgent act and, as the level of chaos reduces, start to re-group and organize options and a loop of Discovery. Once the shift to the Discovery Loop is completed, the project moves from one of chaos to more of a complex project. Most application development these days would be considered complex.

So, where does an application like PetBattle sit in the Cynefin framework, and what is the best delivery approach to consider?

PetBattle—Complex, Complicated, or Clear?

Our PetBattle customers cannot use any of our innovative feature ideas until we build, test, and deploy our code into a production environment. The old hobbyist application was built all in one go and one release. Should the new PetBattle organization continue this approach?

The team walked the walls of all the artifacts produced during their Discovery Loop and options. They talked about what might be the best way to deliver all these great ideas. Some of the comments made during this session include:

- During the Discovery Loop explained in *Section 3*, *Discover It*, we only really explored in any depth a small subset of feature ideas, that is, those connected to the *Increase Site Engagement for Uploaders* part of the Impact Map. That was where the investors invested most of their Donal Dollars! We'll eventually need to consider other impacts or dive deeper into this one when we see what happens with our early feature builds.
- There are a lot of pink sticky notes on the Event Storm—these represent questions, assumptions, and other unknowns that we'll need to get answers to and eventually remove.

- The Event Storming done to date has focused on *the one where Mary enters the daily tournament and wins a prize*. We'll need to do more Event Storming as we consider new features and/or answers to the unknowns.
- We have some good target outcomes and some great hypothesis statements behind all the options. But what if the needle doesn't shift and those hypotheses are proven false? We'll be back to the drawing board.
- When we refined our options in *Section 4*, *Prioritize It*, on the Value Slice board, we identified several experiments we need to run. Some will be run as A/B tests, feature toggles, and canary launches, while some require UI prototyping and technical spikes to run before we can progress.
- The Empathy Map we produced from interviewing Mary was great. But it showed how many things we hadn't thought of about our users. What will happen when we Empathy Map other users? Or, when we start to meet advertisers and partners and capture Empathy Maps on them? What will they say?
- There remain a lot of technical unknowns and learning to be done on the platform.

There are many known unknowns and probably even more unknown unknowns! With so many unknowns, how do we deliver these changes in a way that ensures quality (no breakages to a PetBattle Tournament!), speed (can the team deliver changes quickly and on time?), and efficiency (can the team repeatedly deploy changes cheaply and easily)?

PetBattle, like most user-centric applications, is a complex sociotechnical system with many dynamics, both human and technical. With continued emergent practice, we can now move to a *probe-sense-respond* mindset as we approach our first (of many) iterations of delivery.

Taking a complex product into the first iteration of delivery means we have to decide what, from our list of options, we are going to deliver first. We also need to check whether the team are ready to work on the selected items and are confident they have the right amount of information and understanding to deliver them. This leads to another foundational practice to define and assess readiness to deliver.

The Definition of Ready

Jeff Sutherland, co-creator of the Scrum framework, called out that one of the main reasons that many Scrum projects fail is due to teams working on items that are simply not ready to be worked on. They are either too ambiguous, not understood by business and technical stakeholders, too big, or they lack ambiguity as to the scope of the item in question.

Many teams have chosen to adopt the practice of having a **Definition of Ready** to mitigate this risk. The Definition of Done practice has been popular with Agile teams for many years—we will explore that later in this chapter. The Definition of Ready has been less utilized. A good way to look at this, like every practice in the Open Practice Library, is as a tool that can be taken out of the toolbox to address a problem. If a team is struggling to get work done because of ambiguity and a lack of shared understanding before even starting it, adding a Definition of Ready may improve the team's success in the future.

The Definition of Ready is a practice used by a team to set their criteria and expectations with others as to what constitutes a Product Backlog item that is ready to be worked on. The resulting artifact is collaboratively created, maintained, and enforced by the team, and publicly displayed for the team members and others to easily reference. It is often informed by experience, like all other artifacts generated from the practices in this book. It is a living, breathing artifact that will improve over time with learning and experience. We often find discussions that happen during Retrospectives lead to improvements being made to the Definition of Ready. For example, when a team gets blocked and does not manage to deliver the items they pulled into their sprint, they might ask themselves what they could have done differently to prevent the items from getting blocked in the sprint. If there is a common pattern emerging and an improvement process that can be put in place to prepare future items ready for delivery sprints, the criteria can be written as a Definition of Ready criteria.

The Definition of Ready often starts with the act of writing user stories and/or acceptance criteria. When we look at Product Backlog items, they are often quite high-level, for example, **Log In**, **Choose Item**, **Submit Order.** At first glance, you might think these items are clear in scope and could be delivered according to expectation. However, a few simple questions from a Development Team member opens up the world of ambiguity that exists:

- When you say log in, do you mean log in with a username and password?
- What happens if I don't have a username registered?
- What happens if I put in the wrong password?
- What should happen immediately after I log in?

The practice of writing user stories has become very popular with Agile teams because it turns a feature idea into a conversation (or, at least, a promise to create a conversation).

Mike Cohn of Mountain Goat Software has driven the popularity of user stories and explains how the practice helps us *shift the focus from writing about requirements to talking about them. All Agile user stories include a written sentence or two and, more importantly, a series of conversations about the desired functionality.*[13]

User stories tend to follow the template: as a <type of user>, I want <some goal> so that <some reason>.

So, after some brief conversation between the Development Team members, business stakeholders, and the Product Ownership, the **Log In** feature could be re-written as, "*as a retail banking customer, I want to log in with my username and correct password so that I can be presented with my latest bank balance.*" The conversation may also have driven the splitting of this feature into several other user stories that focus on how to get a username/password, what should happen if incorrect credentials are supplied, other information and options available after logging in, what other types of customers should be able to see and do, and so on.

The user story practice facilitates three outcomes, also known as the three Cs:

1. It generates a **Card**—the user story is short enough that it can be written on a small index card (or a digital equivalent).
2. It generates a **Conversation**—both in writing this user story and capturing unknowns or further clarifications needed, yet provides enough direction for working on the item.
3. It provides **Confirmation** of the scope and expectation of this individual item that will deliver a small piece of value to the end user.

13 https://www.mountaingoatsoftware.com/agile/user-stories

The INVEST criteria is a good pattern to check against user stories.

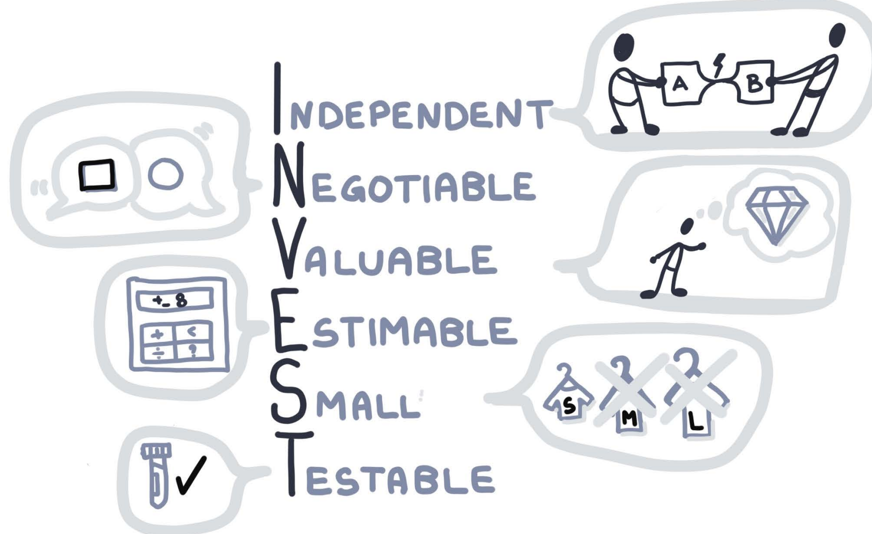


Figure 12.9: INVEST criteria

You'll notice that the act of re-writing a feature as a user story helped answer some of the example questions, but not all of them. While we could write many user stories simply to answer every question a developer might have, this may result in them being too granular and developers grouping them together to write code. So, a second popular practice to prepare items for delivery is the act of writing acceptance criteria.

Acceptance criteria is a further elaboration of understanding and often written on the back of the index card holding a user story. Acceptance criteria are the conditions of satisfaction a Product Owner and stakeholders will have when the item has been completed. These are low-level binary points of clarification – basically, a checkbox list that the Product Owner may go through to convince themselves that a feature has been delivered to meet business expectations. So, some acceptance criteria for our **Log In** feature may include the following:

- There is a textbox for username and password.
- Beneath, there is a hyperlink to create a new username.
- The username will be displayed in plain text when typed in.
- The password will be masked with asterisks when typed in.
- If the username/password is correct, the Customer Details page will be presented with the bank balance at the top of the page.
- If the username/password is incorrect, an error message will appear.
- If an incorrect password is entered three times in a 15-minute period, the session will be locked out for a further 15 minutes.

Each of these items represents small, standalone tests (which can be automated). They are specific to the functionality being built. Later in this chapter, we'll show how another practice, the Definition of Done, is used to drive other types of testing.

The act of writing user stories and acceptance criteria helps generate shared understanding across all team members, Product Owners, and stakeholders. They are often practices used in Product Backlog Refinement sessions, which were discussed in the previous chapter.

If a team sees the value in these practices, they may choose to add them to their Definition of Ready so that they always go through the motions of writing user stories and acceptance criteria for a Product Backlog item, so that it will be ready to be worked on. If an item does not have acceptance criteria, it is NOT ready and the team either needs to do some refinement and write the acceptance criteria, or demote the item on the Product Backlog and find alternative items that are ready.

Now, it's important to call out some misuses and problems that can come about through having a Definition of Ready. First, this is not an opportunity to introduce lots of pre-development phase gates. Definitions of Ready should not be about writing long specifications, having design documents signed off, having architectural review board approvals, and so on. The Definition of Ready is about giving the team the confidence that, if they take some work into a Delivery Iteration, they will get it done.

Secondly, the Definition of Ready is not a mandatory practice for teams. Like all practices in this book, it is a tool that is useful to try and, if the team does experience problems with getting work out of the Delivery Iterations they take in, it would be recommended to try using the Definition of Ready practice.

When we look at a team's Definition of Ready, we look for statements that are focused on the team and their comfort or confidence, as these are clearly the criteria that bring the team to the fore.

Let's look at the Definition of Ready criteria our PetBattle team put in place.

PetBattle – Definition of Ready

The PetBattle team met a few days before their first Delivery Iteration. They looked at the Product Backlog that had emerged from Value Slice and Discovery Loop practices. There was clearly some discomfort and misalignment in exactly what the feature was going to deliver and involve. So, the team booked a series of 40-minute Product Backlog Refinement sessions.

During the first session, they brainstormed their Definition of Ready and agreed this would be used in all future Product Backlog Refinement sessions to validate and assure that they would be ready to work on an item.

The Definition of Ready included:

1. The team has written and agreed on acceptance criteria with the Product Owner and, where it makes sense, business or technical stakeholders.
2. The team feels the item can be delivered within a few days of development.
3. The team has collectively sized the item using a relative sizing technique.
4. The team understands where the item fits into the existing or emergent architecture.
5. The team has seen and has access to UI and technical research.
6. Any dependencies identified by the team to complete the item have been satisfied or the team is confident that they can be satisfied within the iteration.
7. The team sees the value in doing this work.

The team agreed that they would regularly use the Fist of Five confidence vote or Roman vote to assess their comfort and confidence levels for the seven points above. They were unified in that they should never take items into a Delivery Iteration that were not ready. They would need lots of short Product Backlog Refinement sessions to keep the top backlog items ready.

To discuss the Definition of Ready with the wider community or to read more information or even improve the practice, have a look at the Open Practice Library page at https://openpracticelibrary.com/practice/definition-of-ready/.

Now we have some items **ready** to be delivered, let's look at the most popular Agile framework used for iterative delivery, Scrum.

Scrum

When you hear someone say that they are now doing agile, their team is using agile, or their organization has adopted agile, there's a very strong chance that what they are actually doing is some form of Scrum.

Scrum is not the same as agile. As mentioned earlier, Agile is an ideology. It is underpinned by those values and principles set out in the Agile Manifesto. Scrum is one of several frameworks that instantiates Agile and makes the principles a reality through a group of practices. It is also, by far, the most popular Agile framework.

It was created by Jeff Sutherland and Ken Shwaber in the 1990s and was inspired by Hirotaka Takeuchi and Ikujiro Nonaka when they published *The New New Product Development Game*.[14]

There have been numerous books written about Scrum and, as with any good framework, it has evolved and continuously improved with learning and experience. We would strongly recommend reading the latest version of the Scrum Guide as presented by Sutherland and Schwaber at https://scrumguides.org/. It is an artifact we recommend re-reading every now and again to refresh your memory and knowledge of this very powerful framework. This is especially true when updates are made and there are new releases of the framework.

We're not going to teach Scrum in this book as there are many better articles and books and, most importantly, the Scrum Guide can do that much better. What we are going to do is demonstrate how we have used Scrum in our engagements, how it fits into the Mobius Loop, and share some stories of good and bad Scrum adoptions.

One of the reasons we love Scrum is because of its simplicity. Often people confuse the many (now hundreds) of practices associated with Agile as being Scrum. In fact, Scrum has very few practices. We tend to describe them as 3-5-3 (which sounds a bit like a formation for a sports team!).

14 https://hbr.org/1986/01/the-new-new-product-development-game

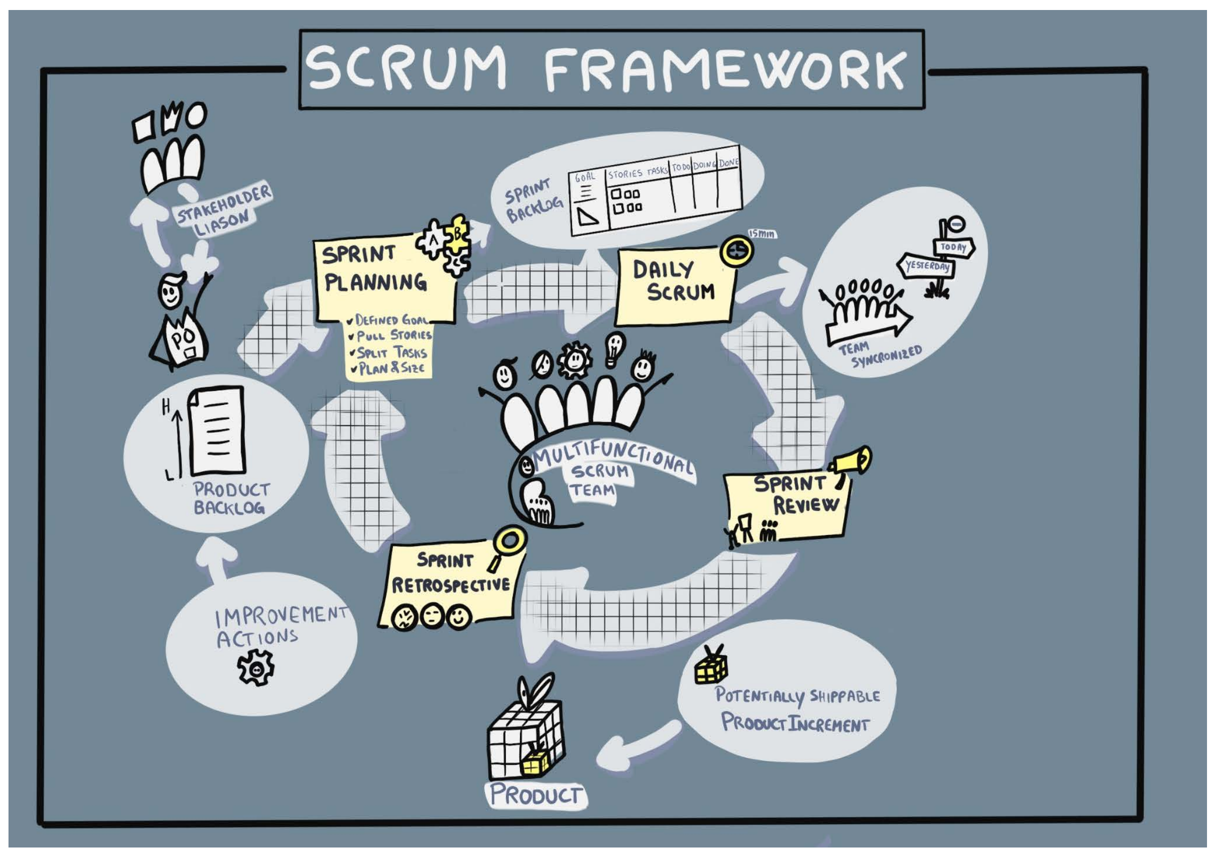


Figure 12.10: The Scrum framework

This Scrum picture can be best described by grouping the roles, events, and artifacts.

The 3-5-3 Formation

The 3-5-3 formation represents 3 roles, 5 events (previously known as ceremonies), and 3 artifacts. The three roles are the Product Owner, the ScrumMaster, and the Development Team. The five events are Sprint Planning, the Daily Scrum (often known as the Daily Stand-Up), the Sprint Review (sometimes called the Showcase), the Sprint Retrospective, and the Sprint itself as a fixed time-box container. The three artifacts are the Product Backlog, Sprint Backlog, and Product Increment. That's it! That's all there is to Scrum. Those three roles, five events, and three artifacts. All those other things you hear and read about, like Definitions of Ready and Definitions of Done, Burndown and Burnup, User Stories and Story Points, and so on are all great supporting practices, but not a part of the Scrum framework. Let's share a few recommendations and examples about each component of the Scrum framework, starting with the roles.

The Product Owner Role

In *Chapter 11*, *The Options Pivot*, we introduced the idea of Product Ownership, and strongly recommended watching Henrik Kniberg's *Product Ownership in a Nutshell* video15. If you haven't watched it yet, then please put this book down and go and watch it now. Even if you did watch it then, we'd suggest watching it again. It's 15 minutes of information that is pure gold.

"*The Product Owner should have been a key contributor and supporter throughout the Discovery Loop and Options Pivot*," explains Kniberg, their big focus being on enabling communication and collaboration between Development Team members and stakeholders. They convey the message of envisioned products and set priorities for the team by utilizing many of the Options Pivot practices.

In a Delivery Iteration, this communication is even more important. Having a Product Owner that can be the voice of the business, the voice of the user (even the voice of technology stakeholders), and be able to regularly converse with developers is a key ingredient in enabling high performance in Scrum teams. That means face-to-face interaction, negotiations, alignment, and driving shared understanding between the Development Team and the Product Owner.

Figure 12.11: Collaboration between a Product Owner and her Development Team

15 https://www.youtube.com/watch?v=502ILHjX9EE

The Product Owner needs to be regularly available to the Development Team, especially during Product Backlog Refinement to, for example, write and agree on acceptance criteria, and during Scrum events, such as Sprint Planning.

Figure 12.12: Development Team member clarifying his understanding with his Product Owner and Business experts

In the case study story in *Chapter 11*, *The Options Pivot*, *Experimenting with different Product Owners*, we mentioned that Product Owners should see their current role as one to self-destruct and not be needed long-term. They can make themselves obsolete by successfully facilitating the Development Team members, directly connecting them with stakeholders and users, and building those relationships to a level so that their facilitative role is no longer needed.

The ScrumMaster Role

We have not included a photo of a ScrumMaster in action because the ScrumMaster arguably should not ever be seen to be in too much frontline action. Their role is an enabling one. ScrumMasters should create an environment and a safe space that enables the Development Team to see all the action. Of course, this does not happen immediately, especially with a newly formed team or a team unfamiliar with Agile or Scrum. In these cases, they have to coach the team on the Scrum process and help move them towards a more productive environment.

Through great facilitation, great preparation of practices, and effective coaching of practices, the ScrumMaster should promote autonomy in the team, self-correction and continuous improvement, and self-organization. It is another role where the person in it should be trying to remove the need for themself. The ultimate autonomous, self-organizing, self-correcting team should no longer need to be facilitated or led.

The Development Team Role

As we have already seen, the Product Owner should aspire to remove the need for themselves because the Development Team is so well connected to users and stakeholders with strong psychological safety for two-way feedback and conversation between the two parties. The ScrumMaster should aspire to remove the need for themselves because they have facilitated an environment that is full of information radiation, very strong adoption of practices by a team that is fully autonomous, self-correcting, self-organizing, and continuously improving all the time.

That just leaves one role in Scrum that doesn't remove the need for themselves, the Development Team. The term *Development Team* may be a little confusing. Firstly, as we've mentioned several times in this book already, DevOps is about bringing down the wall between Development and Operations, yet the Scrum terminology seems to still lean heavily towards Development. In reality, this team should include T-shaped people with breadth and depth and, between all its members, should be able to design, build, test, deploy, operate, and maintain features from the Product Backlog all the way to production.

A great Development Team will most likely have been set up for success by a great ScrumMaster. Their workspace will be full of information radiators and artifacts generated by the Discovery Loop, Options Pivot, Delivery Loop, and foundational practices. Lots of care and attention will have gone into their surroundings, ensuring that the Development Team has been enabled and set up for the best possible success with as many foreseen and unforeseen obstacles removed. They will have organized time away from the computer, fed and watered the team, organized socials, and continuously monitored and improved the open culture and psychological safety of the team. The motivation levels, mood, and feeling of empowerment, autonomy, mastery, and purpose should be assessed and measured regularly (using practices such as the Social Contract, Stop-the-World events, Mood Marbles, and Real-Time Retrospective all introduced in *Chapter 4*, *Open Culture*).

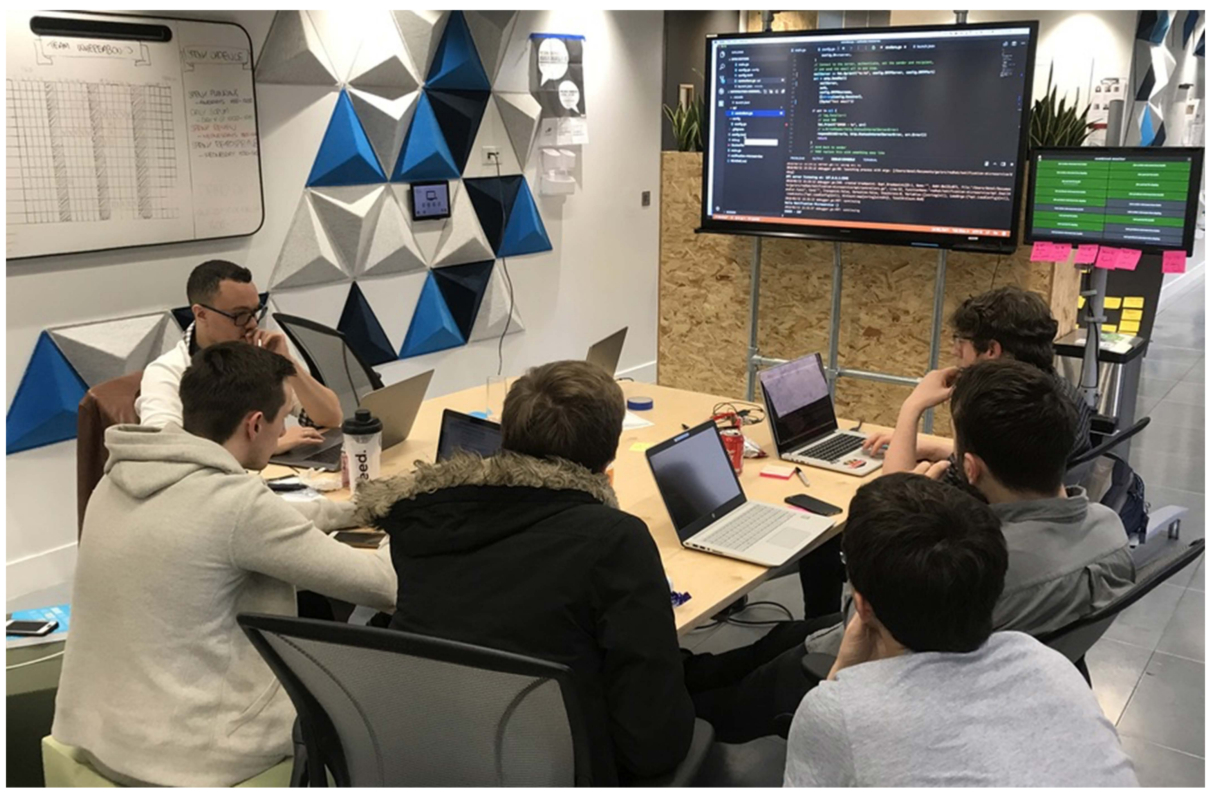


Figure 12.13: Cross-functional Development Team

Later in the book, in *Chapter 18*, *Sustain It*, we will look at larger multi-team organizations and some of the new patterns that help collaboration between the teams. But first, let's look at the artifacts that every team using Scrum should always have.

The Product Backlog Artifact

The term "backlog" is used frequently in Agile. If the team is using Scrum, it's important to recognize that there are different types of backlogs and they serve very different purposes. The Product Backlog shows all the work and ideas of the whole product, prioritized according to value by the Product Owner and available for the stakeholders to see; for example, the most prioritized features to be built into the product next.

In *Chapter 11*, *The Options Pivot*, we introduced the Product Backlog and showed how it emerged from Value Slicing and User Story mapping and was prioritized using practices such as Impact and Effort Prioritization, How-Now-Wow Prioritization, the Cost of Delay, and Weighted-Shortest-Job-First. From earlier chapters, we know that the Product Backlog items are all generated from practices and conversations on the Discovery Loop such as Impact Mapping, Event Storming, Human-Centered Design, Non-Functional Mapping, and Metrics-Based Process Mapping.

The Product Backlog is such an important artifact that it should be displayed prominently, accessible to everyone, and should regularly be used by Product Owners, Development Team members, and ScrumMasters—as well as other stakeholders, and anyone else interested in the next steps of this product.

The Product Backlog is living and breathing and can be updated at any time. All the practices that lead to Product Backlog items also have artifacts that are living and breathing at any time. So, a stakeholder at any time might have a new impact or deliverable idea that can go straight onto the Impact Map. The Development Team may gather for a time-boxed Event Storming session during their sprint to refine and answer some unanswered questions on the existing Event Storm, or they may start an Event Storm on a whole new area of the business. More user interviews, Human-Centered Design, and Empathy Mapping should continue. All of this will result in refined, improved, and new ideas that can be processed through Value Slicing, prioritized, and ending up on the Product Backlog.

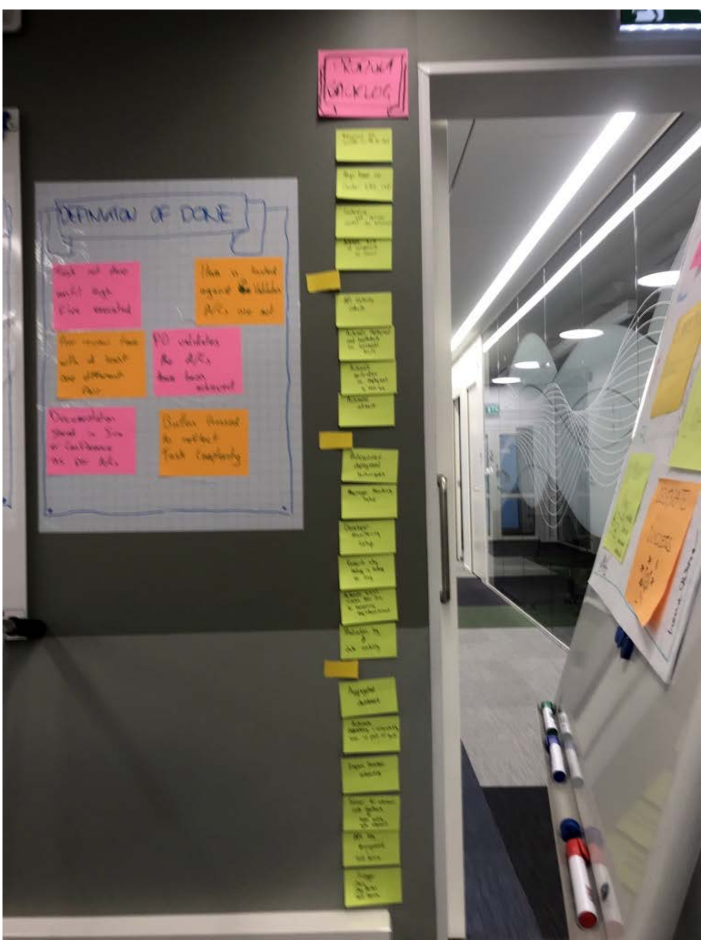


Figure 12.14: Product Backlog publicly displayed on a wall

The Sprint Backlog is a different artifact that is more micro-focused for a specific sprint or Delivery Iteration.

The Sprint Backlog Artifact

The Sprint Backlog is created during the Sprint Planning Event and shows all of the items the team has pulled from the Product Backlog into the Sprint Backlog (having assessed they are ready, according to their Definition of Ready). What results from a Sprint Planning session is a Sprint Backlog on a highly utilized and very important information radiator—the Sprint Board.

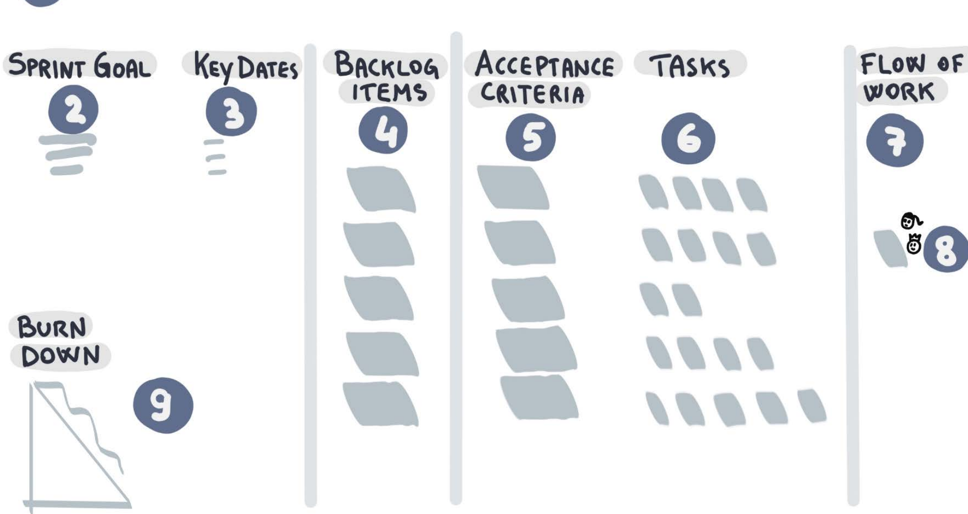


Figure 12.15: A Sprint Backlog on a Sprint Board

The Sprint Board is one of the artifacts that sees the most use during a Delivery Loop iteration and one of our favorite information radiators. In the preceding example, the following information is shown:

1. The Sprint number.
2. The Sprint Goal—a one- or two-sentence description of what the product should do in the next increment that it does not currently do. Ideally, this should link to the overall Target Outcomes agreed in the Discovery Loop.
3. Key dates—the Sprint Planning, Sprint Review, and Sprint Retrospective day, and the time required for this sprint.
4. The Sprint Backlog items that have been pulled from the Product Backlog. There is a one-to-one matching and should be easily traceable from Product Backlog to Sprint Backlog.
5. The Acceptance Criteria that have been agreed for each Sprint Backlog item.
6. The low-level tasks that are needed to deliver the features according to agreed acceptance criteria. These are color-coded to radiate the cross-functional aspect of feature development, with different colors representing frontend and backend development, data, infrastructure, operations, and design.
7. Columns to represent the flow of work on each task—Sprint Backlog, In Progress, Blocked, Ready for Review, and Done.
8. Avatars to show which pairs or mobs are working on a given task in the In Progress column.
9. A Burndown chart,[16] a popular visualization practice showing the pace of—in this case—tasks getting plotted against the timeline (in days) of the sprint.

16 https://openpracticelibrary.com/practice/burndown/

The following template is available in the book's GitHub repository[17] and can be downloaded to produce the same Sprint Board virtually.

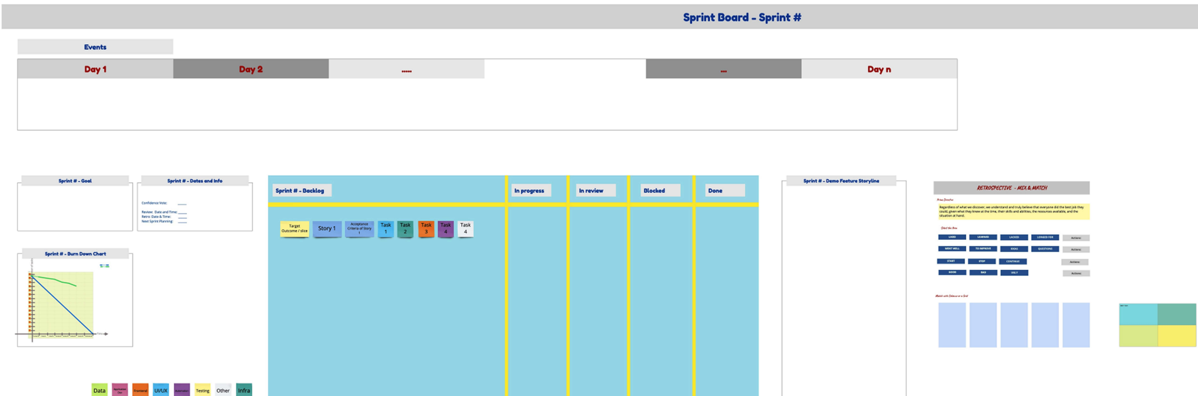


Figure 12.16: Digital Sprint Board template for a distributed team

One of the most impressive and powerful aspects of the Sprint Board is that it provides the (almost) final piece of traceability in the life of developing a given feature. If you walk-the-walls with all Discovery, Options, and Delivery artifacts available, you should be able to see a low-level task (typically a development activity performed by a pair of Development Team members that takes a few hours) and see its connection to:

1. The overall feature/user story and the acceptance criteria on the Sprint Backlog.
2. Where that feature sits on the Product Backlog.
3. How that feature was prioritized using Value Slicing and other prioritization tools.
4. How that feature fits into the event-driven system and overall business process on the Event Storm.
5. The measurable impact that was hypothesized with the feature on the Impact Map.
6. The actors captured on the Impact Map the features can be user-tested with.
7. The overall goal captured on the Impact Map and in Target Outcomes that this feature should be aiming to achieve.

The final piece of traceability is the working software, which is the third increment of the Scrum framework.

17 https://github.com/PacktPublishing/DevOps-Culture-and-Practice-with-OpenShift

The Product Increment Artifact

The seventh principle of the Agile Manifesto states that working software is the primary measure of progress. When the original writers of the Agile Manifesto met approximately ten years after the original manifesto was written, they ran a Retrospective event to see if their learning and experience would drive any updates to the values or principles. The general consensus was that it still held true, and nothing needed to change other than, in some cases, underlining the importance of working software.

Many Scrum teams fall down at this point and cannot show working software at the Sprint Review event. Even if they do have partially built software to show at this event, that is not enough. There should ALWAYS be working software to show.

Show Me the Product!

One of the many techniques we use to create awareness and share ways of working between our staff at Red Hat Open Innovation Labs is to share short write-up summaries during a residency engagement.

Often a similar summary is shared with customers with links to artifacts, photos of practices in action, and feedback captured or actioned.

Mark O'Callaghan is an Open Innovation Labs Engagement Lead based in New York. In January 2018, he was leading an engagement for UNICEF[18] focused on data science with an emphasis on visualizing data from a couple of legacy applications to support UNICEF in Columbia.

Mark would share weekly email updates of this engagement with a Red Hat community of Labs strategy and other interested parties. In his weekly summary, he would always include a link to the latest version of the application. This was working software running on OpenShift.

This is an awesome example: it is always a good idea to have a Product Increment, a link to an app that people can click on – to look at, use, and provide feedback on.

---

18 https://www.redhat.com/en/proof-of-concept-series

The final installation of our tour around the Scrum framework and how we use it is to explore five Scrum events – Sprint Planning, the Daily Scrum, the Sprint Review, the Sprint Retrospective, and the Sprint itself.

The Sprint Planning Event

In the previous part of this chapter, we showed the Sprint Board, an amazing information radiator that has a great amount of rich information about what the teams are currently doing and will be doing in this iteration of the Delivery Loop. The Sprint Planning Event is the event that generates this artifact. The Scrum Guide details how a Sprint Planning event should be run, who should attend, how long it should take, and so on. The following is how we tend to run it.

First, we establish a goal. This should be a one- or two-sentence summary of what the focus of the sprint is. What will the product do in a week's time that it does not do now? What outcome might it achieve? This is often a great conversation to have with the Product Owner holding the pen by the whiteboard ready to articulate his or her motivations in writing. But he or she always has tremendous support from the Development Team and a huge number of artifacts. It's useful to reflect on the Target Outcomes we established on the Discovery Loop, which should be nearby and visible. It's useful to look at the Product Backlog. If it has been well refined and prioritized, the items at the top should offer some clues as to what we might work on in this next iteration. Of course, it's useful to reflect and be reminded of all the wonderful information on other artifacts, Empathy Maps, and other user research, Impact Maps, Event Storms, Retrospective feedback, actions, and so on. Sometimes, we will draft the goal, then clarify and confirm it later on in the Sprint Planning event.

Next, the team will start to pull items from the Product Backlog working from the top – the Product Owner or Project Manager does not do this. It's important to move away from a push phenomenon associated with traditional delivery models and for the team to be engaged in pulling the items needed to achieve the goal.

When pulling items and adding them into the Sprint Backlog, the team should confirm that the items are indeed ready to be worked on by consulting their Definition of Ready. If, like us, you work in short one-week sprints, it can be difficult to find enough time in the week to do enough refinement to have all Definition of Ready criteria satisfied before Sprint Planning. So we often will take some time to ensure items are ready, acceptance criteria are written and understood, and the team is generally feeling comfortable and confident about the item in question.

The third step is to decompose the items into low-level tasks. We capture all the things we need to do to deliver this feature to satisfy the acceptance criteria and to meet the Definition of Done (which we'll cover shortly). This is a great opportunity to follow another diverge-converge pattern or liberating structure as introduced in *Chapter 4*, *Open Culture*, like 1-2-4-all. Getting everyone thinking about the work needed to deliver this increment and visualizing this is a really powerful aspect of this approach. This is detailed design on steroids! This is fixing bugs before we even write any code! As we saw in the example Sprint Board, using different-color sticky notes to reflect different types of tasks such as frontend, backend, data, design, infrastructure, operations, and so on, visualizes the cross-functional nature of the team.

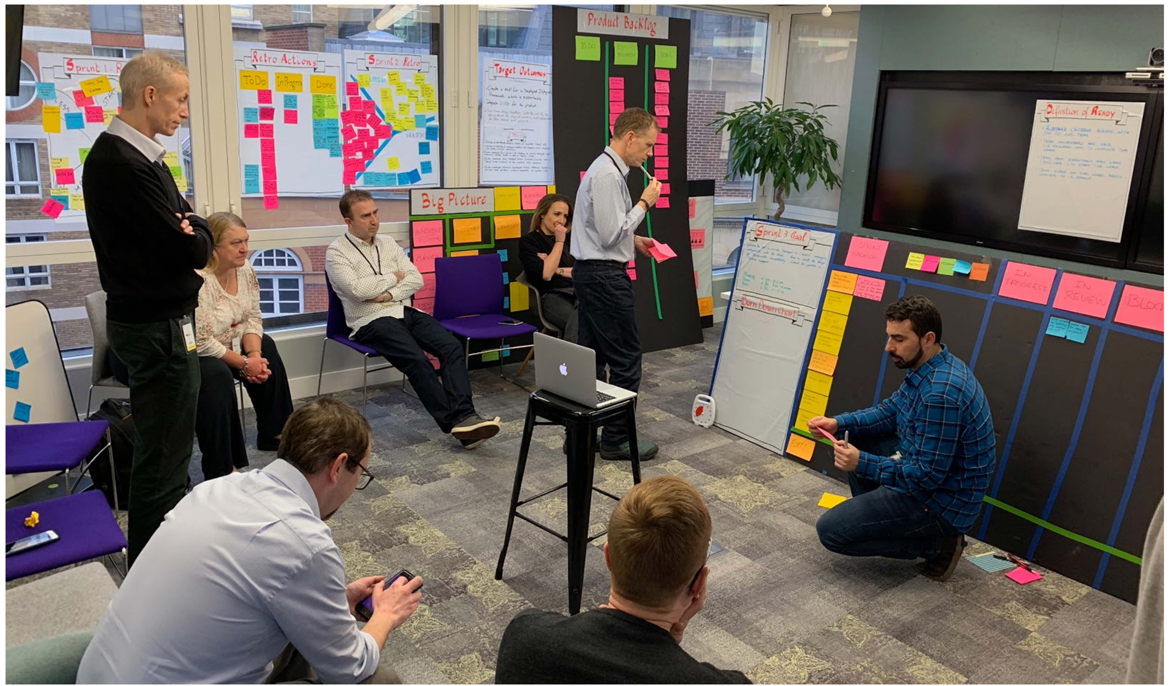


Figure 12.17: Product Owner Paul continues to iterate on his goal while a MakMakMakMak Development Team member pulls items from the Product Backlog into the Sprint Backlog

After much collaboration, merging together of individual ideas for tasks into pairs, and then into pairs of pairs, and finally into a full group, we should have a Sprint Board, a complete visualization of the week ahead. The team should be fully aligned on the work that needs to be done. Now, can we achieve this? We finish the Sprint Planning event with a Confidence Vote – how confident are we? Can we meet the Sprint Goal by delivering the features in the Sprint Backlog? Can we satisfy all the Acceptance Criteria agreed and complete all tasks by the date of the Sprint Review on the board? Team members vote with their fingers—0 or 1 means very low confidence and we need to re-scope or discuss what it will take to increase confidence. A group consensus of 4 or 5 is what we want, and this indicates confidence from the outset.

Figure 12.18: Confidence Voting on achieving the Sprint Goal

As you can see from the photographs, those big 8-foot by 4-foot foam boards, which we explained in *Chapter 5*, *Open Environment and Open Leadership*, are incredibly powerful as Sprint Boards. They are light and portable and can easily be transported to different areas of the build, for example, to the Daily Scrum or to a Daily Stand-Up event.

Iteration or Sprint Planning has its own entry in the Open Practice Library and is an excellent resource to find out more, contribute improvements, and discuss with the wider community the use of this practice. You can find this page at https://openpracticelibrary.com/practice/iteration-planning/.

The Daily Scrum Event

This event is the most popular Agile practice and is often how Agile teams start adopting Agile practices. Just meeting for 15 minutes each day to synchronize activities can be so valuable in keeping the team connected and driving autonomy.

We tend to find the team **talking to the board** works well. As each member summarizes what they worked on yesterday (to take the team towards its sprint goal), what they're planning to work on today (to take the team to its sprint goal), and any blockers or impediments (preventing the team from meeting its sprint goal), the board can be updated to ensure it is radiating the same information as what the team members are saying. Are the avatars of individuals showing the task they're working on? Have we agreed on what pairs or mobs are going to form for upcoming tasks? Are blockers clearly radiated in the Blocked column?

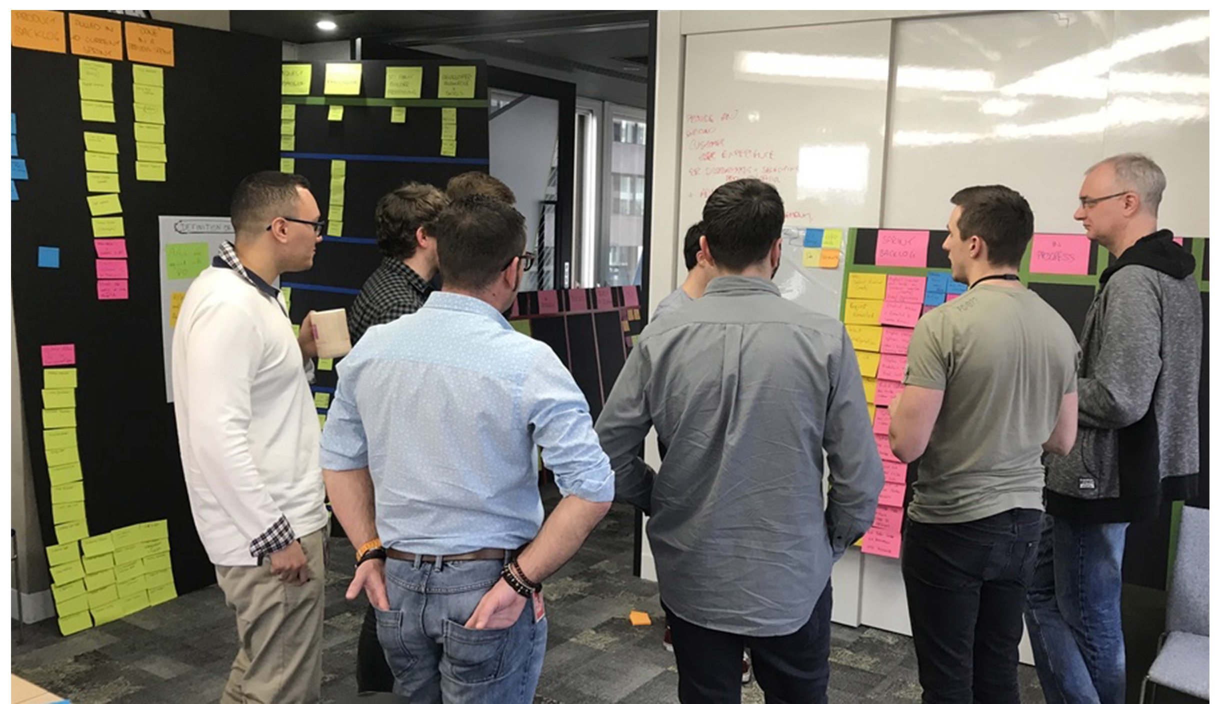


Figure 12.19: Daily Scrum practice

The Daily Scrum is often an event where **Scrum-but** antipatterns start to emerge. Scrum-but represents activities or behaviors that the Scrum team exhibit that do not conform to the Scrum framework. For example:

- We use Scrum but all of our management shows up to Daily Scrums to get status updates.
- We use Scrum but our Daily Scrums tend to run to between 30 and 60 minutes.
- We use Scrum but we vary the time of our Daily Scrums and who attends.

These are all anti-patterns to an incredibly simple practice that is purely for the Development Team to synchronize with each other and highlight any blockers. Jeff Sutherland often says if your Daily Scrum takes more than 15 minutes, you're doing it wrong.

If you have experiences to share about this practice or suggestions to improve it further, please go to the Open Practice Library page at https://openpracticelibrary.com/practice/daily-standup/.

Let's fast-forward to the end of the Sprint and look at a very valuable practice, the Sprint Review Event or Showcase.

The Sprint Review Event

The Scrum framework calls this event the Sprint Review and provides an opportunity to show the new Product Increment and invites feedback. This Open Practice Library entry is called the Showcase.

Some hints and tips we've picked up on the Showcase are as follows:

First, invite the world! This is such a wonderful opportunity to showcase the product increment to valuable stakeholders and users. It's helpful to schedule this event at the same time and place as every iteration and also make it accessible to join remotely – investing in good video conferencing systems can really improve engagement.

This is a great opportunity to reconnect with existing stakeholders and contributors to the Discovery Loop practices. Look at the actors on your Impact Map. Look at the people for whom we produced an Empathy Map. Can we get those folks back in to see the emerging product and provide further feedback?

The Showcase should not just be a showcase of shiny new functional features, but of everything the team has worked on and wants to show off to the wider stakeholder community. Often our Showcases will include:

- CI/CD infrastructure (and perhaps a demonstration of delivering a small code feature and how that triggers the CI/CD pipeline).
- An update on UI/UX including any new user research or user testing performed, the evolution of prototypes, and the learning that has emerged.
- The latest iterations of the Big Picture and emerging architecture.
- Showing how the teams have worked and the practices they have used. This often includes what we call the Practice Corner—an information radiator made up of the Mobius Loop and Foundation, with all the practices that the team have been using.
- An updated Product Backlog with words from the Product Owner about what to expect next.
- A review of the overall Target Outcomes and Enabling Outcomes and whether this latest Product Increment has shifted the needle on the measures.
- Running the Showcase in the same physical workspace as where the team normally works, opening the options for a walk-the-walls and inspection of all Discovery, Options, Delivery, and Foundation artifacts and information radiators.

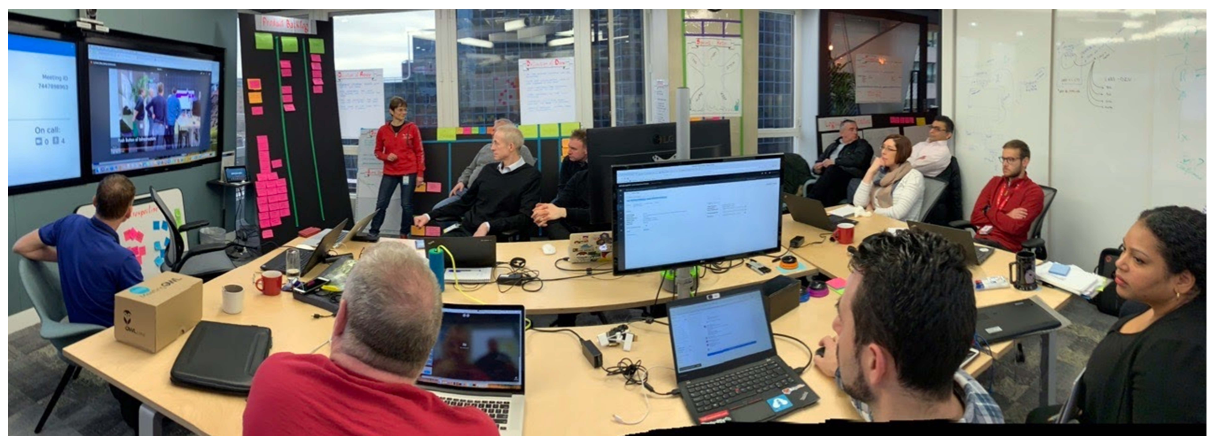


Figure 12.20: Sprint Review/Showcase

We will explore much more about the measurements and learnings, including those captured from Showcase events, in the next chapter. The Showcase should not just be limited to an event. We need to seek other creative ways to show the world what we've delivered and how we've delivered it in this latest iteration of the Delivery Loop.

As mentioned earlier, our Engagement Leads tend to do a short write-up to summarize progress and share the latest Product Increment and other artifacts. They send this via email to all stakeholders and anyone interested in the product and/or team. Visually, this can be enhanced by including a few photos of the event, or a short video montage of the team in action and the product evolving from the last sprint. Our engagement nearly always produces a series of weekly videos, which, when watched one after the other, produces an amazing story.

As we explained with the Product Increment Artifact, having working software is the most important ingredient of the showcase. Being able to provide a link to the application and inviting the Showcase audience to touch and feel the user interface is very engaging. Having a laptop in the workspace that always has the latest increment of the working software product running is great for stakeholders visiting the workspace, walking the walls, tracing feature history through all the practices, and seeing them for real in the product.

Finally, Showcases are limited in value if we do not get feedback from them. Great ScrumMasters and Engagement Leads will be creative around ways to do this. Don't just limit this to an *any questions or feedback* session at the end. Allow online feedback in a Google document, or open a feedback channel on Slack or via SMS. Invite value voting from stakeholders or use Net Promoter Score to capture fresh metric-based feedback on how likely stakeholders are to recommend the product and/or team.

When WOULD We Have Uncovered This In a Traditional Mode of Delivery?

I was working with a team a few years ago to build a new shopping portal for a telecom company in Ireland. We used a Design Thinking framework to gather a list of features to develop a product based on Empathy Mapping, scenario maps, pain points, and opportunities captured. Getting information directly from users was gold, and we distilled all of our learning from them into three Target Outcomes for the initial release of the product. We built up a Product Backlog of features to build up our portal product.

Then, after five weeks of Discovery, we adopted Scrum. Two-week sprints each delivered an increment of the product. Each Sprint included a Sprint Planning event, Daily Scrums, a Sprint Review, and a Sprint Retrospective.

If you look at the photograph, it is actually from the Sprint 2 Review! I chose this photo because seconds after I took it, the lady just out of the shot called out how we'd completely missed some key complexity in pricing administration. It was something we had not considered until this moment.

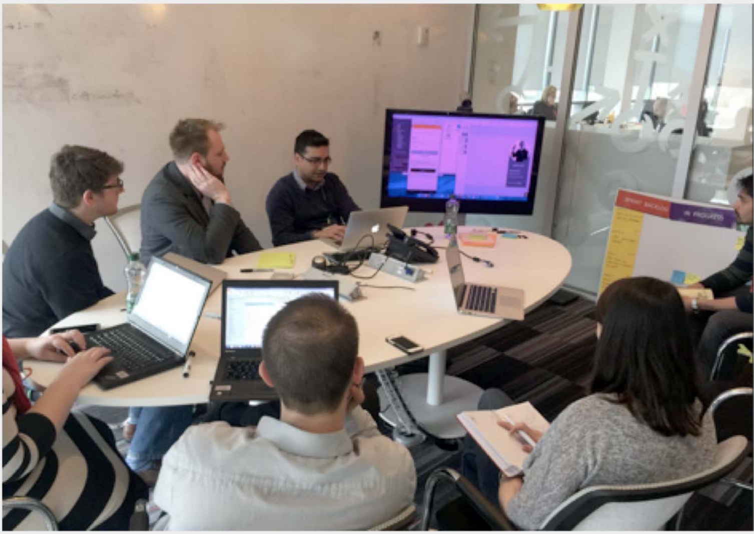


Figure 12.21: Sprint 2 Review

This incident highlights the positives of early and regular feedback on an evolving product. When would this have been caught if we were not attempting Scrum?

It also highlighted that such feedback triggers the need for some further discovery (for example, through Design Thinking).

This example frames nicely where the learning and feedback from a Delivery Loop can trigger the need to return to the Discovery Loop. We'll be exploring this and other patterns when we return to the Options Pivot in *Chapter 18*, *Sustain It*.

To share experiences, learning, contribute improvements to this practice, or simply read more about it, take a look at https://openpracticelibrary.com/practice/showcase/. We'll also return to other ways to capture metric-based feedback in the next chapter. The Showcase is one of the main forums for collecting this. Another one is the Sprint Retrospective Event.

The Sprint Retrospective Event

This is by far the most important practice in this book. If we don't do Retrospectives, we don't learn. If we don't learn, we don't improve. If we don't improve, what's the point of all of this?

In *Chapter 4*, *Open Culture*, we introduced Retrospectives as a feedback mechanism and, in particular, we introduced the Real-Time Retrospective practice as a great way to build an open culture into the team's foundation. Hopefully, the Real-Time Retrospective practice has worked well and continued to be used during the journey through the Discovery Loop, the Options Pivot, and the Delivery Loop. It offers an immediate place to add feedback in reaction to something happening in real time.

Retrospectives are short events where the team can take a break from what they are doing, take some time out to inspect how they have been working, and reflect on what has been working well, what hasn't, and what can be adapted and improved.

If you were involved in a long traditional Waterfall program, you may have done some kind of **Lessons Learned** exercise at the very end. This is where everyone sits down and captures all the things they might have done differently if they were to start over again now. It all gets written up and filed away in some shared drive. If we're lucky, some other team or project might benefit from this learning but, often, it never gets looked at again. Well, Retrospectives are just like this BUT they happen much more regularly and the team in question gets to benefit from the learning as they go along!

Retrospectives can happen at any time. In fact, you may well want to schedule a Retrospective after a long Event Storming session or after a first Product Backlog prioritization meeting. It is a foundational concept in the Open Practice Library as it is a tool that can be used at any time. A Sprint Retrospective is an event we run at the end of the Sprint or Delivery Loop. The Scrum framework states that it should happen immediately after the Sprint Review.

There are hundreds of different formats, styles, and themes of Retrospectives. People are coming up with new, fun, creative, and energetic ways to run Retrospectives all the time. Check out the Retrospective page of the Open Practice Library[19], which has links to many of these. If you have a link to your own personal favorite or have run your own, you could submit a pull request and add it!

We're going to share a few different ways we have run Retrospectives and differentiate between *Retrospectives of breadth* and *Retrospectives of depth*.

A Retrospective of breadth essentially inspects the time period that the event is focused on and asks what worked well, what did not, and what could be done to be better next time so as to address the things that did not work so well.

Our recommended approach to a Retrospective is to use a *diverge-converge* technique or liberating structure to ensure everyone has an opportunity to contribute their feedback. So, start by allowing a few minutes for everyone to silently write down their feedback on what worked well, what didn't, and corrective action that could be taken.

Then we converge and merge everyone's ideas together on an information radiator. We cluster common themes and have short discussions to achieve alignment. Finally, we agree on a set of corrective actions to come out of the Retrospective, prioritize them, and agree on owners to take them forward.

19 https://openpracticelibrary.com/practice/retrospectives/

The next Retrospective should always open by reviewing the previous Retrospective's actions. The following are a few alternative ways to ask those questions and provide headings on the Retrospective's information radiator:

- What should we START in the next sprint that we're not currently doing? What should we STOP in the next sprint that we are currently doing? What should we CONTINUE doing? What should we do MORE OF? What should we do LESS OF?
- What should we ADD? What should we DROP? What should we KEEP? What should we IMPROVE?
- What made us GLAD in the last sprint? What made us MAD? What made us SAD?
- What did we LIKE in the last sprint? What did we LEARN? What did we LACK? What did we LONG FOR? This format is known as the **4Ls**.

Preparing a simple canvas for one of these approaches will allow the team to collect and organize their feedback and radiate the resulting learning and actions for all to see.

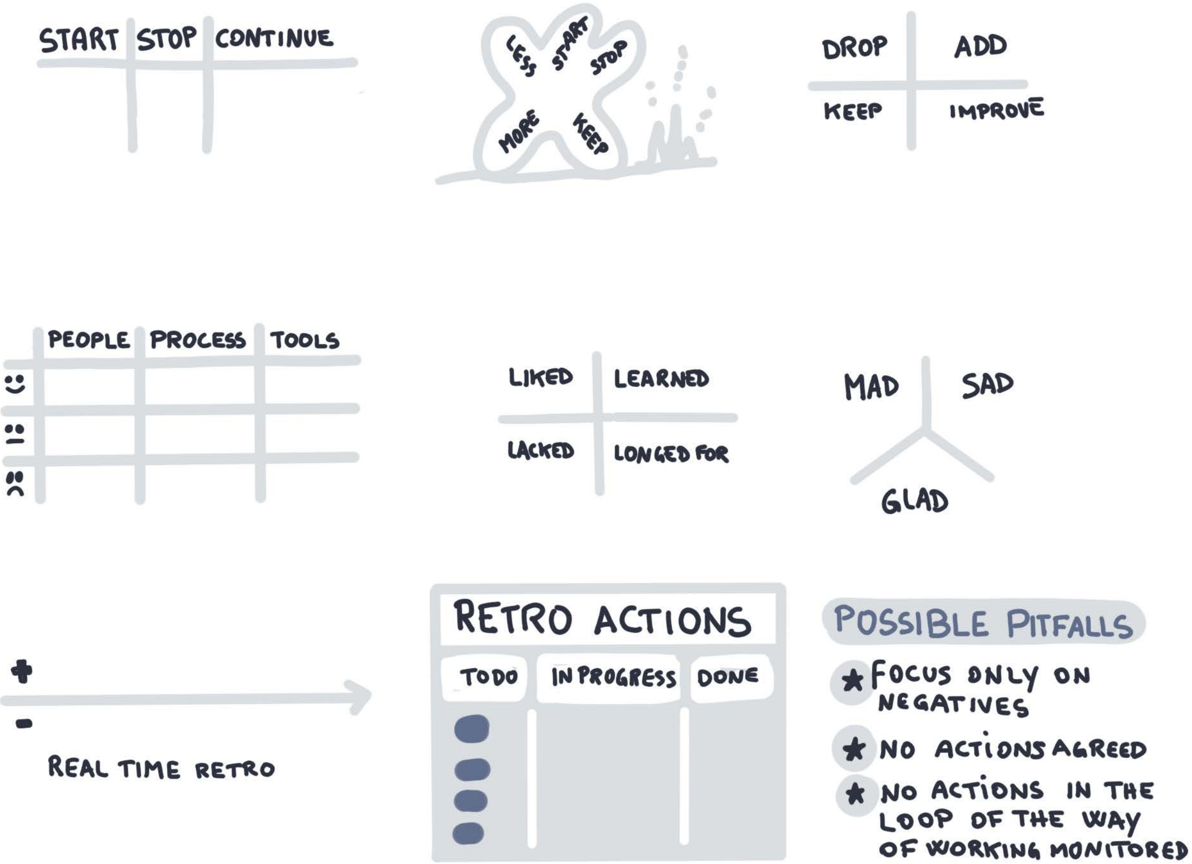


Figure 12.22: Popular Retrospective formats

A slightly more creative and fun approach that has become popular is to use analogies to reflect and inspect the previous sprint. Canvases can be prepared with a theme such as:

- **The Sailboat**: What is our island–the goal or target we are aiming for? What is the wind–the things that will take us there? What things represent the anchor dragging us back and slowing us? What are the rocks or the risks that we might be overlooking?
- **The Hot Air Balloon**: The hot air represents the things that are helping us rise up. What represents the weights that might be dragging us down? What is the sun that we want to steer towards for a smooth ride? What is the bad weather that might give us a bumpy ride?
- **The Motor Racing Car**: What is the engine moving us forward? What is the parachute slowing us down?

There are many other similar formats: the *Three Little Pigs* Retrospective with the houses made of straw, sticks, and bricks; the Star Wars Retrospective with the light side and the dark side. We've even followed musically themed Retrospectives such as Elvis and ABBA, where song titles represent the different inspection aspects.

Some of the common pitfalls of Retrospectives include focusing far too much on the negative and not enough on what has worked well. A good way to look at this is if something has been working really well, how might we **turn up** this behavior and do even more of it? Another common pitfall is failing to capture Retrospective items. It's very easy to whine for an hour about all the things that have annoyed us. But what tangible thing can we do to fix it or at least make it better?

On the other hand, sometimes we come up with too many things we want to fix and too many ideas for improvement. Use a prioritization practice (such as those outlined in the previous chapter) and radiate the actions on its own backlog, for everyone to see and help with.

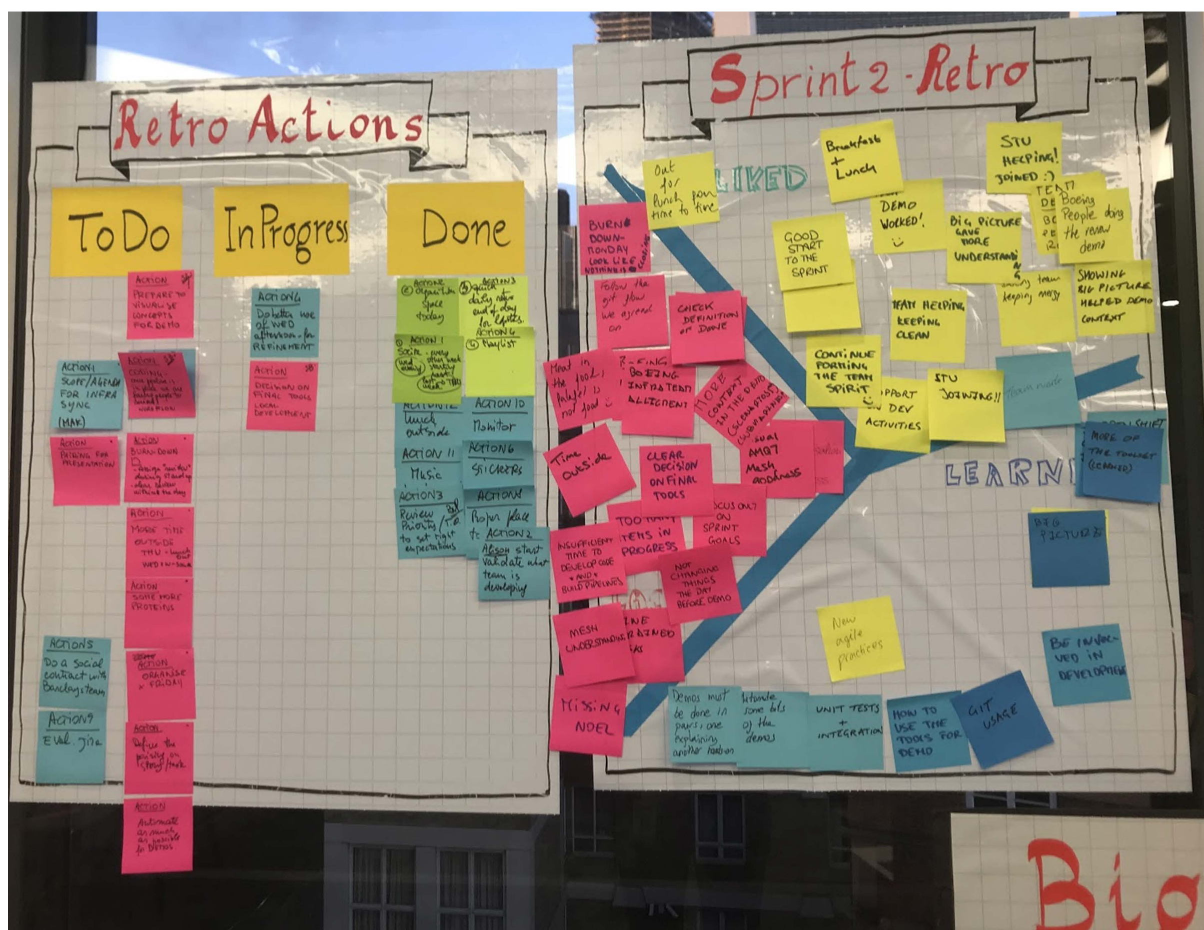


Figure 12.23: Example Retrospective canvas and actions list

Retrospectives should be fun and engaging. They should be something the team looks forward to and has everyone motivated to inject energy into. This is an opportunity for the team to continuously improve.

To help keep them engaged, especially for long-lived product teams, it really helps to mix up the format. Try new ideas out for Retrospectives. Rotate the facilitator around the team so they can share the ownership of this practice, figuring out what works well with Retrospectives and what we should do more of, and what doesn't. In other words, sometimes we should run Retrospectives on our Retrospectives!

A final tip comes from years of experience running Retrospectives away from the normal place of work and what we infamously call **The Pub Retro**.

The Pub Retro!

Back in 2007, I was experiencing a great Scrum team for the first time. I had dipped my toe into Agile in the previous couple of years, but it was really a mismatch of practices with a lot of Scrum-buts! One suggestion from a team member (who had come from a high-performing software house and was very skilled in Scrum) was to take our Retrospectives off-site and run them down at a local pub. He felt that removing people from their day-to-day workspace enabled them to look back and reflect on how the previous sprint had gone.

We also recognized that, in traditional projects, after many months of effort, the software would go live, and we would usually celebrate that with some kind of event. With Scrum, we should potentially ship our software at the end of every sprint and we should strive to go live with every sprint as we steer towards continuous delivery. Therefore, it's right to celebrate, and what better place to do that than down at the local pub?

A final point to make is that people can be a little more honest in a relaxed pub atmosphere, and more creative with their ideas. I must say, over the years, some of the very best ideas for improvement have come out of pub Retrospectives!

Figure 12.24: Pub Retrospective

Of course, it's important to bring all the artifacts and learning captured back to the physical workspace and that the Retrospective actions are included in backlogs, so they are prioritized and actioned.

Experiment with the location on Retrospectives. My teams have gone beyond pubs by doing things like ten-pin bowling, urban golf, and meeting at a nice café and having a *retro-breakfast* together!

There is so much to share about Retrospectives. Please take a look at https://openpracticelibrary.com/practice/retrospectives. There are several great links to resources on the page. If you have found a brilliant Retrospective format to contribute, please do add it by hitting the *Improve this Practice* button on the page.

Now that we have been through the 3-5-3 of Scrum and got some anecdotes and learnings from our use of it, let's see Scrum in action with the PetBattle team.

A Sprint in the Life of PetBattle: Getting Ready

The PetBattle team decided to work on a one-week sprint cadence. Aidan, Emma, and Jen had used Scrum previously, but always with two-week sprints; they saw this as an opportunity to tighten the feedback loop and deliver outcomes quicker.

Reminding themselves of the Scrum values and framework, the team drew up who was in the Scrum team on one of the unused whiteboards. There would be a single Scrum team (for now). Valerie would, of course, be the Product Owner. The cross-functional Development Team would be made up of four engineers (Ciaran, Aidan, Emma, and Jen), the UX Designer (Dave), and the Quality Assurance expert (Susan). Eoin would be the ScrumMaster.

Valerie was very proud of her Product Backlog. She felt, through using all the Discovery Loop and Options Pivot practices, it was well understood by her stakeholders and had been well prioritized using the WSJF economic prioritization model.

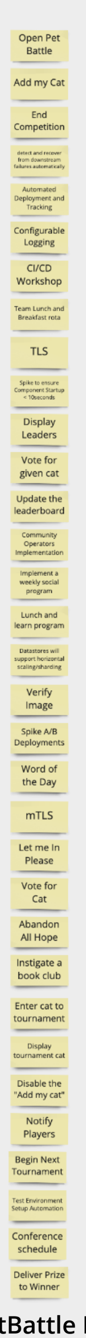


Figure 12.25: PetBattle Product Backlog

The team had their Definition of Ready up on the wall and had been running short Product Backlog Refinement sessions to get the top ten items of the Product Backlog ready according to the Definition of Ready. This meant talking through the items with Valerie (who occasionally would bring in a stakeholder or two to support the conversation), writing acceptance criteria.

The team used the Confidence Voting practice to assess whether they felt the item could be delivered within a few days of development. They would also talk through and update the emerging Logical Architecture diagram to ensure they had a collective understanding as to how the feature would be built. Dave had been doing lots of sketches and wireframes for the UI and would add these to the conversation at the appropriate point.

The Sprint cadence was set. Sprint Planning would take place between 10:00 and 12:00 every Thursday morning. Daily Scrum meetings would be scheduled between 09:30 and 09:45 every morning. The Sprint Review showcase would take place on Wednesday afternoons at 15:00, immediately followed by the Sprint Retrospective, which would be held off-site at 16:30.

A Sprint in the Life of PetBattle: Sprint 1 Planning

The first Sprint Planning meeting was excellently facilitated by Eoin, who had the agenda up on the wall and managed the time effectively. The team spent 15 minutes talking about the Sprint 1 Goal. Valerie wanted to see the PetBattle app up and running so that she could share a link to it with Mary and a few others who volunteered to be user testers. For her to do this, she reckoned a great place to start would be by being able to add her pet and starting a competition. If the app could do this in a week, she'd be happy. Jen was super keen to use all the green from go work they'd done when establishing the technical foundation and ensuring that all components were being built using CI/CD with automated deployment and tracking. The four engineers were really keen to take an Ops-first mindset approach and ensure, from the offset, that they had a good, automated recovery framework in place as well as configurable logging.

Bringing all these thoughts together, the team agreed on the Sprint 1 Goal to build the walking skeleton of PetBattle, connecting f/e to API to database underpinned by CI/CD, and provide the foundation for operations.

Next up on the agenda was to pull items from the Product Backlog, do a final check that they were ready, and place them in the Sprint Backlog. Eoin suggested the team copy the sticky notes and leave the Product Backlog information radiator complete. He would mark the items on the Product Backlog as they were added to the Sprint 1 Backlog. Eoin also had the idea to show on the Sprint Backlog what outcome the item was linked to. This was easy to see by referring to the Value Slice board. Overall, this simply meant that we had a great information radiator showing how outcome-focused the team was.

The team pulled the top ten items on the Product Backlog into the Sprint Backlog. They talked through and made some final edits to the acceptance criteria. Valerie was excited because this would deliver the entire top slice of value from her Value Slicing in just one week! Aidan and Ciaran started to get a bit nervous as the sprint backlog filled up. They were really pleased that the sprint balanced functional development versus non-functional and work focused on operations. The team also loved that they could do some educational workshops on CI/CD and put the long-awaited breakfast and lunch planning in place to help improve the culture. But there was some nervousness around the TLS security and ensuring the performance. This would need some technical spikes and research and, with just four working days until the first Showcase, Aidan and Ciaran were worried that they would not have the time to do everything.

They still proceeded to the second part of the Sprint Planning event and decomposed all the items into tasks. They decided to color-code their tasks, which would all be written on square sticky notes. Pink stickies would represent application development. Blue stickies would represent UI/UX and frontend development. Light green tasks would be focused on data configuration. Purple stickies would be automation. Dark green stickies would be infrastructure and platform work. Yellow stickies would be testing. There were some other tasks that were focused on culture, team building, and other squishy/fluffy stuff! They decided to call these squishy tasks, and these would be on the orange stickies.

The team divided into pairs and each pair focused on two features. Every 5 minutes, they rotated the story so every pair could think about tasks for every feature. All tasks were then added to the Sprint Board, duplicates were removed, and a discussion took place on each task, until the teams were

confident they had all the low-level details to deliver regarding each item on the Sprint Backlog as per the agreed acceptance criteria.

Finally, Eoin facilitated a confidence vote using the Fist of Five. After a count of three, everyone displayed with their fingers (one to five) how confident they were that they would achieve the Sprint Goal, by delivering the sprint backlog according to the acceptance criteria and tasks set out by next Wednesday at 15:00. The votes ranged from two to four with Ciaran, Aidan, Emma, and Jen all voting with just two fingers. The nervousness around getting all ten features done was growing. After some discussion, Valerie suggested removing the security- and performance-related items and putting them back on the Product Backlog. A second confidence vote now had everyone voting either four or five and there were many more smiles around the team!

The Sprint Board was ready, which included a Burndown chart to visualize the progress of the 26 tasks on the board.

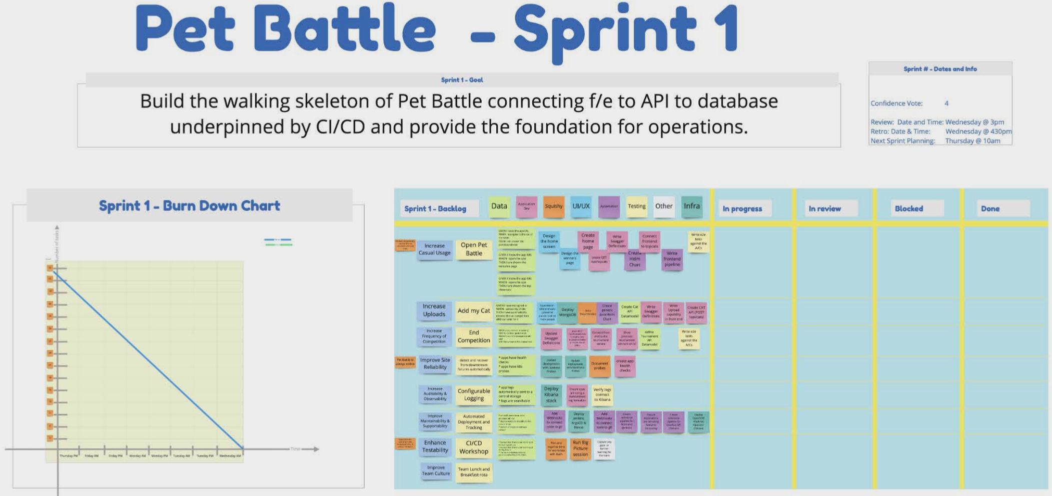


Figure 12.26: PetBattle Sprint 1 Board following Sprint Planning

Now we've seen Sprint 1 Planning complete, let's see how the team got on with its delivery.

A Sprint in the Life of PetBattle: Sprint 1 Delivery

Sprint 1 got underway. True to their social contract, the team worked as a mob when learning something for the first time and in pairs for all other tasks. They had their Daily Scrum every morning, not always at 09:30, and sometimes it ran over 15 minutes. This is something that would end up being discussed at the Sprint Retrospective.

Following constant nagging by Eoin, the team started to get better at updating the Sprint Board, with their avatars showing more clearly who was working on what. They also became more familiar with the Burndown chart, which showed how many tasks had been moved to Done at different points in time during the sprint. There was some confusion as to when the team could say a feature or task was done. Again, this would come up in the Sprint Retrospective.

Before that, at 15:00 on Wednesday, the team had been working frantically to get their final tasks done; they were nearly done, but a few remained in progress. The CI/CD had been working well and there was a version of the app promoted to their testing environment during the morning of their Sprint Review.

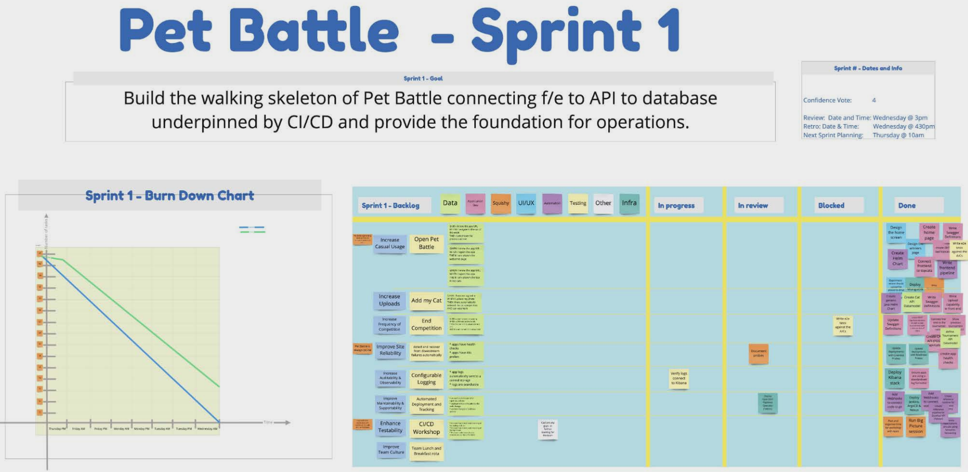


Figure 12.27: PetBattle Sprint 1 Board completed

A Sprint in the Life of PetBattle: Sprint 1 Review And Retrospective

A few stakeholders showed up to the Sprint Review, but not as many as Valerie had hoped for. They had to think about how best to share the updates. The Sprint Review had a few glitches here and there but, on the whole, did not go badly for a first sprint. Valerie explained the goal she'd set for the team and Eoin ran through the Sprint Board. The team, one by one, showed the features they had been working on. Some were shown using the app's new frontend. Some were more technically involved and the demonstration consisted of walking through config files and running some tests. They even showed the breakfast and lunch rota and how they had set up a working relationship with Deliveroo! Dave shared a UX update and his plans to run some more user tests in the next sprint. Susan showed the automated testing framework up and running and even ran the automated tests during the demo.

The showcase closed with Ciaran showing the updated big picture with the latest CI/CD infrastructure that this demo had been based on. Eoin brought out a new Information Radiator called **The Practice Corner**. This was a big Mobius Loop and Foundation that showed all the practices the team had used to date to build their Foundation, to get them round the Discovery Loop, through to the Options Pivot, and now completing the first Delivery Loop iteration. This really brought home how all the practices they had used fitted together. Stakeholders were invited to walk the walls to explore them in a bit more detail.

The Development Team adjourned with Eoin to the local pub where they brought a load of sticky notes. They ran their Sprint 1 Retrospective, which was run using the simple *Start-Stop-Continue* format. The team collectively came up with a great summary of the sprint and they agreed that they should:

- CONTINUE pairing and mobbing
- STOP being late for Daily Scrums
- START holding free weights during Daily Scrum updates to stop updates from going on so long
- START a social playlist to have some music in the afternoon
- START using the Definition of Done practice (including writing the Definition of Done in Sprint 2)
- CONTINUE using and improving the CI/CD solution
- START having more information radiators such as build monitors and test scores

The team raised their glasses and toasted SPRINT ONE DONE! The next morning, they'd start all over again with Sprint 2 planning.

These PetBattle stories provide an example of the full Scrum delivery cycle. As the team choose to do more delivery loops, the same processes are repeated.

Using Scrum with distributed people

As with earlier practices introduced in this book, the practices introduced in this chapter and on the Delivery Loop are highly collaborative and visual. A strong success factor is having Scrum teams co-located and being able to use physical artifacts like Product Backlogs and Sprint Backlogs.

But many Scrum teams have succeeded with using Scrum with the people distributed, especially when they have invested in strong digital collaboration and video conferencing tools. The importance of culture and psychological safety introduced in *Chapter 4*, *Open Culture*, is even more significant here and, crucially, the ability to learn and continuously improve. This is key to the success of Scrum teams, so investing the time to facilitate that environment of continuous learning can set up distributed Scrum teams for success.

There are many tools that will emulate every practice and the way of working used physically in the room. We look to replicate every practice we would use in a room to work online and in real time. So, if a team used sticky notes and whiteboards to collaborate during Sprint Planning, they need to be able to do that using a tool. If they use a big board for the Product Backlog, they need a tool to host that online. If they used Confidence Voting or planning poker for relative estimation, they need a tool to allow them to continue to do that. If they use a Burndown chart in the room, they need to find a way to keep doing that online.

To help you get started, we have provided a simple Sprint Board that can be used to build Sprint Backlogs during Sprint Planning, plan for feature demos during Sprint Reviews, and radiate the learnings during Sprint Retrospectives. This is available to download at https://github.com/PacktPublishing/DevOps-Culture-and-Practice-with-OpenShift.

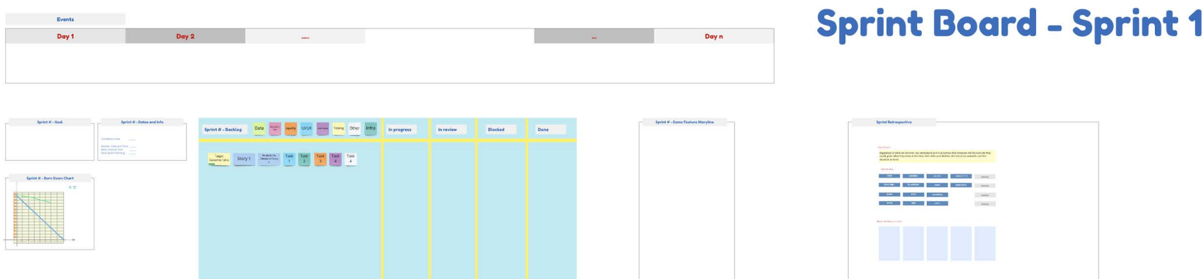


Figure 12.28: Digital Sprint Board for use by distributed teams

To complete this section about Scrum, we're going to drop a bit of a bombshell. Scrum is sub-optimal. Scrum does not comply with Continuous Delivery. Scrum can actually create bottlenecks in delivery rather than removing them!

You might now be wondering why we have spent so much time and pages explaining Scrum, sharing stories and learnings from using Scrum, and having our PetBattle team adopt Scrum. Well, Scrum provides a much-needed cadence and set of guardrails to start working in a more agile manner. Often, new or immature teams need these guardrails and frameworks to stay on the right path and ensure value is being delivered. But, as teams mature and get better, there may be a time when we can remove some or all of those guardrails.

When should we stop Scrumming?

Many teams love Scrum, as do stakeholders. The rhythm of being able to see outcomes delivered earlier more frequently is very satisfying and it corrects many of the problems experienced with the traditional Waterfall development of complex systems.

In *Chapter 11*, *The Options Pivot*, we talked about how the Product Owner should aspire to remove the need for themselves because the Development Team is so well connected to users and stakeholders, with strong psychological safety for two-way feedback and conversation. Earlier in this chapter, we talked about how the ScrumMaster should aspire to remove the need for themselves because they have facilitated an environment that is full of information radiation and very strong adoption of practices by a team that is fully autonomous, self-correcting, self-organizing, and continuously improving. The Product Owner and ScrumMaster have two very strong guardrails that facilitate the Scrum framework. They may never come down, but we should aspire.

Let's look at some other examples of guardrails that we might wish to remove.

Teams asking questions that suggest we've matured out of Scrum

Having worked with long-lived Scrum teams for several months, it's incredibly pleasing to see them continuously improve and become more autonomous with time. A pattern I've noticed with high-performing teams is that, after a while, they start to ask some very good questions about the processes and practices they are using. Some examples of questions I've had from ScrumMasters, Product Owners, and Development Team members include:

- Why are we waiting X days to demo/release? Scrum has the Sprint Review event scheduled on the last day of every sprint. But sometimes a team will complete a feature and be ready to demo it and release it on the first day of the sprint!

- Why don't we continuously run Retrospectives? The Real-Time Retrospective is an awesome practice at capturing retrospective feedback and actions in more of a real-time manner rather than waiting until the end of the sprint. If this is being used effectively by the team, maybe there is less of a need to wait until the end of the sprint to agree on improvement actions.
- Why are we doing these 15-minute daily standups when we already know what everyone is doing? This holds particularly true when the team is using mob programming more and more. The team members are so connected with each other and collaborate so closely that there is less of a need to formally synchronize every day.

A Product Owner I worked with at a European automotive company had a big A-*ha* moment in his fifth sprint when he asked, "*Why am I waiting till the end of the sprint to see feature* X?" He wanted to steer the team into a mindset whereby, "*When you've done a feature*, *show it to me and*, *if I'm happy with it*, *let's just ship it then and there.*"

The examples above show how Scrum might actually inhibit continuous delivery. They can, of course, release more frequently than just once in the sprint. The Scrum framework does not say that you can or should only release once.

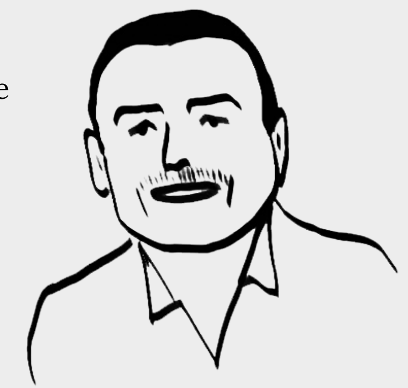
Teams may also feel, when they reach these points of realization and maturity, that they would be able to deliver more continuously if they moved to a different approach, to deliver and adopt Kanban.

Kanban

Kanban is Japanese for *signboard*. Kanban has its roots as a scheduling system used in *lean* manufacturing in the car industry and, in particular, Toyota cars.

In the context of Agile, Kanban is another methodology. It specifically strives to better coordinate and balance work amongst team members. Most famously, it employs a *Kanban* to help visualize the process. The board is split into categories of *work to be done*, *work in progress*, and *completed work*. Each task is written onto a card that moves from column to column as it progresses through the team's process. Tasks are prioritized. The board keeps everyone on the same page and is highly visible, so it allows corrections to be made easily.

Kanban Board!



When running our DevOps Culture and Practice Enablement workshop with members of our Open Innovation Labs team in 2018, we had the pleasure of meeting our colleagues in Japan. A lot of the lean culture and processes we describe in this book originate from Japan, so we always learn a lot from collaborating with our peers there.

During the course, we referred to the Kanban board many times, which seemed to generate some laughter! Our Japanese colleagues eventually told us not to say, **Kanban board** as we were really saying **signboard board**. That's why we now refer to the artifact as the Kanban.

Kanban requires strict limits on the number of tasks in progress at any given time. This is called the *Work In Progress*, or WIP limit. No new work can enter the column once the WIP limit is reached. Teams need to work collaboratively to fix issues, identify bottlenecks, and get tasks Done. This collaborative work style leads to continual improvement of the team's processes. The team can meet periodically to discuss changes that are needed, and these are displayed on the Kanban.

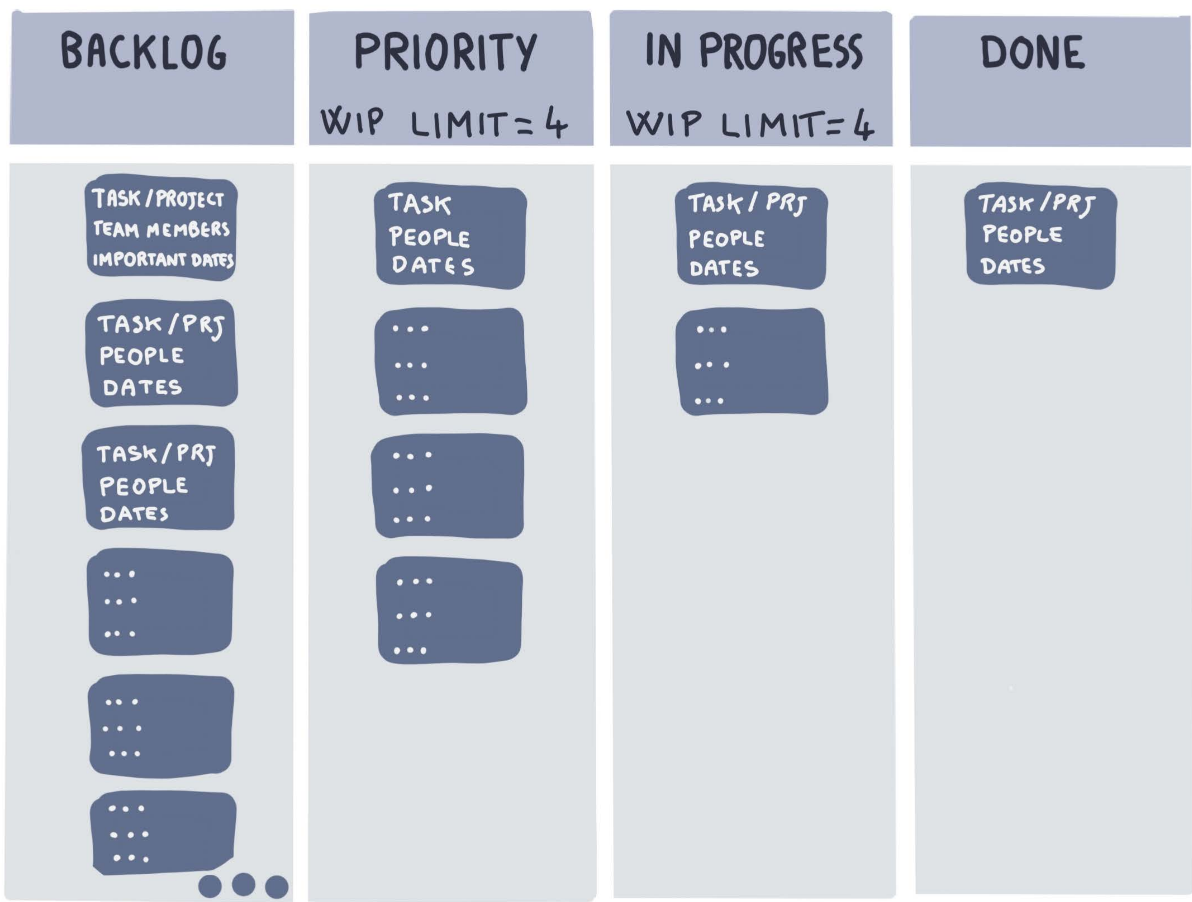


Figure 12.29: Example Kanban

Scrum and Kanban are not mutually exclusive. In fact, in our PetBattle example, we walked through the team using Scrum but, in the Sprint Board, they were using Kanban to visualize and track work.

PetBattle – Release Early, Release Often, Release Continuously

Implementing and delivering the PetBattle Tournament feature needed to be done quickly. The existing hobbyist application was developed in a very ad hoc manner, so there is little current process. Adopting Agile Scrum or Kanban looked like a great place to start for the cross-functional team.

Can the team decide when the *PetBattle Tournament Feature* gets deployed to production? YES! Can the team decide how to build, test, and deploy the *PetBattle Tournament Feature*? YES! By allowing the team to be autonomous, decision-making happens wherever the information is within the team. This trust and delegation of the PetBattle app delivery by the founders to the team is crucial to the product's future success.

Kanban has its own page in the Open Practice Library, which can be used for more information, discussion, and improvement, at https://openpracticelibrary.com/practice/kanban/.

An important practice that we have not yet introduced and that applies to teams using both Scrum and Kanban is the Definition of Done.

The Definition of Done

What does Done mean in the context of writing software features? Does it mean your code builds and runs on your computer? Does it mean testing has been performed? Does it mean the code has been checked in? What about documentation? What about operational readiness? These are all good questions, and the chances are that if you ask these questions to different people, you will get very different answers.

When we use a Kanban and/or a Sprint Board in Scrum, there is a column on the right with the title DONE. So, what does Done mean here? This is why we use a Definition of Done practice. The Definition of Done is a criterion agreed across the team and shared with others that should be met before any work items are considered complete by any team member. It is *collaboratively created, maintained, and enforced by the team* where non-functional and functional work that should be performed for each and every work item can be managed.

Earlier in this chapter, we talked about how we write Acceptance Criteria against features or stories on the Product Backlog. This is a specific criterion that applies to the context of the feature in question and only that feature. The Definition of Done criteria is additional criteria that also needs to be considered in the planning and delivery of every feature.

So, if we say that we are always going to check code into a source-code control system to trigger a CI build and a feature should not be considered done unless this has been done, we can add this to the Definition of Done. If a team agrees that we should always have a percentage of the code covered by automated tests and we should never fall below that given percentage, we can write Definition of Done criteria to manage this.

PetBattle – Definition of Done

In Sprint 1, the PetBattle team had a Definition of Ready in place and it worked effectively in ensuring that items they accepted into the Sprint were well understood and the team had confidence they would complete them in the sprint.

However, they did not write a Definition of Done. This led to some challenges during the sprint as there was ambiguity as to what should happen before a feature is considered Done. It came up in the Sprint Retrospective discussion and the team agreed to run a Definition of Done workshop during Sprint 2.

Eoin facilitated this and asked the question of what should always be done when working on a Sprint Backlog item, regardless of what it is. The team came back with several suggestions:

- The code and any other artifacts must always have been committed to a feature branch in Git.
- The feature branch should have been merged into the master.
- The CI build should always have run with all tests passing.
- We will have checked the documentation folder and made any updates to product documents to support the work being done.
- The Product Owner should have seen the feature working and the acceptance criteria satisfied.

This provided a great starting point and, in particular, Aidan and Emma recognized that the Definition of Done was a great tool for ensuring the non-functional aspects that they had brainstormed using the Non-Functional Wheel could be considered for each and every item.

Some of those items had ended up on the Value Slicing board because there was specific work involved in configuring them. But many of them needed team behaviors or some automation to be focused on continuously addressing the **non-functional requirements** (**NFRs**).

The team revisited all of the items on the Non-Functional Wheel and categorized each item into one of three lists:

1. Requires platform setup/config/spike
2. Comes for free with platform
3. Team behavior/principle

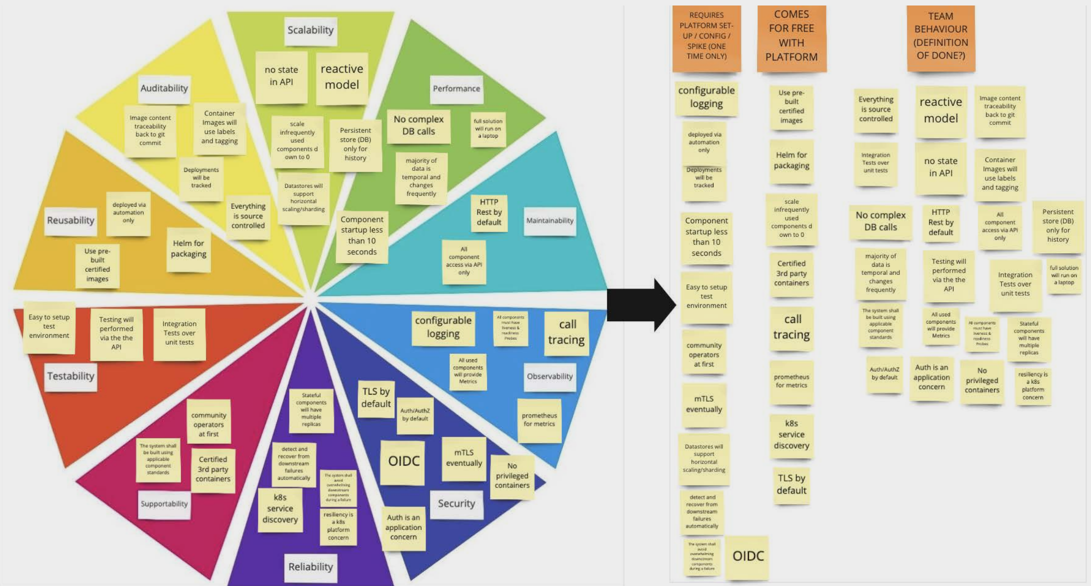


Figure 12.30: Transferring non-functional requirements to Definition of Done criteria

Items in the first column were seen as features to be included on the Product Backlog, as they needed specific work to be done to enable them in a sprint. Items in the third column were all items that should be considered as a part of the Definition of Done. However, given they were so low-level and detailed, the team decided to add a few more items to their Definition of Done that would mean they were duly considered:

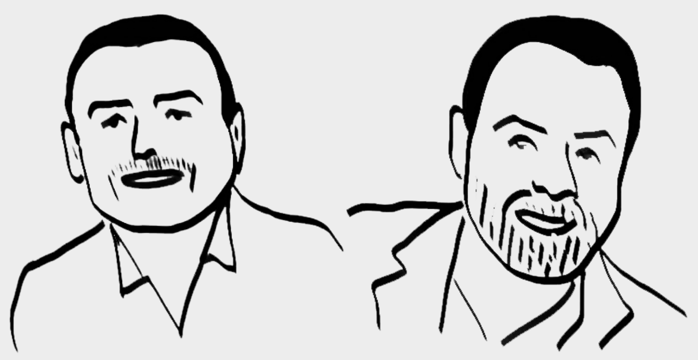
- There must be at least one automated integration test covering acceptance criteria.
- Checks will be run (automated where possible) to ensure that all non-functional principles remain upheld.
- The feature is running on the demonstration laptop and ready for the stakeholders to see demonstrated at any time.

The Definition of Done provides a place for non-functional requirements and quality to be managed effectively. While it starts as a big visual information radiator, it drives more and more quality into the product. It also provides many opportunities for automation. Given the Definition of Done criteria is something that will be tested for each and every backlog item accepted into a sprint, the same tests to meet the criteria should be run over and over again. So, automation is certainly our friend here. To explore the Definition of Done practice further, visit the Open Practice Library page at https://openpracticelibrary.com/practice/definition-of-done/.

Now that we have introduced the Definition of Done, when we plan sprints or the tasks to deliver a specific feature, we can improve the questions that teams should ask about each backlog item. What tasks are needed to deliver this feature to satisfy the agreed acceptance criteria and comply with the Definition of Done? And, as Retrospectives continue to inspect how to improve, the team should consider what they can add to their Definition of Done to improve the overall product and what to automate to make Done a part of CI/CD.

When a team does not have a Definition of Done visible or they are failing to follow it or build it into their CI/CD pipeline, it is a Bad Agile Smell that tends to result in the rise of technical debt and the slowdown of team velocity. Let's look at a few other Bad Agile Smells we've seen.

Bad Agile Smells



Over the years, we have spotted several anti-patterns in Agile teams and things we look out for and test early in our engagements with customers. Here are our top ten:

1. There is no Product Owner or there is a Product Owner who lacks the three important attributes—empowered, available to the team, and understands the business.
2. No working software: Either CI/CD is not working sufficiently (or not in place at all) or automated testing is not sufficient. There should always be a working version of the product that can be demonstrated or released at any time. It may not do very much, and the business may want more features before the release, but this has to always be a business decision rather than a technical constraint.
3. Sprints are pre-planned in advance. With variable scope, we need to let the teams control what they load into their sprints. Over time, they will establish metrics on their velocity and predictability—both should get better and there are powerful tools to help them size and forecast accurately.
4. Retrospectives don't happen or don't result in actions. Continuous improvement should be a part of the DNA of everyone involved in this system of work. The Retrospective is the fundamental building block to enable this.
5. Stand-ups take over 30 minutes and lots of meetings. For Agile delivery to be effective, we have to allow the teams to get on with the work and figure out the solutions themselves. Overloading them with meetings will only slow them down. Let's keep the meetings to the basic events of Scrum and Product Backlog Refinement.

6. Water-Scrum-Fall: Upfront planning and design, delivery of features in iterations, but only to pre-production because a tail-end operational readiness phase is needed. This removes all the benefits of Agile and does not provide continuous delivery. Think about using the Mobius Loop to articulate how quickly and frequently we're really traveling round these loops.
7. Teams are not cross-functional. This results in dependencies, blockers, and an inability to deliver continuously.
8. Lack of empowerment in teams to say no to items that are not ready. Using tools like the Definition of Ready and empowering teams to be autonomous drives up the culture, motivation, and ability to deliver continuously.
9. Imbalance of functional versus non-functional features work pulled into sprints. We need to continuously balance the needs and feedback from the team (for example, Retrospectives), including the need to continuously improve architecture and CI/CD and keep the team filled with autonomy, mastery, and purpose. Otherwise, technical debt rises, motivation declines, and continuous delivery slows or stops.
10. Push not pull. The systems we put in place are to empower teams to pull work rather than have it planned for them and pushed upon them.

These Bad Agile Smells should be kept in mind and, if they start to appear, we recommend using the Retrospective event as a starting point to discuss and course-correct.

Conclusion

In this chapter, we progressed from the Discovery Loop and Options Pivot and focused on how we deliver features into products using practices on the Delivery Loop. We explored different domains of delivery using the Cynefin framework and saw how Waterfall remains effective for work in the *clear* domain, whereas Agile is more suitable for work in the *complex* and *complicated* domains.

We explored where Agile came from, the Agile Manifesto, and took a detailed look into Scrum and Kanban methods, the practices they use, as well as supporting practices such as the Definition of Ready and the Definition of Done.

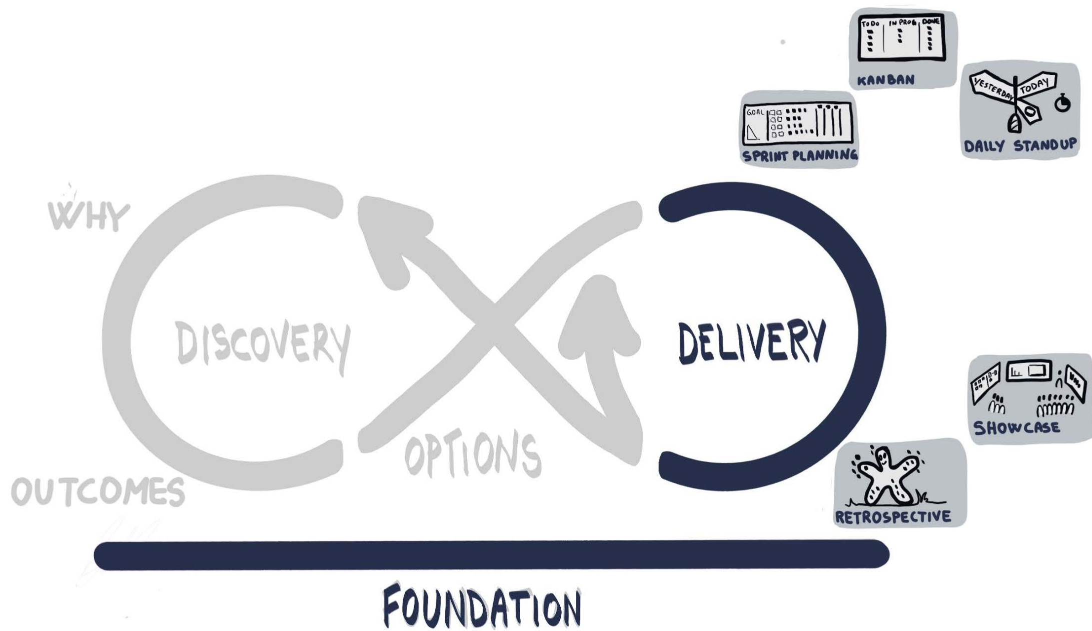


Figure 12.31: Adding Delivery Loop practices and more practices to the Foundation to support delivery

We can now see how Agile frameworks and practices help achieve continuous delivery when using platforms such as OpenShift and, when coupled with high-performing teams and an autonomous culture, we can deliver outcomes that matter earlier and much more frequently.

In the next chapter, we're going to dive deeper into the Measure and Learn part of the Delivery Loop and look at different mechanisms and tools that facilitate measurement and learning.

13

Measure and Learn

Startup success can be engineered by following the process, which means it can be learned, which means it can be taught. – Eric Ries

In his book, *The Lean Startup*, Eric Ries describes a startup company as a *human institution designed to create a new product or service under conditions of extreme uncertainty*. He outlines a process to help deal with this uncertainty where a tight feedback loop is created around the creation of a **minimum viable product** (**MVP**). He argues that being able to react, fail fast, and use a data-driven approach to measurement assists in decision-making that is based on reason rather than emotion. This ability to learn from small experiments can be seen as a form of business agility – the ability to pivot quickly in the face of ever-changing circumstances. In lean terms, this feedback loop can be summarized as **Build, Measure, Learn**.

The cultural and human aspects of this process cannot be overlooked. Uncertainty and humanity are common bedfellows. Ron Westrum, an American sociologist, posits that organizations with better "information flow" function more effectively. He argues that a good culture requires trust and cooperation between people across the organization, and therefore it reflects the level of collaboration and trust inside the organization.

Second, better organizational culture can indicate higher-quality decision-making. In a team with this type of culture, not only is better information available for making decisions, but those decisions are more easily reversed if they turn out to be wrong because the team is more likely to be open and transparent rather than closed and rigid.

So, how can we take these ideas and make them actionable in our delivery processes? When timelines for delivery are tight and deadlines are fast approaching, a team's ability to deliver and operate software systems is critical to the business performance.

With software, there are often two competing forces at work. Innovation, which inherently is accompanied by system change, and running software, which is serving end customers and implies that the system is stable. We can identify two important areas to focus on here:

- To help measure the effectiveness of a team's development and delivery practices
- To start measuring and monitoring activities that allow the rapid diagnosis of issues

In this chapter, we are going to explore different mechanisms and techniques to take measurements out of our Delivery Loops and use those measurements to drive decisions and next steps. We call this approach **metrics-driven transformation**. We will also learn what to measure and how we can take these measurements and make them visible, ultimately to help answer the question of whether or not we have moved the needle at all during our transformation journey.

Metrics-Driven Transformation

Metrics-driven transformation focuses on using value-based business metrics to understand how technology-related investments impact organizational performance and provide specific tools and guidance to help improve those metrics.

In the previous chapter, we looked at different approaches to doing delivery, whether that be Waterfall, or using an Agile framework such as Scrum or Kanban. As we complete loops of delivery, we want to take value-based metrics to validate hypotheses, confirm the results of experiments, clarify the impact of our feature deliveries, determine whether we have moved toward the Target Outcomes we set out, and make decisions around what to do next.

There are many different levels of measurements we can take in our delivery ecosystem and a growing number of sources we can collect them from. In this chapter, we will explore metrics we can collect automatically from our software and our platform as well as practices we can use to collect metrics from our users, customers, employees, and the wider organization. Let's start by re-visiting some of the practices we've already used and see how we can use them to collect measurements and learning.

Where to Measure and Learn

In *Chapter 10*, *Setting Outcomes*, we introduced the practice of setting Target Outcomes based on all of the learning that came out of the practices on the Discovery Loop. We showed how to make these measurable and how to visualize them as an information radiator so everyone can inspect them. Using metrics, we can inspect where we are now in quantifiable terms, where we've been, and where we want to get to for each measurable outcome.

We explained the difference between primary (business-focused) outcomes and supporting enabling outcomes, which are non-functional-based outcomes. Since then, we have organized all of our work around those Target Outcomes and have radiated them in other practice artifacts, including the Value Slice on the Options Pivot and Scrum boards in the Delivery Loop. Those outcomes should be our starting point for measurement, and re-visualizing them will allow us to measure the progress we have made (or not) toward the target value:

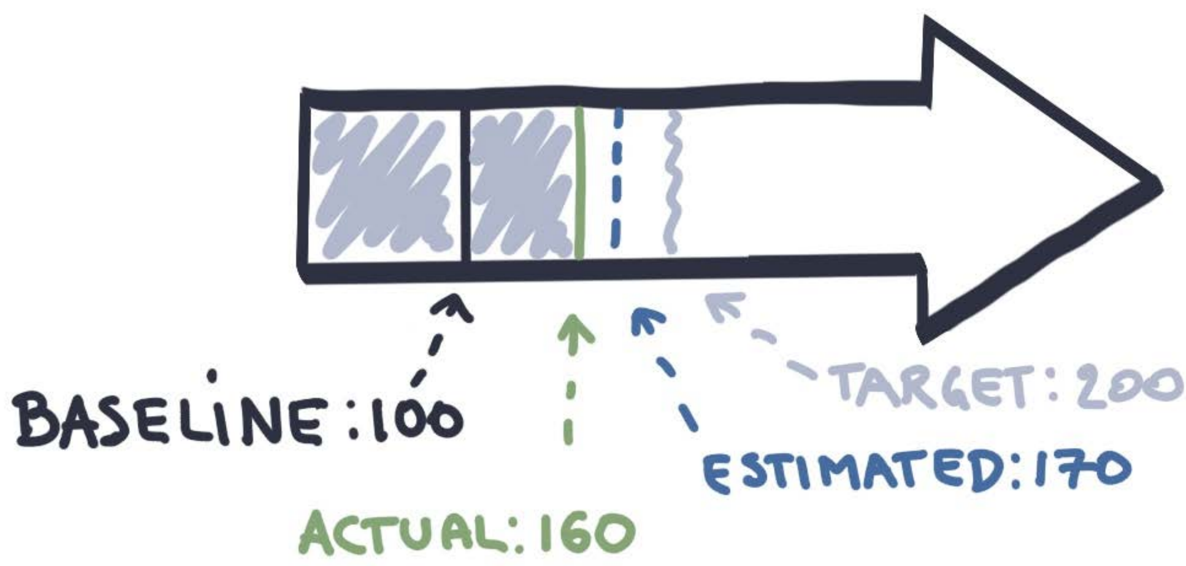


Figure 13.1: Measuring targets

When measuring the outcomes, we need to be aware of any prior measurement that may form an historic baseline, and potentially what any estimate of value may be against the actual value. Thinking about these different forms of measurement leads us to ask the question, *Where and when should we take measures and learn, and where should we inspect them?*

The Showcase

Ideally, we take measurements as soon as a delivery item is complete. That might be at the end of a sprint in Scrum or after delivering a completed feature in Kanban. Perhaps we can build in metrics data collection to the application code and run a report just before or during a Showcase. For example, if we have a Target Outcome around increasing the user base to 10,000 people, every Showcase could provide an update on what the current user base is and whether it is moving in the right direction toward the target.

There may be a lag in the availability of data that means it cannot be presented at Showcase events. In this case, there are two plans of attack. First, we can use the Showcase itself to collect some measurements and learning from stakeholders. Alternatively, we could collect measurements and learning from stakeholders and users on the latest Product Increment. But how?

Perhaps we have an outcome around stakeholder confidence and/or user happiness in the application we are incrementally building. If so, then what better opportunity than to ask the stakeholders after seeing the Showcase of the latest increment how confident or happy they are with what they have seen? This can be quantitative, in that we can ask stakeholders to rate the product on a score of 1-10. It can also be qualitative in the conversation that happens around this scoring and supplementary feedback from collecting feedback data.

Perhaps we have an outcome around employee engagement and team skills. Again, what better opportunity than at the end of a Delivery Loop iteration to survey the team on their happiness level and ask them to rate themselves on different skills? This visualization will not only allow us to see trends, but also identify the positive and negative effects as a result of the team education and cultural activities that we are undertaking.

Showcase events allow the opportunity to show many different kinds of metrics: Software Delivery metrics, Platform metrics, and Team Velocity metrics. We will explore the details of these shortly. If these are things that will be of interest to the audience and help them understand the impact of everything the team is doing, by all means, include them. However, the event where a more in-depth conversation can happen is the event that typically follows the Showcase, the Retrospective.

The Retrospective

We introduced the Retrospective practice in the previous chapter and looked at many different formats of running them. Let's consider metrics a little further and also an engineer's perspective of Retros.

The Retrospective – an Engineering Perspective

Back in the late 1980s, I vividly remember being introduced to **feedback loops** and **control theory** in chemical engineering classes, way before software engineering totally enveloped my career! If you wanted to control the level in a tank, the flow rate in a pipe, or nearly any dynamically changing system, you needed to learn how to keep that system stable. The software engineering practice of Retrospectives always makes me think of those second-order feedback loops, the physical connection back into our software design and development process that allows us to learn and adapt so that the system can become more stable.

Retrospectives all have a common goal of allowing the team to inspect what has just happened, while allowing them to adapt and refine what goes on in the future. This is a critical function. If there is no feedback or feedback is not actioned, the team may start to lose faith in the whole delivery process. In the engineering world, if feedback fails, the tank overflows!

It should come as no surprise that metrics should be included as hot topics in any Retro. Teams that discuss their poorly performing SDO or CI/CD metrics can gain a lot of insight into what is going wrong with the software delivery process. So, celebrate when the build starts to take 20 minutes to complete. This means that the metrics measurement and trending are in place and the team can now take action to find out why it has gotten so slow and seek to improve it.

It's possible to save a lot of wasted time and resources by listening to timely feedback. If feedback is ignored, you can incur an opportunity cost by spending the next year carrying out remediation projects instead of rolling out new, revenue-generating features. The moral of the story is to Retro, Retro, Retro! You can really never have enough feedback.

With all the tools and technology put in place when we created the technical foundation (in *Chapter 6*, *Open Technical Practices – Beginnings, Starting Right*, and *Chapter 7*, *Open Technical Practices – The Midpoint*), there are huge amounts of data, metrics, and analysis we can collect and conduct. If you run pub Retros (as introduced in the previous chapter), there is nothing better than taking a few print-outs of reports from these tools and taking them down the pub to analyze together over a pint of Guinness!

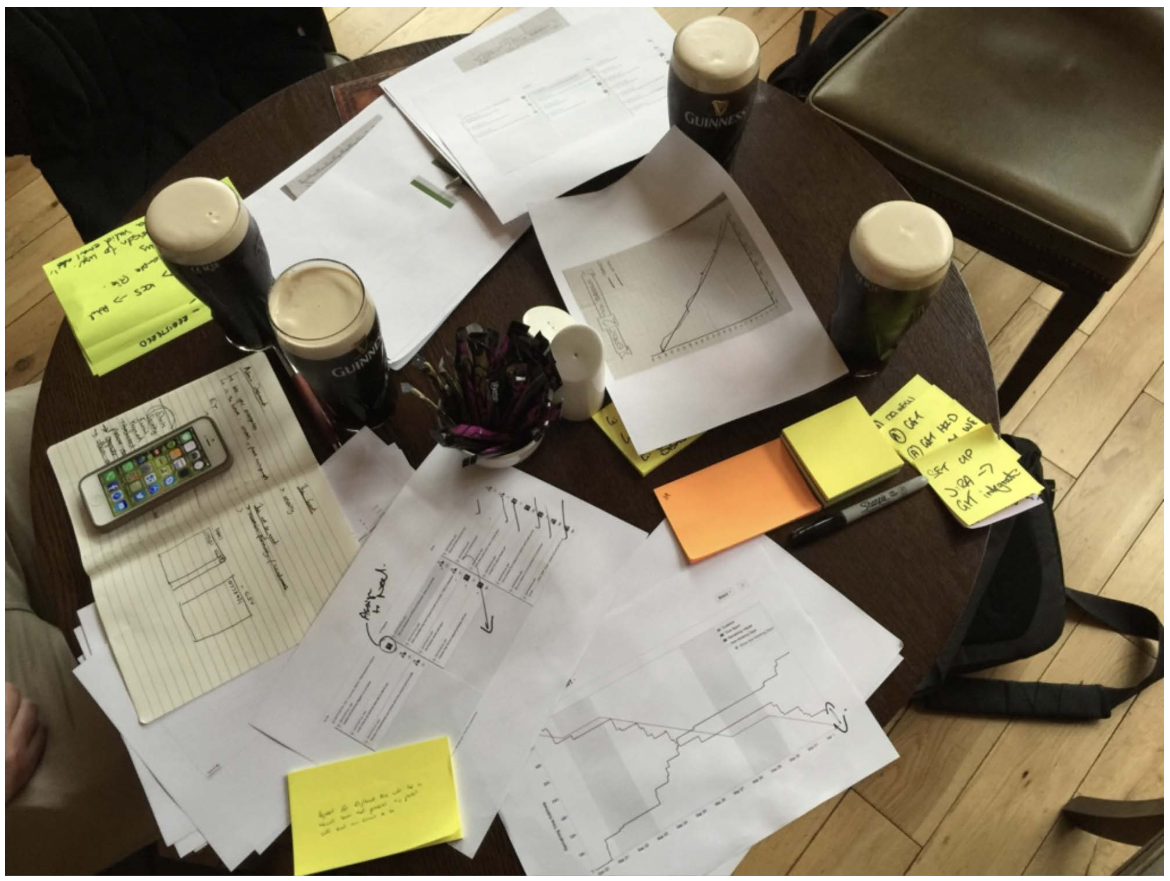


Figure 13.2: Pub Retrospective – discussing metrics

Export your build statistics, latest test results, static code analysis reports, Burndown charts, and whatever else you can find, lay them all out on the table, and ask yourselves: what does this data tell us about us as a team that we don't know already? What can we learn about ourselves? What patterns exist? What can we do in the next sprint to make these measurements better?

Retrospective actions that are often taken include an increased focus on CI/CD infrastructure, increasing thresholds around code coverage for testing, and adding more or faster feedback loops for the team to learn from.

Inspecting the Build Stats at Retrospectives

This is one of my favorite examples for highlighting the power and impact of the Retrospective. I collected this chart back in 2008 when delivering an access control management solution for a UK telecoms company. My team was building a Java-based application using Scrum and continuous delivery. They used Hudson for continuous integration and had a huge number of automated tests that were incrementally built into the app as part of the team's Definition of Done.

Several build graphs were exported every 2 weeks and taken to the team's Retrospective. The following chart shows the duration of the build as well as the success of the build (red means the build failed, yellow means the build succeeded but some automated tests failed, and blue means a successful build with all automated tests passing):

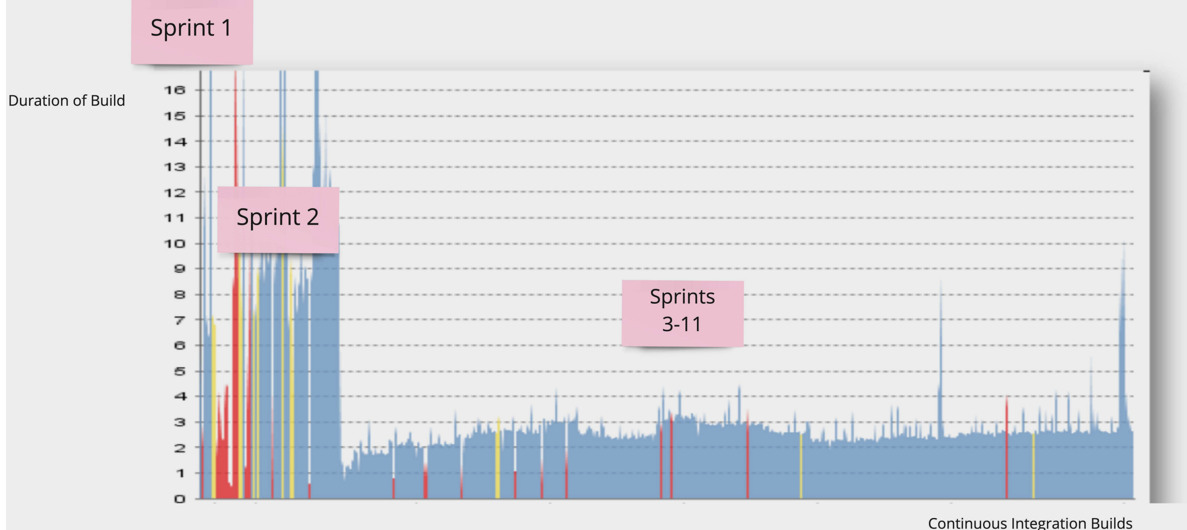


Figure 13.3: Inspecting the build time statistics at Retrospectives (the graph has been annotated with a few labels to reflect the sprint that the data was a part of)

As we can see, the build was very unstable in Sprint 1. A much earlier version of this chart was taken to the pub Retro at the end of Sprint 1. The team inspected it and agreed that, during Sprint 2, they would invest some time investigating the build stability issues.

Two weeks later, Sprint 2 was completed and the ScrumMaster brought another print-out of this chart from Hudson. The good news was that the Sprint 1 Retro action had paid off as the build was much more stable. However, it was noticed that the build was sometimes taking more than 15 minutes to complete. This was a much longer feedback loop than needed, so a further retrospective action was taken in Sprint 3 to address this. We can see that from Sprint 3 onward, the build was mostly stable and relatively quick.

Imagine if we had not used the Retrospective to look at this data. Imagine if we had let the slow build just fester over time. Imagine how much time would have been lost. This is why we do metrics-driven Retrospectives.

As we come out of the sprint, we now have the opportunity to learn the results of experiments we've designed.

Experiments – the Results!

In *Chapter 11*, *The Options Pivot*, when we introduced the Options Pivot, we introduced some of the advanced deployment strategies we could design in our experiments. If we decided to use one or more of these strategies when designing our experiments, now is the time to measure and learn from what actually happened.

If we designed an A/B test, we look at the data collected about the traffic, interaction, time spent, and other relevant metrics so we can judge the effectiveness of the two different versions based on the change in users' behavior.

If we designed a Canary Release, have we learned from the behavior of the users that were a part of the Canary Release candidate to validate whether the feature should be rolled out to the wider population? Similarly, with dark launches and feature toggles, we collect metrics from usage to assess whether the feature release should be extended to a larger user group, or should be switched off and dark launches rolled back.

All of the learning captured from these experiments and the resulting analytics is quantitative. This means that you can study the data, observe trends, decide to extend the experiment, or create whole new experiments. As you run more experiments and gather more data, your decision-making capability becomes stronger and is based on metrics gathered from it.

Of course, you can't validate everything through the use of numbers. Some commentary with further learning and understanding is needed by talking to end users, known as quantitative feedback. Let's explore a couple of practices to help us do this, starting with user testing.

User Testing

User-based testing is a technique that focuses on user interactions with a product. These types of evaluation directly involve end users and focus on the person. Let's dive a little more into this by looking at two user testing practices: usability and guerilla testing.

Usability Testing

In a usability testing session, the team observes real users interacting with the product. Typically, a facilitator sits with a user and asks them to complete tasks and explain their thinking as they go. The team sits in a separate room and observes the testing by video link.

A usability test is not a focus group; it's focused on what the user thinks and does in the real world. An Empathy Map, as introduced in *Chapter 8*, *Discovering the Why and Who*, can be a very useful supporting practice.

A usability test can be run on an existing product, a prototype, or even a competitor's product. The prototype could be working code, or it could be something as simple as a few clickable images. Test early and often to create products that delight users and solve real needs.

Usability testing often highlights something that's obvious to someone who has been working on a product, but might be confusing to a user. What we think users need might not be what they actually need. Indeed, what users think they need may not be what they actually need! Usability testing can help answer questions such as, *Are we on the right track*? *What problems do we still need to solve*?, or *Which features should we build next*? With early feedback from real users, teams can avoid sinking time into a feature that's confusing or not useful.

"We Are Not Our Users"

One of the best designers I've ever worked with once told me that *We are not our users*. Since I spent a fair amount of time in front of a code editor among my development team, I thought I got what he meant. I didn't fully appreciate what he meant until the following incident occurred sometime later.

We were building a lightweight prototype for a mortgage application. The frontend was using an existing bank API for calculating mortgages and the application we were building on top asked a few initial questions before prompting the person to enter an estimate of their income. The text box initially filled in by the user had a placeholder saying, *e.g.* 100,000.00.

While doing some development and testing with the product owner and user experience designer, we must have filled out the form hundreds of times! Every time we used the form, we filled it out the exact same way, just by putting 100000 into that box. Not once had any of us ever put a decimal place in the box!

Fast-forward a few weeks and we were doing some real-world testing. Of course, the very first person we have to use the application goes through the form, putting "120000.00" into the box and hitting submit. As you can probably imagine, we expected everything to work swimmingly and the user feedback session to continue; but instead, the application crashed.

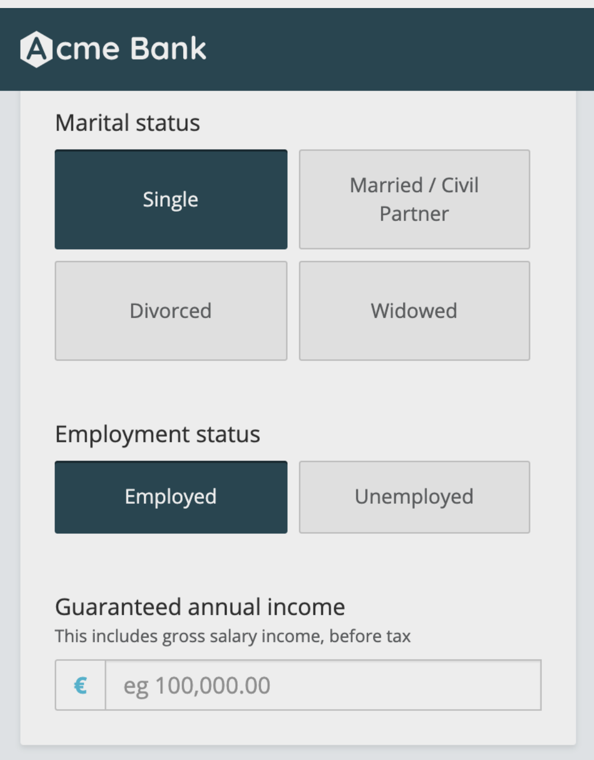


Figure 13.4: Mortgage application

The business analyst who was running the usability test immediately phoned in to say that the app was broken. We replayed the exact thing the user did, only to discover that the bank's API was unable to accept a decimal point. But for us, the real surprise was that no one on our team had ever noticed this issue before. As a small team, we were quick to fix the issue by updating the placeholder text and sending only an integer to the API.

This example always reminds me that we are not our users. Users do weird and wonderful things in an application. Even if you try to test all the scenarios, there is almost always going to be some edge case you have not thought of, and the chances are the first person to use your app will do the one weird thing!

Read more about the usability testing practice, share your own experiences, ask questions, and improve the practice further in the Open Practice Library at openpracticelibrary.com/practice/usability-testing.

Some of the challenges you may encounter when trying to organize usability testing include getting actual customers to access legacy systems or not having the time, money, or experts to conduct this level of testing. Guerrilla testing offers a low-cost alternative or supplementary form of qualitative user testing.

Guerrilla Testing

Guerrilla testing is a low-cost, lean, and Agile method of collecting data for testing and validating a hypothesis in a short session focused on specific tasks. Participants are not recruited in advance, but instead are approached in a number of environments by the team, where similar demographics are targeted; for example, customers in coffee shops, or administrators in an office environment.

This testing provides a simple method for collecting enough data to make well-informed strategic design decisions. It can also assist senior stakeholders and product teams in understanding the importance of usability testing and customer feedback. Everyone on the team can facilitate without any research experts. It is a flexible approach that can be implemented at any stage of product development.

Guerrilla testing with a box of donuts in a busy Dublin bank!

Doing any usability testing or generating feedback from a control group doesn't have to be a large costly exercise involving huge numbers of focus groups; it can be a simple activity! In the previous example, we undertook guerrilla testing after every sprint. It was an amazing way to generate real-world feedback and cost us next to nothing.

We would go into a branch of a bank, choosing a different one each week in different parts of the city to get a range of viewpoints. All we brought with us was the application and a massive box of donuts! The plan was simple. All we would do is lure people over to our stand and ask them to take part in a simple feedback session in exchange for free donuts! Users would road test the application and give us some valuable insight into things we had overlooked or failed to account for. The cost of the box of donuts was cheap but the feedback it gave us was invaluable.

Read more about the guerilla testing practice, share your own experiences, ask questions, and improve the practice further in the Open Practice Library at openpracticelibrary.com/practice/guerilla-testing/.

PetBattle Usability Testing

By looking at both the smooth delivery of changes and the overall stability of the production system, the PetBattle team hopes to avoid the types of failures seen when the hobbyist version of the application went viral and, at the same time, rapidly introduce new fixes and features to the software product.

The first area the team would like to gather data from and measure is how long it takes to get new features into their testing environment. By automating and reducing the amount of time it takes to get fixes and features into an environment where testing can occur, this will shorten the feedback loop so the team can get fast feedback on their changes rather than having to wait for a release to happen. The PetBattle team is not sure exactly what the right metrics to measure here are, but thought that starting with this one would help them discover more software delivery metrics that could be useful.

The second area of measurement the team would like to focus on is some simple performance testing. In the test environment, the plan is to concurrently load up the PetBattle application to see how it behaves and to try to identify any bugs or bottlenecks in the system. Some pre-emptive checking here will hopefully reveal any issues the team saw in the hobbyist version of the app. Again, the team is not quite sure exactly what parts of the system to focus on with their performance testing, or what metrics they should target yet, but deploying the PetBattle suite into an environment where this testing activity can happen is the right first step.

Taking a Measure and Learn approach to our Showcases, Retrospectives, experiments and user testing moves us into a continuous cycle of building something small, finding a way to measure from it, and gathering the data and learning from it, which then generates further ideas.

The *Build*, *Measure*, *Learn* feedback loop made famous in *The Lean Startup* is one of the most powerful tools and mindsets that is enabled by the platform, technology, and cultural tools we've been equipped with throughout this book:

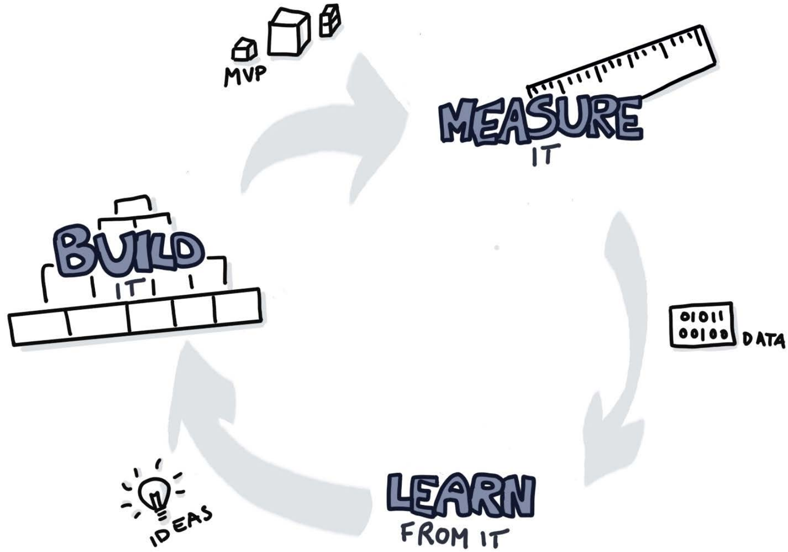


Figure 13.5: Lean – Build, Measure, Learn

Let's go into a little more detail about what to measure.

What to Measure?

What you measure is what you get. – H. Thomas Johnson

In assessing what to measure as part of the *Build, Measure, Learn*, feedback loop, we are going to be standing on the shoulders of giants. There is a whole literature section on DevOps metrics available to us and we are going to call out our current favorites here. Top of that list is the DevOps DORA report[1] and the book *Accelerate*,[2] which are works where scientists described how they have taken a data-driven approach to measuring DevOps culture and practice. In the **DORA** report, effective IT delivery organizations take about an hour to get code from committed into trunk (in Git) to "running in production". This sounds great! So let's look at some of the detailed metrics that allow us to hone in on such a goal.

Measuring Service Delivery and Operational Performance (SDO)

A key question for any team is, *What does good look like?*

From the research in Accelerate, it appears that leading organizations update their software many times a day instead of once every few months, increasing their ability to use software to explore the market, respond to events, and release features faster than their competition. This huge increase in responsiveness does not come at a cost in stability or quality though, since failures are found and fixed quickly.

Measuring software delivery performance is a difficult task. It would be easy to think that output-based measurements would suffice. Some often state that measurements might look similar to the following:

- Lines of code written and committed by developers per day
- Team utilization as a measure of team productivity
- Team velocity (or the number of stories/features delivered per sprint)

Unfortunately, if we dig into these a little, we can quickly find problems with all of these measurements.

1 https://www.devops-research.com/research.html#reports

2 https://itrevolution.com/book/accelerate

Is a solution that can be written with a 100-line piece of code better than a 200-line piece of code? On the surface it would seem so, but which solution is easier to maintain over time? Which solution is clearer and easier for new developers to discover and figure out? It may be that the 200-line piece of code is much easier to maintain and learn for the team rather than the expertly crafted bit of black magic code that only the expert understands. If a team is 100% utilized on feature development, when do they get time to learn, do unplanned work, or manage technical debt? Not having time for these activities ultimately slows delivery and innovation to a grinding halt. If a team delivers 10 stories per sprint, is this better than a team that delivers 5 stories per sprint? Can we really compare different teams' output when they work on unrelated work items? Probably not. Are the user stories the same size and value for our business and end customers? It is not easy to judge across teams with different products and ownership.

These measurements are what we call output-based. We can instead shift the focus from what a team produces more toward the goals or Target Outcomes. By focusing on the global outcomes, the team members aren't pitted against each other by measuring the wrong thing. A classic example is rewarding Development Team members for throughput on new features and Operations Team members for service stability. Such measurements incentivize developers to throw poor-quality code into production as quickly as possible and operations to place painful change management processes in the way to slow down change.

By not focusing on measurement or the right things to measure and focusing on outputs rather than outcomes is a quick way to get into trouble. Luckily, the DORA report starts to lay out some of the key metrics that a team can use for SDO:

Figure 13.6: DORA – Performance metrics

Software Development Metrics are metrics that capture the effectiveness of the development and delivery process by measuring the following:

- **Lead Time**: The time code is checked in to when it is released into production
- **Deployment Frequency**: How often the team can release code into production

Lead time is a key element of Lean Theory: by shortening the amount of time it takes to get product features to end users, the team can shorten the feedback loop for learning what their end users like or do not like. If the team is building the wrong thing, they can correct their course (or pivot) quickly when they have short lead times.

Another element to this is the size of the work delivered, or **Batch Size**. By delivering small incremental value to end users quickly, lead times are kept lower. With a much faster delivery cadence, how do we make sure system stability does not suffer?

Software Deployment Metrics are the metrics that capture system stability and release quality. They are measured by the following:

- **Time to Restore**: The time taken from detecting a user-impacting incident to having it fixed or resolved
- **Change Failure Rate**: The number of released changes that fail or result in a user-impacting incident

What percentage of the changes made to production fail? How long does it take to restore service to end users in the event of failure? Development teams will have happier customers if outages are fixed quickly when changes occur and ideally have changes that do not fail at all!

Service Operation Metrics captures operational performance via metrics called **Service Availability**. Assume your product sales website has crashed. The company accountant might ask the question, *Is our application generating revenue right now*? Measuring service availability is a good way to link a technical goal with the desired business outcome.

The metrics defined so far go a long way in helping a team understand key software, delivery, and operations metrics that are desirable for better organizational outcomes. To help capture and radiate these metrics, Red Hat has been investing in an open source project that has resulted in a dashboarding tool called Pelorus[3].

3 https://github.com/konveyor/pelorus/

Pelorus

Pelorus is an executive dashboard that helps visualize the progress we make on the SDO success metrics. It makes use of open source tools such as Prometheus and Grafana to track progress both locally (within a team) and globally (across the organization):

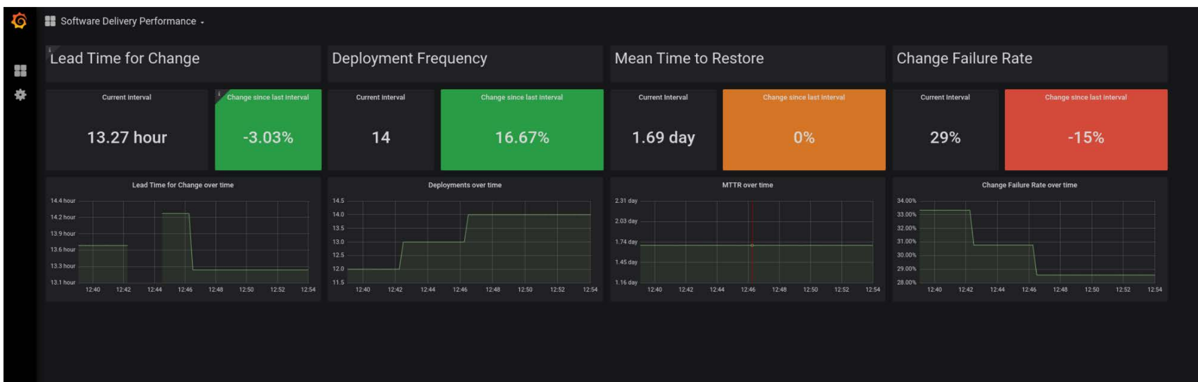


Figure 13.7: Pelorus – SDO dashboard and metrics

Pelorus consists of a set of exporters to customize data points to capture metrics from different providers. The sources from which exporters automate the collection of metrics are growing as more people contribute. This currently includes OpenShift (as Deploy time exporter), Git providers GitHub, GitLab, and Bitbucket (as Commit time exporters), and JIRA and ServiceNow (as established Issue trackers).

Using the data points that are collected from the providers, metric indicators are calculated to represent a measure. Each outcome is made measurable by a set of representative measures: **Lead Time for Change**, **Deployment Frequency**, **Mean Time to Restore**, and **Change Failure Rate**.

Pelorus offers a great opportunity to have real-time information radiators next to teams giving out these important metrics. Teams can also regularly inspect and discuss these at Showcase and Retrospective events and ask themselves what improvement actions or experiments they can run to try and improve these metrics further. What else should we measure?

Measuring Lean Metrics

The Lean movement also proposed a number of metrics to help measure software delivery performance. In their simplest form, you only need two pieces of information per work item or story: the start and finish date. This sounds easy to measure! But, of course, you need some policy around what those definitions are. With these two measurements we can start to measure a lot of different things, in particular:

- **Time in Process**: The units of time per unit of work.
- **Lead Time**: The amount of time between when a feature is requested to when it is delivered into production.
- **Flow Efficiency (or touch time/lead time)**: The time that the product is actually being worked on by the team, and value is being added.
- **Due Date Performance**: How often the feature gets delivered on time.

Most of these metrics are part of what is known as **lagging** indicators for performance. In other words, they measure what has already happened. It is also possible to formulate **leading** indicators that can help predict future performance from this data. One such measurement example is based on the flow of work items or stories into and out of a team. This net flow of work items allows us to predict the confidence of due date delivery. As teams accept more and more work, this slows down their ability to deliver items on time. So, the net flow of stories in and out of a team becomes a lead indicator for measuring work item delivery performance.

It is surprising what start and end dates can tell us. The distribution of items with time can also help categorize the type of work undertaken by a team. For example, normal work items may look different to priority work items that come to the team from production failures. Another example may be work that requires long wait times for approval from, say, security or compliance teams. These would have large lead times compared to normal work items that don't require such approval.

Measuring SLOs, SLAs, and SLIs

The goal of these **service level (SL)** metrics is to get customers, vendors, and users all on the same page regarding system behavior and performance. In particular, everyone needs to know and agree upon common SL questions, such as the following:

- How long for and how often will the system be available?
- If there is an outage, how quickly will the response be to restore the service?
- How fast will the system respond if we make a single request?
- What about if we make many concurrent requests?
- Users of a service want to know the answer to these questions so they can plan and design how they consume any given service. For example, if you are building a solution that must be available all the time for your end users and it depends on a third-party service that is only available during business hours, you may need a different design, to implement a caching solution, or use another service with higher availability.

The SL acronyms can be broadly defined as follows:

- **SLA**: An SL agreement that is normally a formal contract made between your organization and your clients, vendors, and users.
- **SLO**: SL objectives are the outcomes your team must deliver to meet the agreement.
- **SLI**: SL indicators are the actual metric numbers that are used to measure a team's performance.

It can be very hard to measure SLAs properly. A service may be available but with degraded service performance, for example. It can also become more complicated when only some portion of your users experience a partial outage. Capturing this type of SLA complexity and measuring it accurately is hard to do.

One benefit of SLAs is that they allow IT managers to quantitatively measure business outcomes. So rather than having to deal with a generic qualitative complaint, such as, "My application won't load and it is very slow", they can measure application availability (uptime) and percentile page load speed instead.

An SLO is an agreement about a metric within a given SLA. A simple example may be that we agree to return search results quickly to our end users, with an average search latency of 200 milliseconds. In contrast, the SLI typically measures the SLO. So, we might define a target upper bound for our search, for example, by specifying that search latency for 99% of all performed searches must be less than 300 milliseconds.

By quantitatively specifying and publishing SLAs, SLIs, and SLOs for our services, end users can set their expectations on how well the service will perform. This prevents qualitative complaints about the service being slow or over-reliance on a service where users expect it to be more available than it actually is.

PetBattle Service Levels

The current PetBattle hobbyist application has no SLAs – zero, zilch, nada. PetBattle V2 is going to be built for high availability. Given the app's popularity and the fact it is the sole source of potential income for our fledgling company, it has been designated as "mission-critical". Redundancy is going to be built in at every level: at the cloud infrastructure layer (network, storage, and compute), and the application layer (scalable redundant highly available services). PetBattle V2 will be re-designed to

have a shared-nothing architecture to allow maximum horizontal scalability without bottlenecks. PetBattle V2 will be required to meet a 99.99% availability SLA. That's slightly more than 4 minutes of downtime allowed per month!

Any external system that has a lower availability SLA will negatively impact the PetBattle V2 SLA when it fails. The PetBattle team writes the code and runs the service, and ideally there are as few external service providers in the Development and Delivery Loop as possible. Any new service that is incorporated into the application must be designed to meet these SLAs.

By quantitatively specifying and publishing SLAs, SLIs, and SLOs for our services, end users can set their expectations on how well the service will perform. This prevents metrics-driven qualitative complaints about the service being slow or over-reliance on a service where users expect it to be more available than it actually is.

Measuring Security

Malicious users are out there. How can we ensure that the user data that is kept in our applications is not misused? Or that our application services are not put to unintended usage, causing organizational or reputational harm? It is commonplace to see data leaks and security breaches related to software applications and services in the media today. Answering these questions is most often the primary concern of an **information security** (**InfoSec**) analyst or team of people. A modern approach to help tackle these security concerns is called **shifting security left**. The term is associated with teams that build InfoSec into the software delivery process instead of making it a separate phase that happens downstream of the development process.

Building security into software development not only improves delivery performance, but also improves security quality. By designing and implementing security and compliance metrics into the system, it becomes possible to measure continual compliance against security standards:

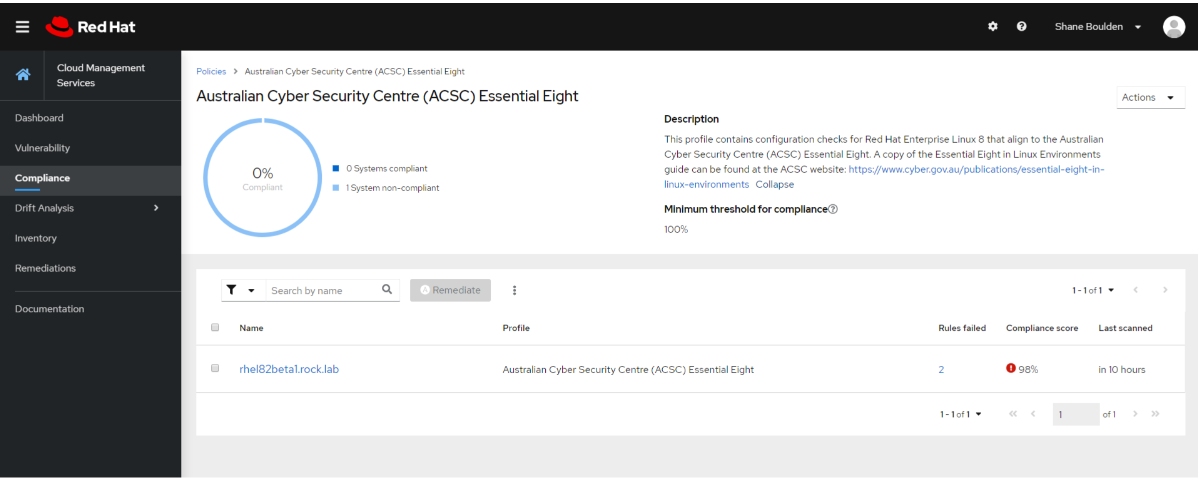


Figure 13.8: Standards-based system compliance

PetBattle Security

The PetBattle team has asked the founders to hire an InfoSec professional who can be part of the development process. They are worried about potential data breaches of user details in PetBattle V2, especially if we need to start collecting payment details when we start monetizing the site. The PetBattle team has educated their developers of the common security risks, such as the OWASP Top 10 and how to prevent them.

The team plans to implement security scanning and testing measurement into their build pipeline. By using the OpenShift platform with a trusted software supply chain, the team can spend significantly less time remediating security issues that may arise.

The team has identified that everyone needs to become very familiar with the detailed network design of the overall solution. This should help avert attacks by malicious actors. The team also wants to make sure all of the platforms and applications are easy to patch—so they can easily keep frameworks up to date when **Common Vulnerabilities and Exposures (CVEs)**[4] arise.

4 https://cve.mitre.org

Having a secure system involves making sure all of the layers in the stack are themselves secure – having a secure hardware environment, securing the operating system, securing the containers image layers being used, securing the dependencies that your application uses, securing your application code, securing the network that exposes your application services, and ultimately ensuring that your end users can interact with your applications securely.

Measuring these security layers, making sure they comply with various industry standards, and having concrete action plans for when new vulnerabilities arise, requires teams to work together at every stage of the software delivery life cycle. Security should not just be the **Chief Information Security Officer's** (**CISO's**) job. Done right, security is pervasive and designed into the platforms and software systems, with security professionals actively participating as part of the core delivery team and not just being the Mr. No[5] when penetration testing is carried out.

Security is one of those topics that needs its own book. By shifting security left, we will cover technical topics later in the book that include container image and vulnerability scanning, the container health index, CVE patching, OpenShift Compliance Operator, OpenShift Container Security Operator, and Security Policy enforcement with ACM. These tools can help you to complement and build out a continually compliant platform and application suite for yourself.

Measuring Performance

There is an adage that software developers use—*first make it work, then make it work fast*. The truth to this statement is that some functionality must exist before it can be made to work fast! Qualities such as the performance and security of our applications are usually the most important non-functional requirements and they must be designed into the system from the very start in order to be successful. Let's define what we mean by performance.

Performance measures how fast the system processes a single transaction. This can be measured in isolation or under load. The system's performance has a major impact on its throughput. When end users talk about performance, what they are usually talking about is throughput; and they only care about the performance of their own transactions, not anyone else's. As far as they are concerned, if a system's response time exceeds their expectation, the system is down.

Throughput describes the number of transactions the system can process in a given time span. The system's performance clearly affects its throughput, but not necessarily

5 *Mr. No* is a *Mr. Men* book only available in France. Mr. No always disagrees with everyone and everything. *Mr. No* is published under the title *Monsieur Non* in France. This is one of the two Mr. Men titles that were not published in English.

in a linear way. Throughput is always limited by a constraint in the system, what is known as a bottleneck. Trying to optimize or improve performance for any part of the system that is not a bottleneck will not increase throughput.

If we measure the number of end user requests, the throughput will vary depending on how many requests there are. This is a measure of scalability. When a system is horizontally scalable, it means we can add capacity (more servers, more pods, or containers) to handle more throughput. In a shared-nothing architecture, we can add capacity until we reach a known bottleneck. For example, in OpenShift, this may be the number of pods per node, or the maximum number of nodes per cluster.

It is also worth noting that for any given system, an "acceptable response time" may vary! For a mobile/web app, any response longer than a second or two means users will use their fingers to walk away to another website or app. For a trading system at a bank, the response time may be in the order of milliseconds or less. To understand the capacity a system requires, we need to understand the system as a whole, before breaking it down into its constituent parts. This is called *systems thinking*. By thinking holistically and then in detail about a system, we can determine the bottleneck in a system.

At any point in time, exactly one constraint determines the system's capacity. Let's imagine it is the database that limits the transaction throughput. Once we improve that bottleneck—for example, by using faster storage or adding indexes, or using better database technology—the next bottleneck in the system then becomes the performance constraint—the application server capacity is now limiting throughput, say.

PetBattle Performance

When PetBattle is released to the world, users will gang up on us. Sometimes, users come in really, really big mobs. Picture the Instagram editors giggling as they point toward the PetBattle website, saying "Release the cat hoard!" Large mobs can trigger hangs, deadlocks, and obscure race conditions in our applications.

The PetBattle team wants to run special stress tests to hammer deep links or hot URLs within the UI and API layers. Currently, there is a direct coupling between users and the database, so the developers already know that some form of caching could be required when scaling.

If we make monitoring a priority, we can refine our infrastructure and application monitoring systems to ensure we are collecting information on

the right services and putting that information to good use. The visibility and transparency yielded by effective monitoring are invaluable. Proactive monitoring is a key part of a strong technical foundation.

The team plans to deploy the PetBattle application suite into an environment where they can start performance testing parts of the system under load. They want to start simply by targeting various parts of the system, for example, the API to gain an understanding of system behavior under load. By doing this early and often, they can identify bottlenecks and from there work out a plan to fix them.

Like security, moving performance testing left in the build and deployment process provides faster feedback and the opportunity to remediate and reveal issues early in the development life cycle. Complex system testing for a full suite of applications is often not feasible until closer to the end of development; however, it is possible to test individual APIs, endpoints, or even parts of the UI often and early in an automated manner. One of the main benefits of doing this type of automated performance testing is to build up a baseline understanding of how the system performs. Any code changes that alter the system performance characteristics often go unnoticed if this type of testing is not being automated and you end up trying to firefight performance issues right before a go-live.

Measuring Deployment Pain

If you have ever had to work night shifts or weekends to help deliver a Big Bang release of software products into a production environment, you will understand the feeling of anxiety and concern engineers and technical staff feel prior to a go-live event. In the literature, this is referred to as deployment pain, and it can highlight the disconnect that exists between common development and testing tasks when compared to release and operational tasks.

Fundamentally, most deployment problems are caused by a complex, brittle deployment process where the following occurs:

- Software is often not written with deployability in mind.
- Manual steps are required to deploy to production.
- Multiple hand-offs are undertaken in deployment phases.

Disruptions are a fact of life in all systems, OpenShift included. There are a host of

Kubernetes primitives that will help our business services stay up and available for customers (including replication controllers, rolling deployments, health checks, pod disruption budgets, horizontal pod autoscalers, and cluster autoscalers). But even with the best infrastructure, failures can still occur. Cloud service disruptions, hardware failures, resource exhaustion, and misconfigurations can still threaten the business service.

Measuring Culture

There is a link between metrics and transformation. Measurements help inform the practices that are used. Gathering CI/CD metrics, for example, the time that builds and deployments take for the PetBattle apps, allows the team to adopt and develop streamlined CI/CD practices so they can release quickly, more often, and with confidence that failures will be rare. As the team becomes comfortable releasing software into production regularly, the business starts to trust that releases can be performed with little risk to end users. This leads to the faster releasing of new features, a tighter, shorter feedback loop, and ultimately, a culture that encourages rapid change and innovation. Now we have really started to unlock transformational change for the company:

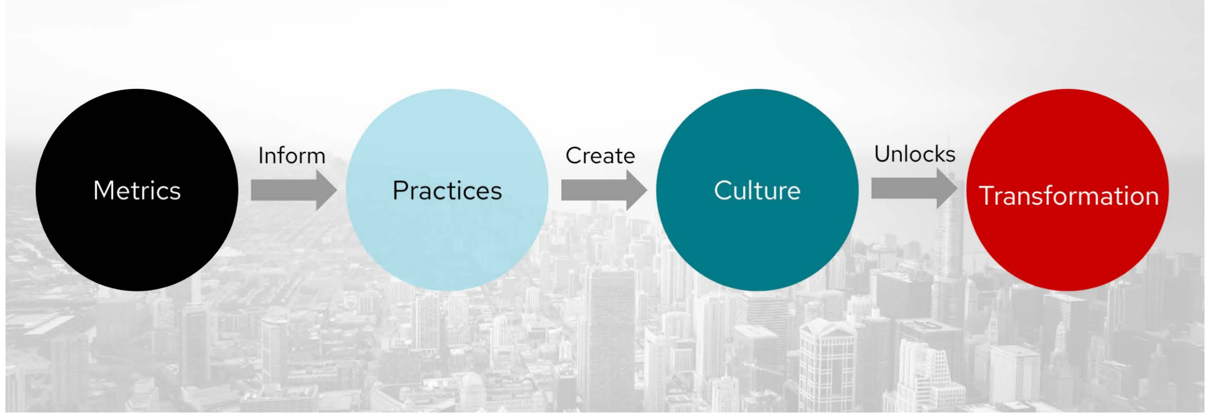


Figure 13.9: The link between metrics and transformation

Operational metrics often measure failure in very complex adaptive systems. When things fail, an authoritarian blame-like culture will look to find the "human error" and assign blame. This type of response to failure is not only bad but should be considered harmful.

The goal of any failure analysis should be to discover how we can improve information flow so that people have better and more timely information, or to find better tools to help prevent catastrophic failures following apparently normal operations.

Measuring Application Metrics

All of the metrics collected so far are focused on non-functional parameters. What about the business application? Can we get some real-time metrics about the application usage that we can radiate alongside these other metrics?

PetBattle Application Metrics

The application developers have discovered how easy it is to generate custom application metrics by utilizing the MicroProfile metrics extension in the PetBattle API server. The founders wanted to know how many cats were being uploaded in real time. The developers added the metrics extension to their application, which provides default Java VM metrics such as heap, CPU, and threads. It also allows the team to expose custom metrics from the application. For example, by adding Java annotations to their code, they can easily measure the frequency of cats uploaded.

These metrics can now be scraped and displayed on a dashboard using Prometheus and Grafana:

Figure 13.10: Analyzing PetBattle application metrics using Prometheus and Grafana

There are internal metrics we can monitor as well as external metrics. An external fake user (often called a synthetic user) can be set up to monitor the PetBattle app on a regular basis. This synthetic client experiences the same view of the system that real users experience. When that client cannot process a fake transaction, for example by trying to enter their fake cat in a Tournament, then there is a problem, whether the internal monitoring shows a problem or not!

There are lots of possible stability metrics the team thought they could easily measure and alert on if they were over a certain threshold. Slow responses tend to propagate upward from layer to layer in a gradual form of cascading failure. When the website becomes slow, users tend to hit the **Refresh** button more often, causing more and more traffic. If we give our system the ability to monitor its own performance (in other words, it becomes observable), then the system can also tell the team when it isn't meeting its SL agreements. Some examples include the following:

- The number of users and active sessions
- Blocked threads in the API layer
- Out of memory events in the API or database layer
- Slow responses in the user interface
- High database CPU

Measuring Infrastructure Platform Costs and Utilization

OpenShift comes with a cost management and metering application that can be used to show infrastructure usage. You deploy the Cost Management Metrics Operator in OpenShift and the reporting and APIs are delivered as part of a SaaS solution from cloud.redhat.com.

It allows the PetBattle team to do the following:

- Visualize, understand, and analyze the use of resources and costs.
- Forecast their future consumption and compare them with budgets.
- Optimize resources and consumption.
- Identify patterns of usage that should be investigated.
- Integrate with third-party tools that can benefit from cost and resourcing data.

There are a lot of visualizations and dashboards available. *Figure 13.11* shows the overview dashboard in a large demo environment as an example. It is possible to track cost and usage at both an infrastructure and business level. Users can tag projects and applications to gain detailed breakdowns as well as historical trends. The dashboards can help answer common questions, for example:

- Show me the top project and top clusters by usage and cost:
  - Which projects are costing me the most?
  - Which clusters are costing me the most?
- Show how metrics contribute to the costs:
  - What is driving costs? CPU, memory, storage?
  - What are the predicted costs for the next calendar month?

It is possible to change between accumulated and daily costs, as well as filter and drill down across clusters, clouds, and projects:

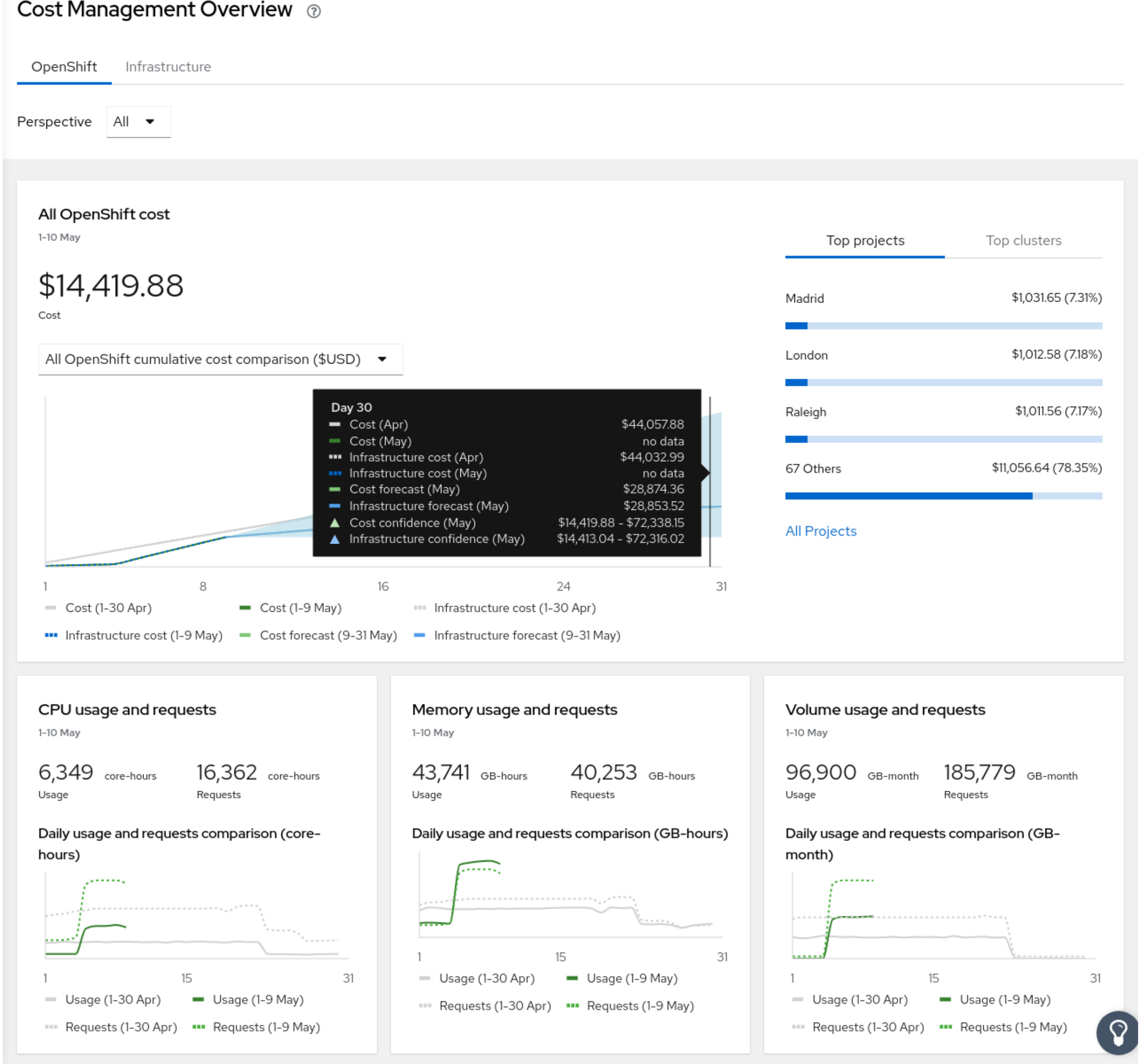


Figure 13.11: Cost Management Overview dashboard at cloud.redhat.com

You can check out the cost management product documentation,[6] which has a lot more details about this service, including common configuration options across a hybrid cloud.

6 https://access.redhat.com/documentation/en-us/cost_management_service/2021

Measuring Resources and Services

There are a number of simple high-level checks that should be made for all of our resources and services. The **USE** method is a simple checklist that can be summarized as *for every resource, monitor the following items*:

- **Utilization**: The percentage of time that the resource is busy.
- **Saturation**: The amount of work a resource has to do that is extra or overloaded, often a queue length or similar.
- **Errors**: The count of error events that occur.

Errors should be investigated because they can degrade performance, and may not be immediately noticed when the failure mode is recoverable. This includes operations that fail and are retried, and devices from a pool of redundant devices that fail:

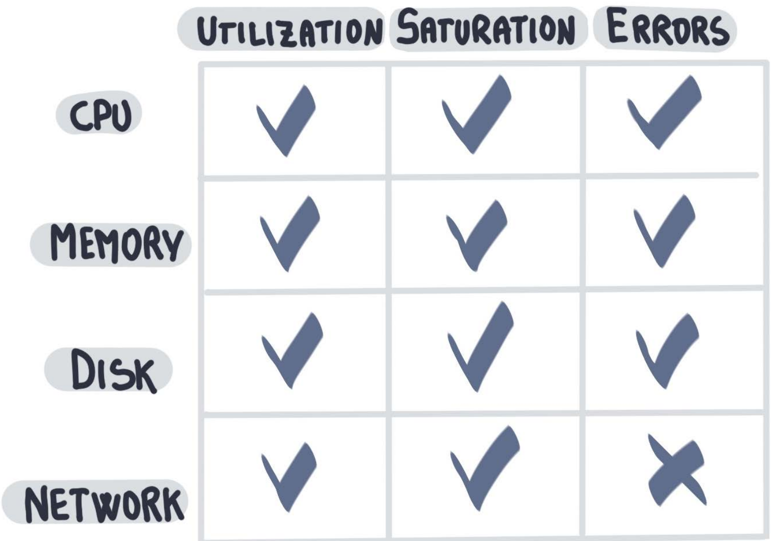


Figure 13.12: USE metrics

We can map the USE metrics against common resources in a tabulated form, as shown in *Figure 13.12*, allowing for quick identification of the types of issues that are occurring in our system. The OpenShift metrics stack supports pre-configured USE method dashboards at the cluster and node level:

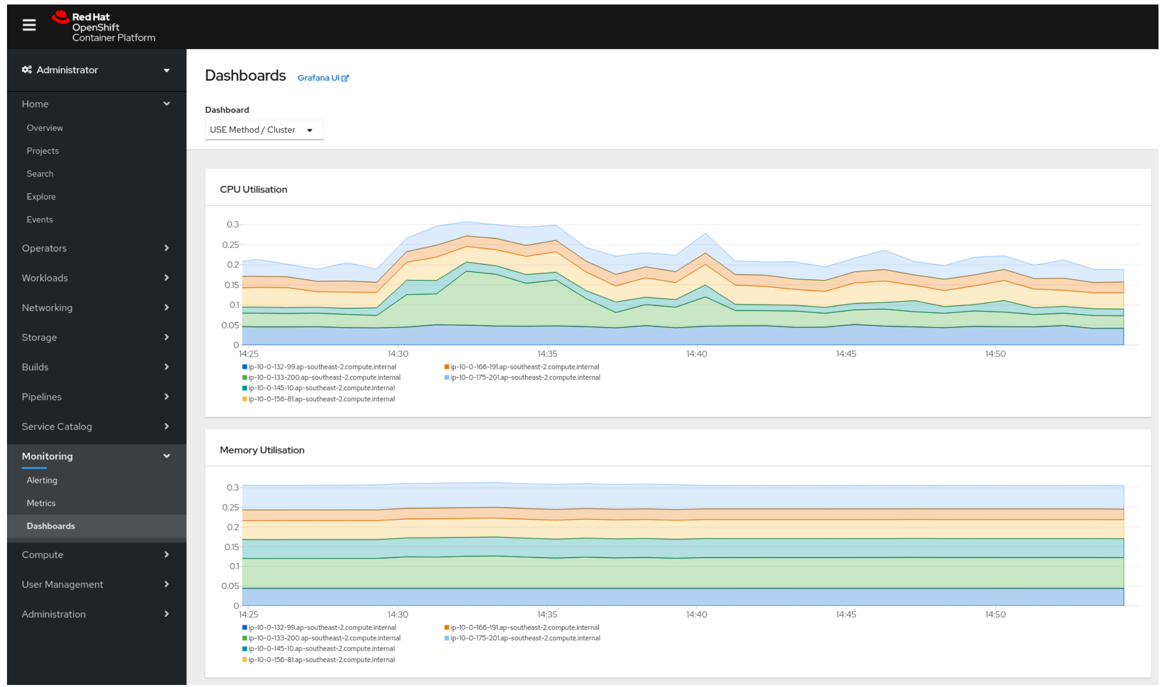


Figure 13.13: OpenShift Monitoring USE dashboards

Similarly, the **RED** method can be summarized as *for every service, monitor the request*:

- **Rate**: The number of requests per second
- **Errors**: The number of requests that fail
- **Duration**: The amount of time requests take to complete

The RED method is a good baseline that can be applied to most request-based services. It reduces the cognitive load for SREs; in other words, they can think the same way about a large range of supported services for baseline metrics. The RED method does break down for batch-oriented or streaming services. In the Google SRE Book, the original "Four Golden Signals" included the RED and Saturation metrics.

User Experience Analytics

There is a vast treasure trove of user analytics that can be sourced from the PetBattle user interface. The most obvious analytics are to do with non-functional performance measures, for example, page load times and response latencies. Hopefully, by measuring and baselining basic user interface performance, the team can prevent the application services from being trampled by a flood of customers!

Standard practice is to measure using histograms (percentiles), which allow for a better understanding of outlier performance. Reliably measuring and aggregating quantiles/percentiles of high-velocity metrics from multiple sources is no easy task.

PetBattle User Experience Analytics

Given the primary goal of monetizing PetBattle, signing up to one of the many global businesses that provide web analytics to help sell targeted advertising to end users is likely a better use of resources than trying to build a custom analytic and advertising solution from the ground up. Analytics services can quickly give us page load times, click-through rates, advertisements, site search optimizations, and suggestions, as well as help identify upsell opportunities – for a fee.

As part of testing the end user experience, an automated load testing of the user interface can give the team a preview of what operating the PetBattle V2 site would look like.

We have shown in this section many different levels of quantitative analysis we can perform using metrics. Let's now see how these translate to our focus on outcomes.

Visualize Measurable Outcomes

We have a lot of things we can now wmeasure. How can we tell if we have *shifted the needle* and made an appreciable difference to the status quo? Quite often, the key metrics and data are not visible to everybody in the room; they are hidden away behind a login on a computer.

To help solve this hidden data problem, we can make use of more information radiators, putting everything on walls around where the team works, using dashboards and large screens so we can visually represent all aspects of the delivery work. We can share all sorts of information that is useful to team members, stakeholders, and users alike. It can be physically presented on walls, windows, doors, and other flat surfaces and positioned in the line of sight of those people who will get value from consuming the information.

There are some interesting consequences of making information more accessible. These include better speed and accuracy of teamwork. Because the information is now visible all of the time, people are frequently reminded of it. There is also less time wasted having to search for important information if it is constantly on display. It is also more likely that the information is accurate because people are continuously being reminded of it, and if it is inaccurate, questions get asked!

When people who are not part of the core team come into the team space, information that is radiated on the walls can instantly be read. Stakeholders and people interested in the team's work can immediately gain a better understanding and awareness of the work that is going on. This activity is often referred to as *walking the walls*. Interested parties can inspect artifacts on the walls and have a conversation with team members about them. This is a hugely different experience from when information is hidden away in a system behind a login.

Proactive Notification

So, what happens when things start to go wrong? What happens when the system itself cannot resolve an issue by automatically restarting a pod, or scaling up cluster nodes? Enter the realm of **Alerts**.

Alerting can take many forms. It could be that horrible text at 03:00 when things really start going wrong, or a more subtle alert, such as a Slack message to say something has updated successfully. The key here is that the information is being pushed, and not pulled. OpenShift has alerting capabilities built into it and we'll explore this in more detail in *Chapter 16*, *Own It*.

The classic forms of alerting are when there are spikes in memory usage in applications. This could lead to an application failing or constantly having to restart. In these instances, the team might spot the spikes on their dashboards and go and investigate the issues. We can, of course, make this feedback loop even shorter by combining data from different sources and alert on that. For example, if our application memory spikes, if we could capture the logs from around that time and push both events to the team, it could help diagnose the problems quicker. The real power of smart notifications is being able to respond even more quickly to the event.

Alerting the Development teams to things that have broken is extremely important. Notifications can come from all layers of the system; it doesn't just need to be that terrible call in the dead of night to say the website is down! Whenever a job runs to build our code or deploy a new version of an application, sending a quick alert to the team's instant messaging software is a good way to notify the concerned stakeholders. If the information is timely, then we can respond more effectively. This could mean pulling the Andon cord and halting the production line while we gather together to fix the issue.

Altering the Customers

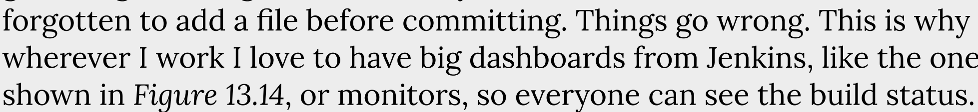
The software we rely on daily sometimes experiences downtime. Notifying the developers and SREs of critical failures is vital to resolving issues. Notifying customers in a transparent way of any system failures and your path to resolution can set you apart from the crowd. Lots of companies have status pages for their software to give an idea of whether something is up or down; for example, the Quay container repository status[7] page gives you insight into what parts of the service are still working. This provides a great information radiator! Some of the best software companies in the world will not only publish the status of their downtimes, but also the steps they are taking to resolve issues in near real time. Companies such as GitHub and GitLab will post messages to Twitter or open up a Google Doc to post updates and provide as close to real-time updates as they can to their user base. This level of transparency is great to see, especially when the issues they are facing can be discussed further by others using the same technology. They can become proactive, and not make the same mistakes on that next big critical upgrade!

Alerting is only useful if it happens in a timely manner. It's no use alerting on a condition if no one can do anything about it because it's too late. It's often a bit of a balancing act to design alerting thresholds so that the humans who look after the product or systems can proactively take action, as opposed to pinging alerts too frequently by "crying wolf" when a limit is reached.

Having Fun with Notifications and the Build!

As a self-confessed lazy developer, I know that when I push some code, it's unlikely that I will check the results of an automated build. I know this is arrogance, but hear me out. Usually, my thinking is, It works for me locally, what could possibly go wrong in the build?

The truth is that often something can, and does, go wrong, I've forgotten to link my code or I've forgotten to add a file before committing. Things go wrong. This is why wherever I work I love to have big dashboards from Jenkins, like the one shown in *Figure* 13.14, or monitors, so everyone can see the build status.



7 https://status.quay.io/

They are such an important element of the feedback loop. I look at a dashboard like this and I can immediately see critical information that makes my job easier, such as what step in the build is running, who triggered the build, or how many tests are failing! It's simple information, but it's next to impossible for me as a lazy developer to ignore something if it's flashing red in my face like this!

I love anything that can help shorten the feedback loop. I am particularly fond of a plugin that works well with a dashboard in Jenkins, called the Build Fail Analyzer. It parses the build log for a specific Regex and, if found, it can display a message. On the project, this screenshot is taken from the team who had gotten into the habit of trying to codify the failure issues, so when one was detected, it could prompt us to resolve it without looking through the log.

On this dashboard, on the middle left side, I can see that the **dev-portal-fe-e2e-tests** testing suites have failed and the problem identified was selenium not started. With this information, I don't need to open Jenkins and read the log to see what happened – I can go straight to OpenShift to see why the pod did not start:

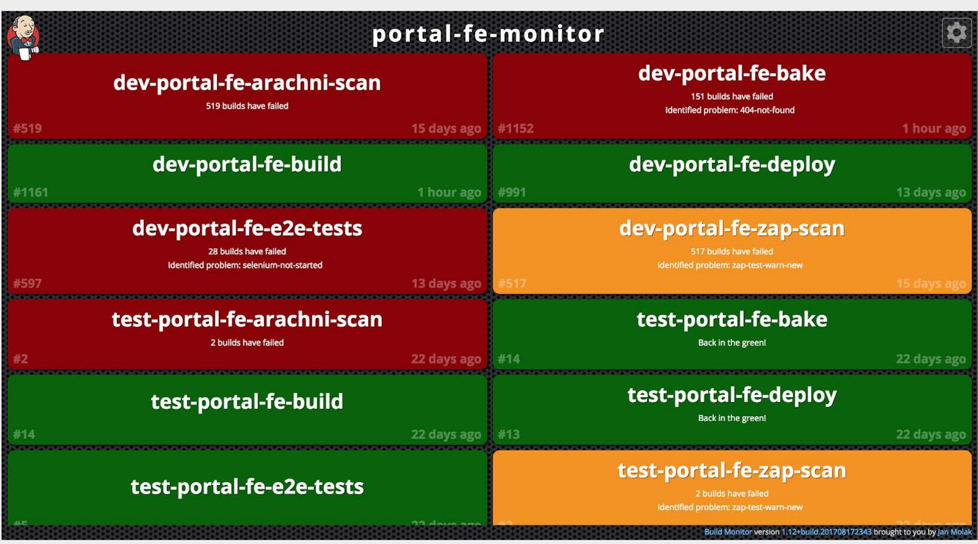


Figure 13.14: The Build Fail Analyzer in Jenkins

Failing the build is seen sometimes as the cursed state. We try to encourage a no-blame culture, but sometimes it can be a bit of fun to have a bit of a blame game. I am a fan of this one for sure! Whenever you fail the build, you're deemed to be the Cowboy Coder, the one who rides into town and shoots from the hip without a care for the consequences. Or maybe you just left a typo in your code!

Either way, if the dashboard turns red, then you have to dress like the cowboy you are. This team took it one step further: not only did you have to wear the pink cowboy hat until you had fixed the problem, but you also had to ride around on the wooden hobby horse! Even if you had to go to the bathroom or get a coffee, the horse and hat went with you! You'd be amazed at the funny looks you get on the way to the canteen wearing this attire:

Figure 13.15: The Cowboy Coders

More silly examples come from a project I was working on years back. This was in the days prior to using a container platform, and we had several manually configured VMs that were critical for us when releasing software. It was a massive project, with seven Scrum teams globally distributed. We were building a suite of 50 product microservices so our build farm was always busy!

On one occasion, we had to get the IT company that was managing our infrastructure to roll back to one of the previous backups, as someone had executed a large number of commands as root and broken several things for Jenkins. Raising a sev1 ticket still took a few days to resolve! We couldn't let this kind of thing happen again as it was a massive drain on the team's morale and productivity. So, we decided to get inventive. We were using Slack as our messaging client and knew you could send messages to channels via a webhook. We also knew that if anyone logged into a machine, we could execute a bash script. Tying these items together, we created the Kenny Loggins channel in our Slack instance...because when you log into a server as root, you're in the DANGER ZONE!

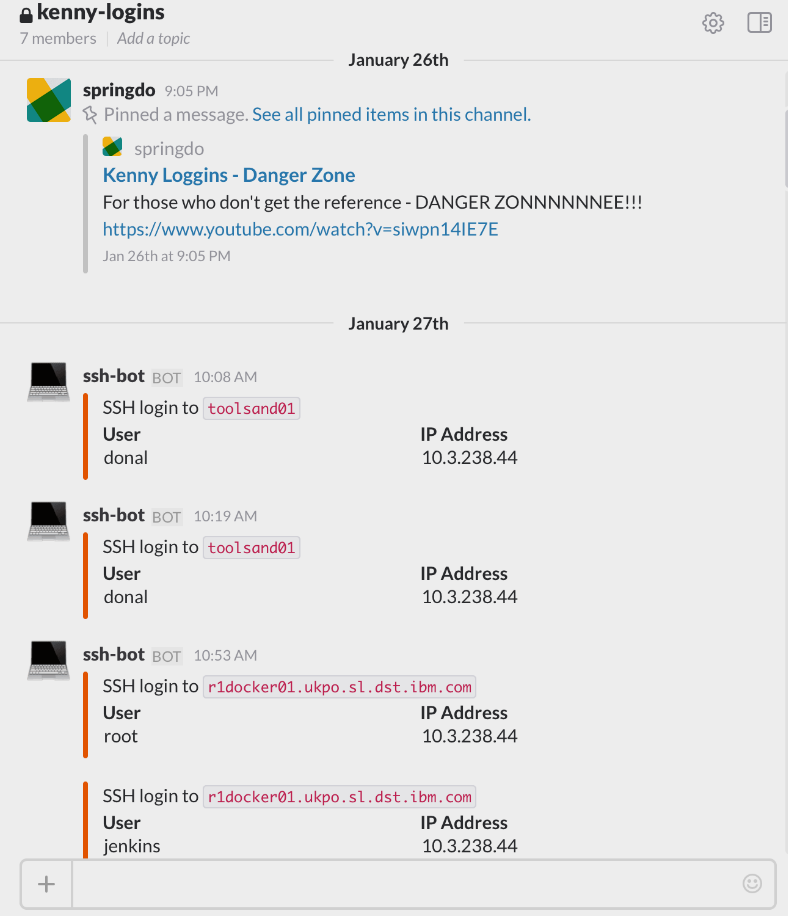


Figure 13.16: The Kenny Loggins channel

This section has shown many different ways we can visualize outcomes and use metrics to trigger proactive notifications to learn faster. Let's see how this can be summarized with everything else we've learned in our Delivery Loop.

Creating a Delivery Map

We concluded *Section 3*, *Discover It*, with a Discovery Map, with a single information radiator that summarized the Discovery Loop iteration. We concluded *Section 4*, *Prioritize It*, with an Options Map, which summarized the ideas and hypotheses we wanted to validate, how we would deliver the options, and what options we planned to work on first.

We will conclude *Section 5*, *Deliver It*, with a Delivery Map. This is another open source artifact available in the Mobius Kit under Creative Commons that you can use to summarize all the learnings and decisions taken during your journey around the Delivery Loop.

This map should slot neatly next to your Discovery Map and is used to summarize the following:

- **Actions**: What can we get done this week to improve the outcomes?
- **Doing**: What is our current work in progress?
- **Done**: What is ready to review?
- **Impact**: What progress did we make toward the outcomes?
- **Learn**: What did we learn?
- **Insights**: What are our next steps?

As we come out of the Delivery Loop and return to the Options Pivot in *Section 7*, *Improve It*, *Sustain It*, we will complete the final section of this map by asking, *What are our next steps?*

Now let's look at PetBattle's Delivery Map at the end of their first iteration of the Delivery Loop.

PetBattle – the Delivery Map

Figure 13.17 has a lot of detail that may not all be readable in print. To explore it in full, you can access the image at the book's GitHub repository at https://github.com/PacktPublishing/DevOps-Culture-and-Practice-with-OpenShift:

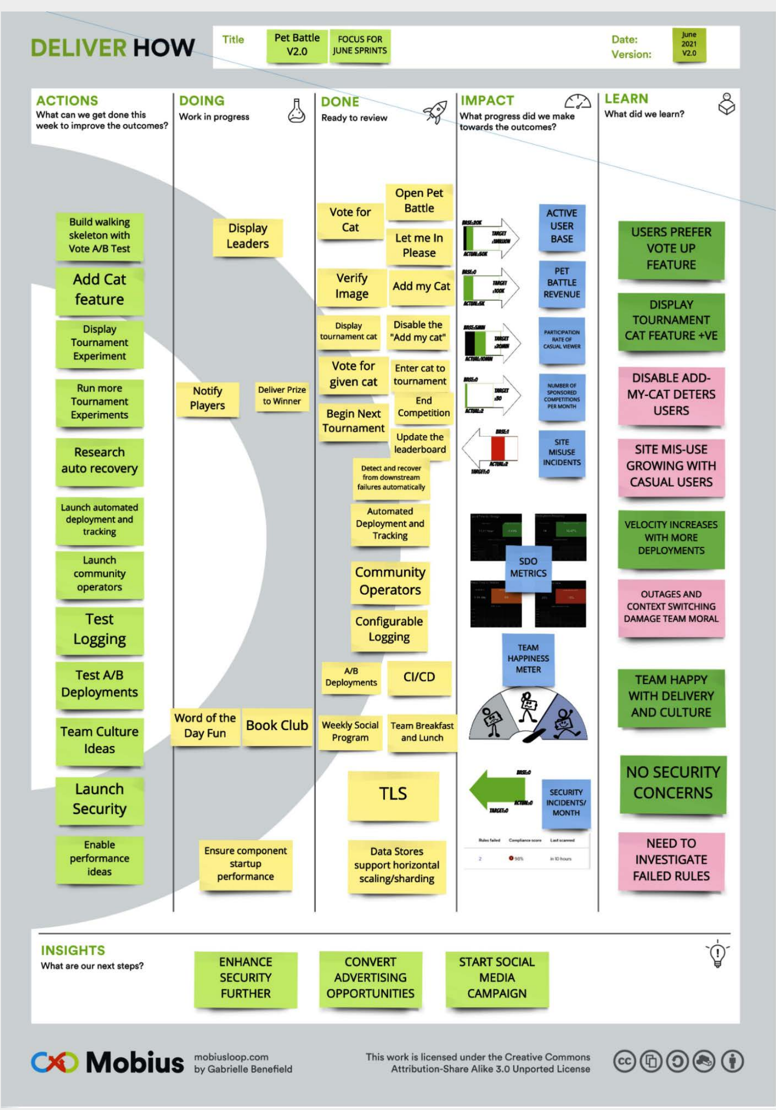


Figure 13.17: The PetBattle Delivery Map

The Delivery Map provides a powerful summary of the journey we've been round the Delivery Loop. Like all other artifacts, it is a living and breathing summary and should be revisted regularly and updated after every subsequent iteration.

Conclusion

We have now completed our journey around the Mobius Loop. In this chapter, we have focused on the measurements and learning we can take away from the features we launch, the experiment we run, and the research we conduct. A relentless focus on measurement enables us to take more concrete decisions that are backed by metrics-based evidence.

The Showcase and Retrospective events that are often run by Scrum and other Agile teams provide ample opportunity to showcase metrics and highlight learnings. We take this opportunity to re-examine the experiments that we designed on the Options Pivot and investigate what actually happened. That often involves looking at the analytics provided by the advanced deployment capabilities offered by the platform – the results of A/B tests, canary launches, feature toggles, and dark launches.

We also highlighted the importance of running usability tests with the full team involved, while being connected directly to end users to develop further empathy and see them testing the evolving application. Guerilla testing also provides a low-cost and simple way to gather learning from users:

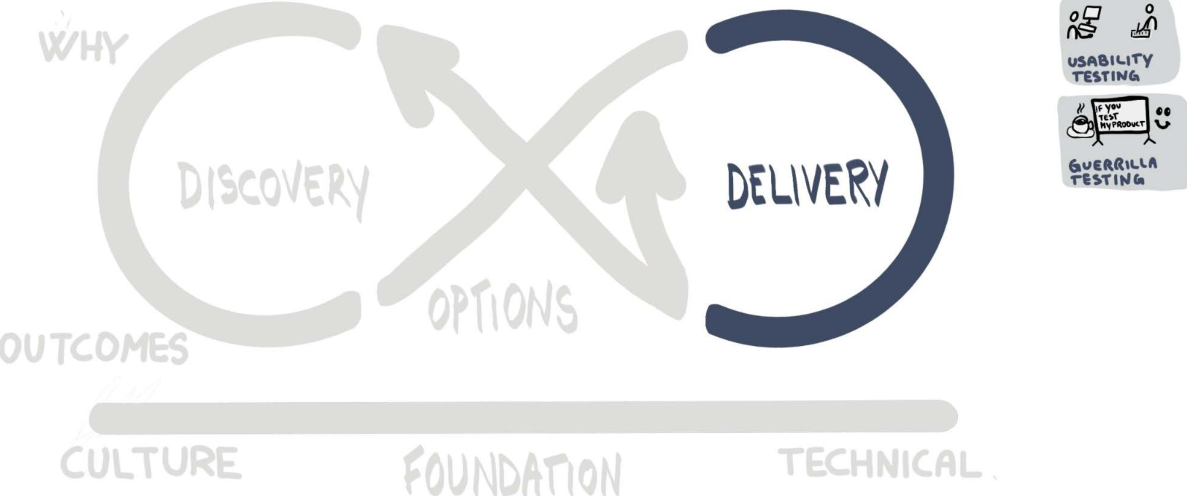


Figure 13.18: The Delivery Loop

We explored the many different metrics made available by the platform, the software, and our teams. Service Delivery and Operational Performance metrics popularized by DORA and Accelerate, and made available by open source tools such as Pelorus, provide leading indicators of the success of DevOps culture and practice. These can be supported by further metrics captured about security, performance, culture, the application itself, and the infrastructure. The importance of radiating these, in real time, in a very open and transparent manner cannot be underestimated, nor can putting the behaviors and practices in place to be reactive and responsive to changes in metrics.

As we conclude *Section 5*, *Deliver It*, we can see just how many practices have allowed us to navigate the Mobius Loop, on top of our foundation of culture and technology:

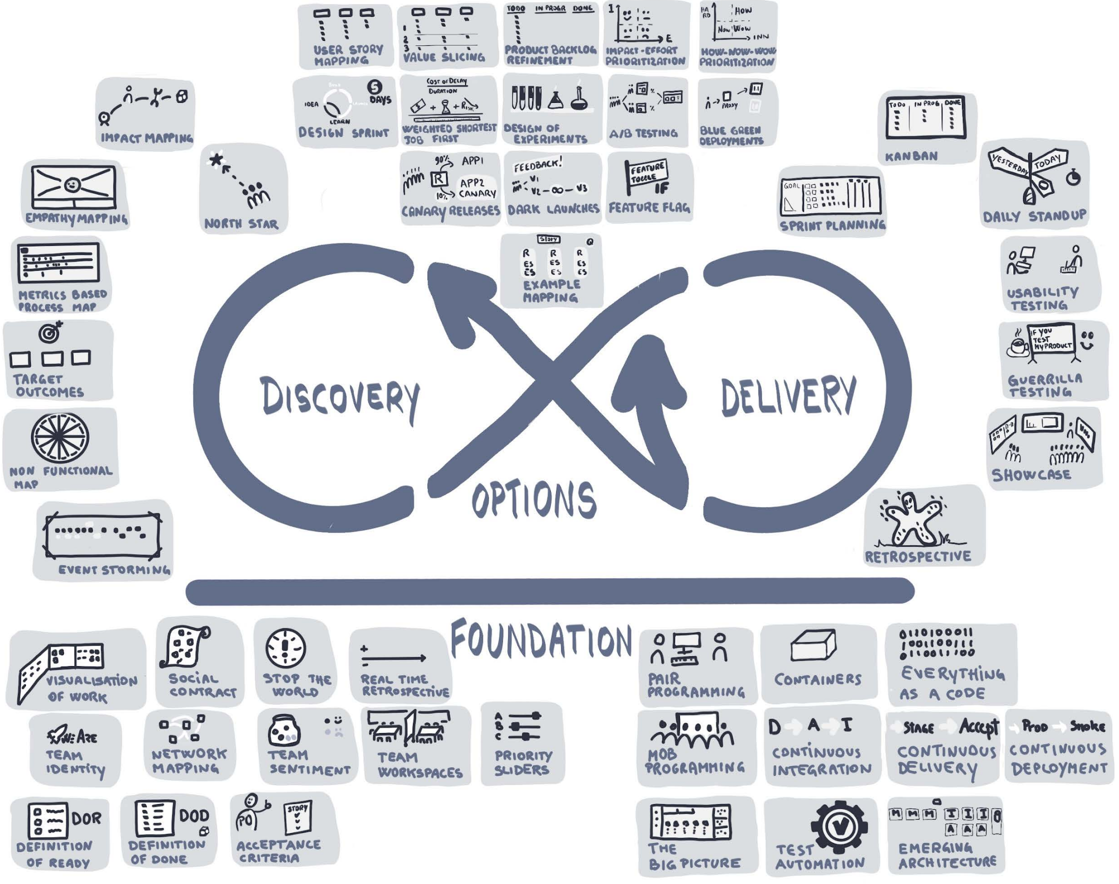


Figure 13.19: The practices mapped onto the Mobius Loop

While we have completed one revolution around the Mobius Loop, we have not completed the journey. We will never complete the journey until the whole product is turned off and decommissioned. This is because the Mobius Loop is infinite and will never end. As we come out of the Delivery Loop, we return to the Options Pivot. We will do this in *Chapter 17*, *Improve It*, when we explore the insights from our trip around the Loop and ask what we have learned, followed by what we are to do next.

Before that, we are going to spend a few chapters diving a bit deeper into the technical solution. We have already started that in this chapter. In *Chapter 14*, *Build It*, we will look at other aspects of how we build the solution. In *Chapter 15*, *Run It*, we'll focus on running the solution. In *Chapter 16*, *Own It*, we'll explore what it means to own the solution. These three chapters form *Section 6* of our book and are all about how product teams *Build It*, *Run It*, *Own It*.

Section 6: Build It, Run It, Own It

In previous sections, we've been discussing the approach we're taking in discovering and prioritizing work to deliver applications such as PetBattle. This includes the many aspects we need to consider when building the solution's components. Now it's time to actually deliver working software:

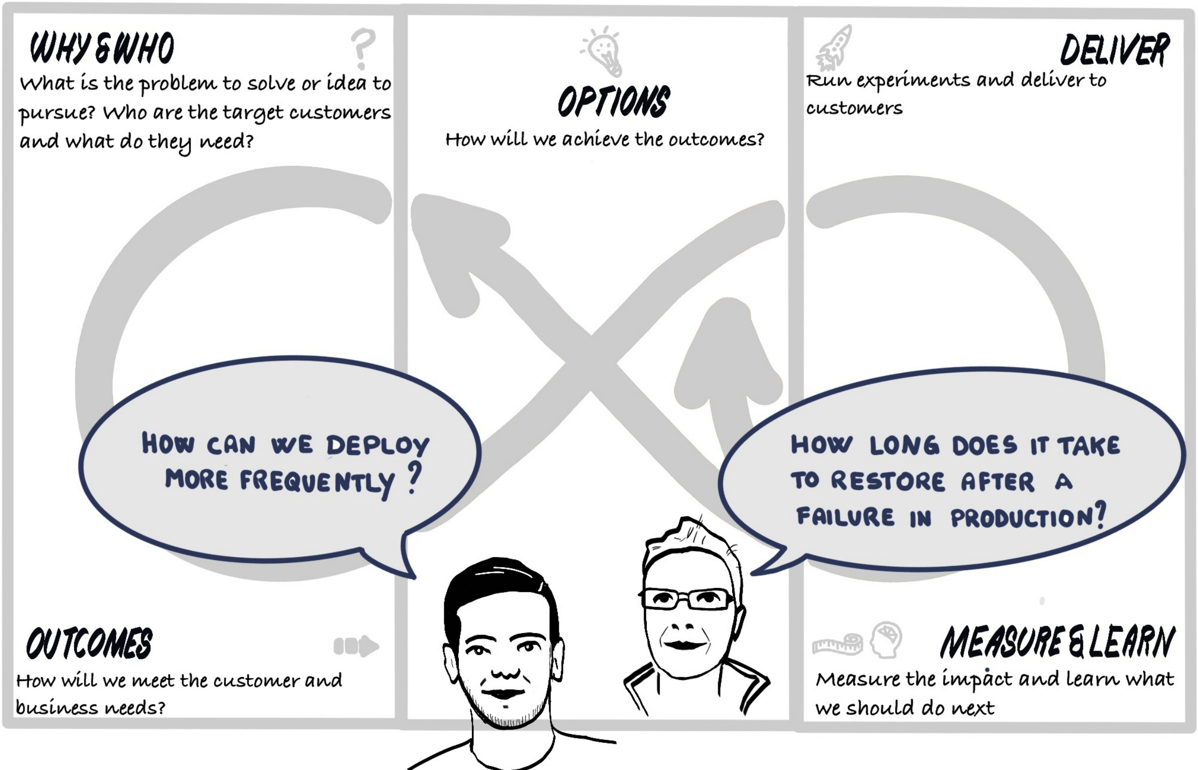


Figure 14.0.1: Focusing on the How

This section is where we do the following:

- Build the solution components (the API, frontend, and so on) using our preferred automation stack.
- Deploy these components onto the OpenShift Container Platform in a repeatable way.
- Own and manage them so that we can watch the site grow and thrive in a culture that empowers our community of motivated developers.

Once our master cat plan is complete, we will become filthy rich in the process! As PetBattle investors, we can then buy islands, invest in space engineering, or just sip martinis by the pool all day until it becomes so boring that we start a new cryptocurrency named *Pet-Coin* and lose it all.

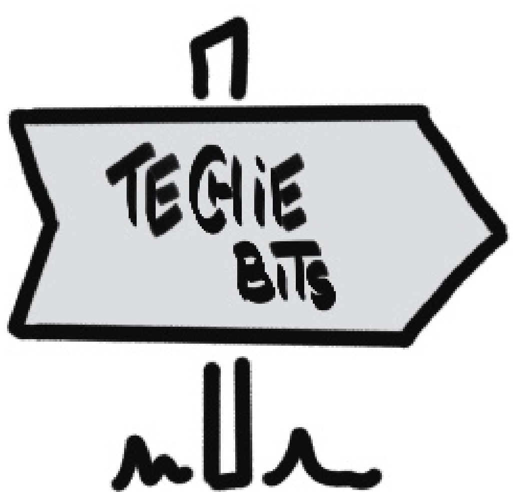
To set expectations correctly, we're not really going into detail about all of the application code itself. All the source code is available (in the Git repositories for the book) and you can go through it at your own pace. There is a lot of value going through the code, in particular the Helm, Quarkus, and AI code.

Our focus is on how to use the tooling provided within OpenShift to build, deploy, and run the applications. We will go into certain advanced features, such as Operators, Serverless, Service Mesh, and CI/CD tooling examples and techniques. Our intention is to provide examples and advice around why we choose to use certain tools or techniques and approaches and how all of them fit together in a cohesive approach specific to PetBattle. You can then pick and choose which ones you want to use in your own projects.

The section is broken down into three parts:

1. *Chapter 14*, *Build It*: This is where we introduce how we will use Git as the single source of truth. We will cover taking our source code and packaging it using either Tekton or Jenkins.
2. *Chapter 15*, *Run It*: This section covers testing, introducing a new component to our app with Knative, running A/B tests, and capturing user metrics using some of the advanced deployment capabilities within OpenShift.
3. *Chapter 16*, *Own It*: This section covers keeping the lights on with monitoring and alerting feedback loops. We will also touch upon the importance of Operators in Kubernetes.

Throughout these chapters, we cover a core belief that all teams should want to take pride in their code. A team member should want to Build It, Run It, and ultimately Own It.



If you feel like the low-level technical details are a bit too much for you or not something you're super interested in, that's OK! The next three chapters may seem like a change in gear, but we'd recommend against skipping over the chapters completely.

Don't worry too much about the low-level details, code snippets, and screenshots, it's important for everyone to grasp the value of the concepts in this book. That includes the value of any technical improvements the team wants to make. The following chapters should help articulate that.

14

Build It

"*It works on my machine*"—a phrase heard time and time again by developers, testers, and operators as they write, test, and verify their code. *It works on my machine* is a phrase rooted in siloed teams where ownership of the problem moves around like a tennis ball on a court at Wimbledon. The metaphorical wall that exists between teams that have operated in silos, passing work over the wall and not taking full responsibility for the end-to-end journey is a problem that has been around for decades. We need to break away from this behavior! From now on, it's not, "*It's working on my machine*" but rather, "*How has your code progressed in the build system*?" The build and deployment pipeline our code runs through is a shared responsibility. In order for this to be the case, all team members must contribute to the pipeline and be ready to fix it when it breaks.

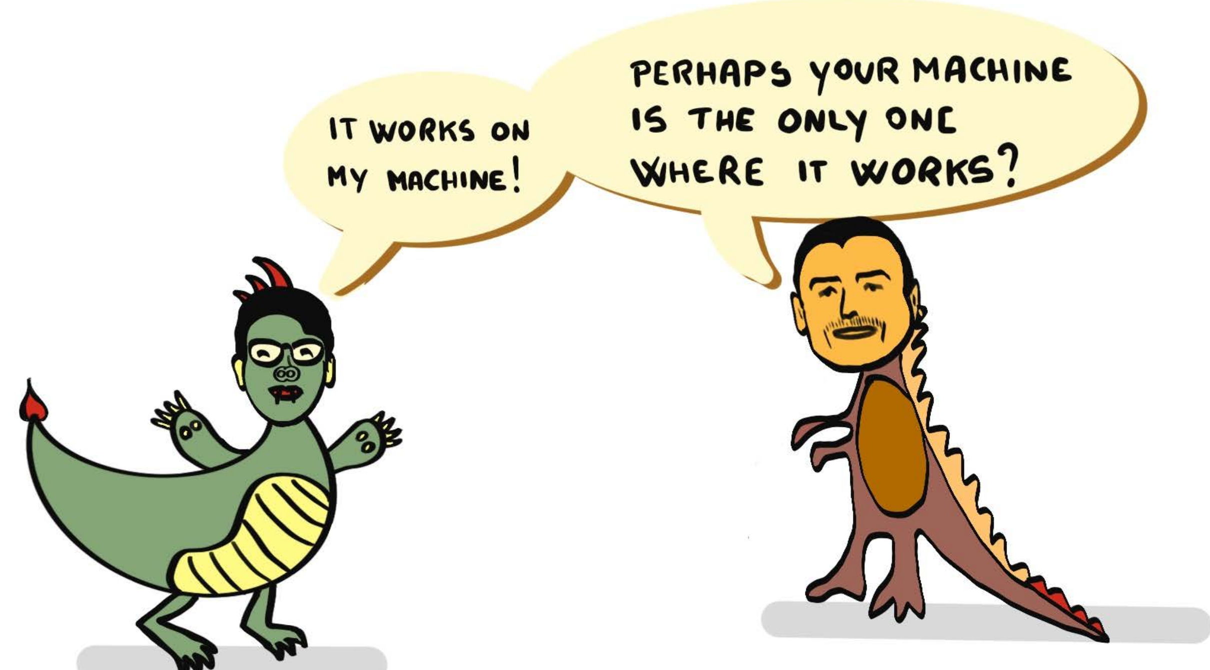


Figure 14.1: It works on my machine

If your code has failed the build because you forgot to check in a dependency, or if your tests are not passing, then it is your responsibility to fix it! The purpose of the deployment pipeline is to create a repeatable process that our code will pass through and accelerate releases while also de-risking them. If we know that on every commit to a repository all of our tests are executed, we're going to have a large amount of confidence that the resulting application will work well. If we're continuously increasing the test volume as the application's complexity increases, that too should grow our confidence. It's critical for teams to want to own their software pipelines.

Having a platform such as OpenShift is a bit like the Beatles singing on the rooftop of the Apple Corps building on Saville Row about people *coming together*. Developers, operations, testers, architects, designers, database administrators, and analysts, everyone coming together and using a platform like OpenShift provides a shared space to collaborate upon. Building applications and business services on the platform where developers can self-service all of their requirements in a safe, access-controlled manner, bringing down the walls between teams, and removing bottlenecks such as having to wait for permissions to deploy an application—this gets everyone speaking the same language to deliver business outcomes through modern application delivery and technological solutions.

Cluster Resources

This section of the book will be one of the most technical. As described in the *Appendix*, the minimum requirements for running the code examples using **CodeReady Containers** (**CRCs**) in this chapter are as follows:

Scenario	CRC Command	Comments
Chapter 14, Build It	`crc start -c 4 -m 16384 -d 50`	Argo CD bootstrap, tool install, application install, CRCs, 4 cores, 16 GB RAM, 50 GB disk

Table 14.1: Minimum requirements for running code examples using CRCs

With the amount of memory required to follow, and the technical content out of the way, let's dive into things in more detail. We'll start by looking over the components of the existing PetBattle applications as they move from a hobby weekend project into a highly available, production-based setup that is built and maintained by a strong, cross-functional team.

Existing PetBattle Architecture

The initial PetBattle architecture was pretty basic and revolved around deploying application components running in a single **virtual machine** (**VM**). The initial architecture had three major components: a JavaScript frontend; a Java-based backend, providing an API and a database; and a single instance of MongoDB. Nothing here was too exciting or complex but hidden inside was a minefield of technical debt and poor implementation that caused all sorts of problems when the site usage took off.

The issues with this architecture seemed to include:

- A monolith—Everything had to be deployed and scaled as one unit, there were no independent moving parts.
- Authentication and access control were non-existent.
- Tests? Unit tests? But seriously, there weren't many.
- It required a lot of data maintenance as everything was stored in the database.
- Bad actors adding inappropriate images to our family-friendly cat application.
- Fragile application—If something went wrong, the application would crash and everything had to be restarted.

Back in *Chapter 9*, *Discovering the How*, we went through an Event Storming exercise that helped drive a newly proposed architecture. It consisted of a UI component and a backing service that provided different REST-based APIs to the UI, as shown in *Figure 14.2*:

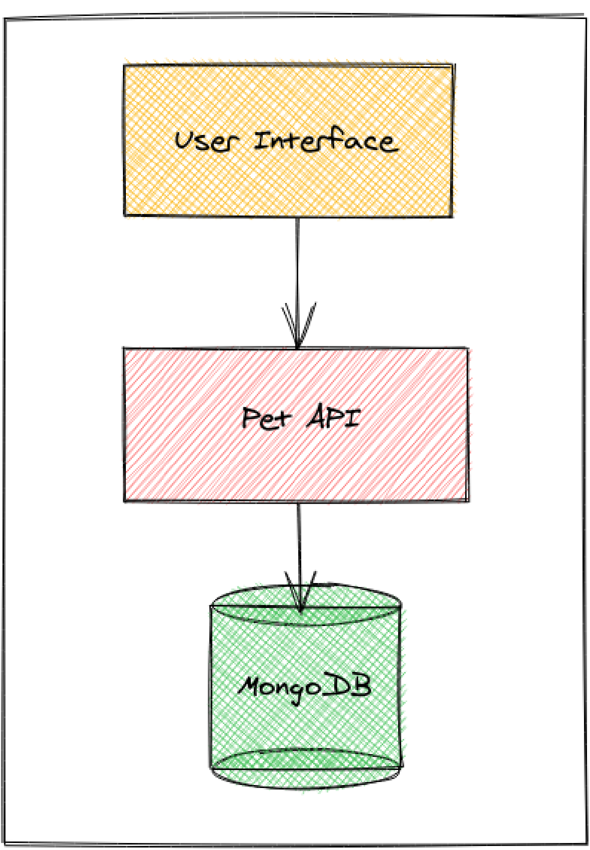


Figure 14.2: PetBattle's initial hobbyist architecture

Let's now take a look at the individual PetBattle components.

PetBattle Components

In the coming chapters, we will explore automation, testing, and the extension of PetBattle to include aspects such as monitoring and alerting, Knative Serving, and Service Mesh. But first, let's imagine that the PetBattle team has completed a few sprints of development. They have been building assets from the Event Storms and now have the components and architecture as seen in *Figure* 14.3. Through the Event Storm, we also identified a need for authentication to manage users. The tool of choice for that aspect was Keycloak.

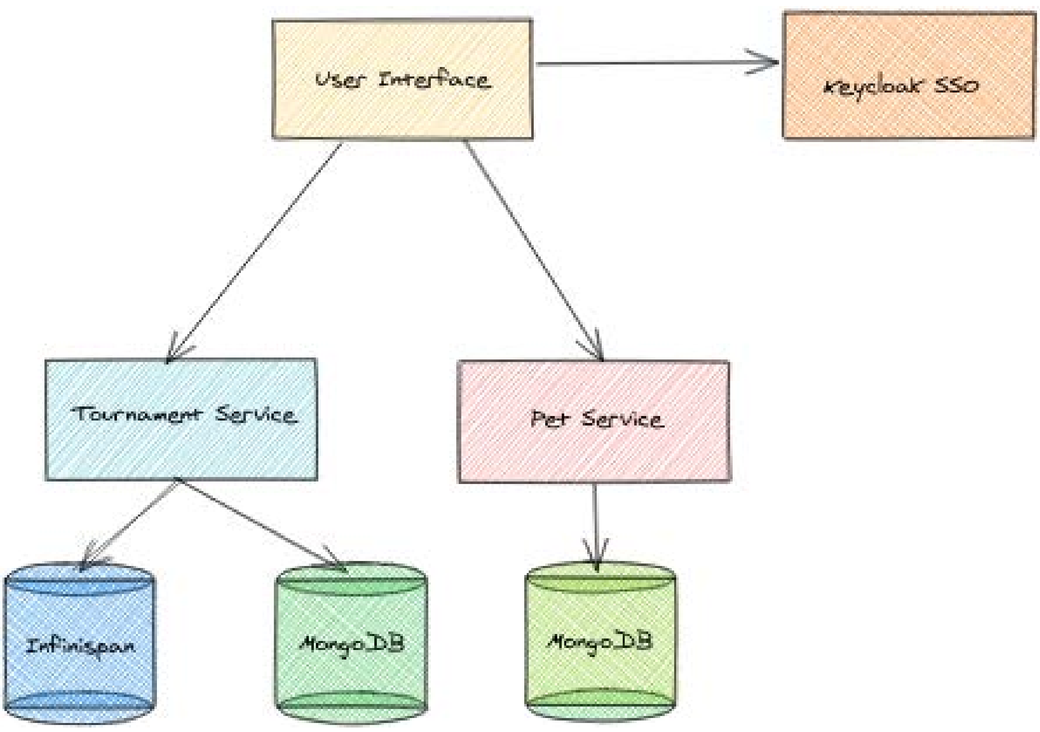


Figure 14.3: PetBattle's evolving architecture

PetBattle's architecture is now increasing in complexity. It has a UI that connects to two services to provide it with data. **Single Sign-On** (**SSO**) and user management are provided by Keycloak. Let's take a look at each component of the architecture in more detail.

User Interface

The UI is written in Angular[1] v12, a complete JavaScript framework from Google for building web and mobile applications. The application is transpiled and the static site code is then served from a container running Nginx (a webserver) instance provided by Red Hat. The application is set up to pull its configuration on startup, which sets up endpoints for all of the dependent services, such as Keycloak and the APIs. This configuration is managed as a ConfigMap in OpenShift.

Pet Service

The Pet service is a straightforward service that uses Java Quarkus[2] as the framework, backed by a MongoDB database to retrieve and store details of the pets uploaded to partake in a tournament.

Tournament Service

The Tournament service also uses the Quarkus framework and stores the state in both MongoDB and an Infinispan distributed cache. MongoDB is used to store the details of the tournament such as which pet won the tournament—but why did we use a cache?

Well, the answer is that a tournament only exists for a finite period of time and using a database to store temporal data is not a great fit for our use case. Also, Infinispan stores the cache data in memory, which is much faster to access than data on disk. The drawback of this is that if the Infinispan pod dies/crashes, then the data is lost. However, we plan to circumvent this in production by having at least two replicas, with the data being replicated between the pods.

User Management

User management, authentication, and access control are a few other critical parts of the architecture that need to be addressed. We're using Keycloak,[3] an open source identity and access management tool, to provide this functionality. We could have written some code ourselves for this functionality, but security is an area that requires a lot of expertise to get it right, and Keycloak does a great job of using open standards to do this job correctly.

1 https://angular.io/

2 https://quarkus.io/

3 https://www.keycloak.org/

Plan of Attack

Initially, we are going to get the core PetBattle application components and services up and running on OpenShift in a fairly manual way. We want to be able to develop locally, adding new functionality to show how easy it is to combine Helm and OpenShift to repeatedly deploy our code. Once that is completed, we are going to automate the setup and deployment process using various tools, including Tekton/Jenkins, Argo CD, and GitOps. We will explore how to add new components to our architecture using Knative and experiment with some of the more advanced deployment capabilities that we can utilize. Finally, in *Chapter 16*, *Own It*, we will look at application monitoring and alerting along with Service Mesh for traceability. *Figure 14.4* shows the additional components added to the architecture, including the Knative Cat Detector Service being proxied via the Service Mesh.

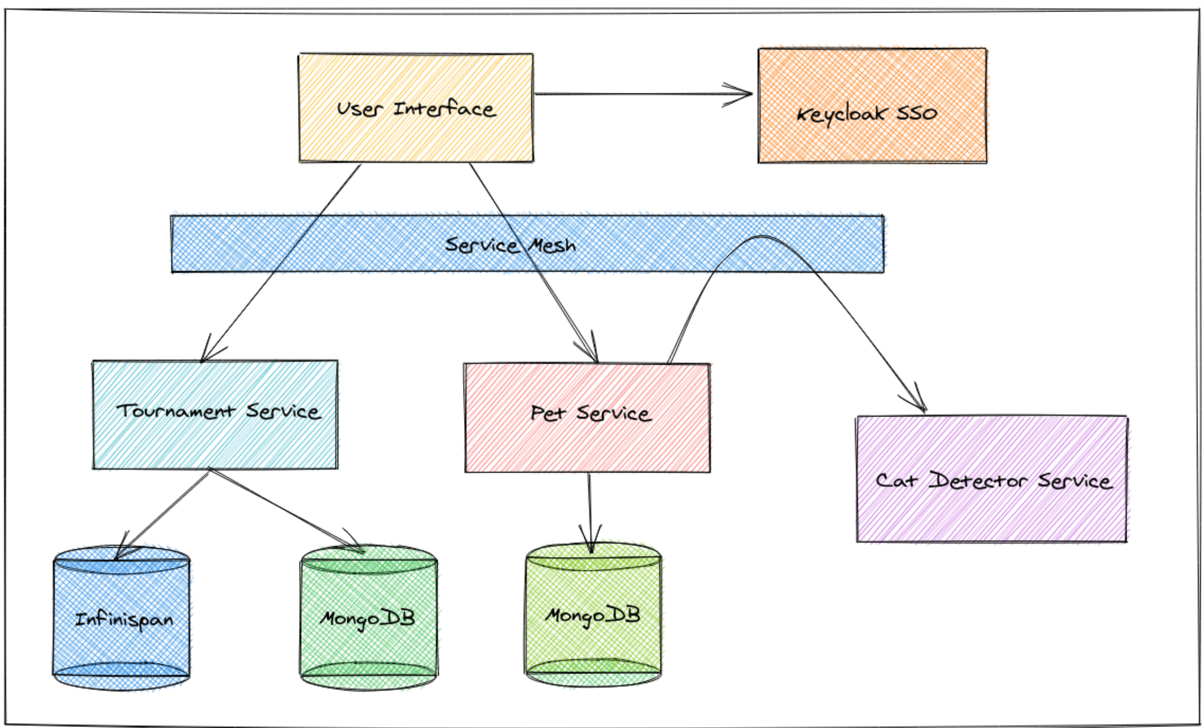


Figure 14.4: PetBattle's target architecture, including final OpenShift deployment

We will be using the command line as much as possible to show and explain the commands involved. Each step can also be performed via the OpenShift web console. If you're new to OpenShift, the web console is a great place to get started as it's full of tips and tutorials!

Running PetBattle

In *Chapter 6*, *Open Technical Practices – Beginnings*, *Starting Right*, we talked about Helm and its use as an application lifecycle manager for installing, upgrading, and rolling back application deployments. We are going to start with the command line, but you can skip to the end of this section if you would like to follow the web console method. If you need help installing the Helm command-line tool, take a look at *Chapter 6* as a refresher. Now let's see how we can easily deploy the PetBattle suite of applications as Helm charts into a single project on OpenShift. On your terminal, add the PetBattle Helm repositories:

```
$ helm repo add petbattle \
    https://petbattle.github.io/helm-charts
```

There are three main applications that make up PetBattle and are searchable in the Helm repository:

Name	Description
`pet-battle`	PetBattle frontend – Angular app deployed on Nginx
`pet-battle-api`	PetBattle API – Cats API that stores our uploaded images in MongoDB
`pet-battle-tournament`	PetBattle Tournament – Service for managing and running each weekly competition

Table 14.2: The three main applications making up PetBattle

The infrastructure Helm chart is normally deployed as a dependency of the Tournament Helm chart but can optionally be deployed alone. This can be useful for debugging purposes. The **Not Safe For Families** (**NSFF**) component is an optional chart, adding a feature whereby the API checks uploaded images for safe content for our family-friendly application.

Name	Description
`pet-battle-infra`	PetBattle infrastructure – Chart contains Keycloak, Grafana, and alerting components
`pet-battle-nsff`	PetBattle NSFF components; includes the machine learning algorithm and APIs

Table 14.3: The infrastructure and NSFF Helm charts

We can search for the latest versions of these charts using the following command:

```
$ helm search repo pet-battle
```

Let's now deploy the main PetBattle application into our OpenShift cluster. We need to update a local copy of the PetBattle frontend's Helm `values.yaml` file to match our cluster URLs. This is needed to connect the frontend when deployed to the correct collection of backend services. We can provide these values to our Helm charts when deploying the suite of PetBattle applications. Let's download an example of the `values.yaml` file for us to edit:

```
$ wget https://raw.githubusercontent.com/petbattle/pet-battle/master/chart/
values.yaml/tmp/values.yaml
```

Open the `values.yaml` file and replace the five URLs listed in the `config_map` to match your OpenShift cluster (change the `apps.cluster.com` domain to `apps-crc.testing`, for example, if you are using a CRC). For example:

```
# custom end point injected by config map
config_map: '{
  "catsUrl": "https://pet-battle-api-petbattle.apps.cluster.com",
  "tournamentsUrl": "https://pet-battle-tournament-petbattle.apps.cluster.
com",
  "matomoUrl": "https://matomo-labs-ci-cd.apps.cluster.com/",
  "keycloak": {
    "url": "https://keycloak-petbattle.apps.cluster.com/auth/",
    "realm": "pbrealm",
    "clientId": "pbclient",
    "redirectUri": "https://pet-battle-petbattle.apps.cluster.com/*",
    "enableLogging": true
   }
}'
```

Gather the **latest chart version** for each of the PetBattle applications from the preceding Helm search command and install the three applications `pet-battle`, `pet-battle-api`, and `pet-battle-tournament` into your cluster. To do this, you will need to be logged in to your OpenShift cluster. For example:

```
# Login to OpenShift
$ oc login -u <username> --server=<server api url>

$ helm upgrade --install pet-battle-api \
petbattle/pet-battle-api --version=1.0.15 \
--namespace petbattle --create-namespace

$ helm upgrade --install pet-battle \
petbattle/pet-battle --version=1.0.6 \
-f /tmp/values.yaml --namespace petbattle

$ helm upgrade --install pet-battle-tournament \
petbattle/pet-battle-tournament --version=1.0.39 \
--set pet-battle-infra.install_cert_util=true \
--timeout=10m \
--namespace petbattle
```

If the `pet-battle-tournament` install times out, just run it again.

Each Helm install chart command should return a message similar to the following:

```
NAME: pet-battle-api
LAST DEPLOYED: Thu Feb 25 19:37:38 2021
NAMESPACE: petbattle
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Using `helm list` should give you a list of the installed charts. You should see the following pods running in your `petbattle` project. An example is shown in *Figure 14.5*:

```
virt:~$ oc get pods
NAME                                   READY   STATUS      RESTARTS   AGE
grafana-deployment-7776b9b67c-wtvx6    1/1     Running     0          4m16s
grafana-operator-9ddb559ff-wp7dw       1/1     Running     0          5m31s
infinispan-0                           1/1     Running     0          4m33s
infinispan-operator-645c945c4-zj4g7    1/1     Running     0          5m13s
keycloak-0                             1/1     Running     0          4m19s
keycloak-operator-5fb789674d-fsxlr     1/1     Running     0          5m11s
keycloak-postgresql-58fdd7fbc7-97q62   1/1     Running     0          4m19s
pet-battle-1-42dkh                     1/1     Running     0          9m47s
pet-battle-1-deploy                    0/1     Completed   0          9m50s
pet-battle-api-6d7894bc97-2gdnr        1/1     Running     0          19m
pet-battle-api-mongodb-1-deploy        0/1     Completed   0          19m
pet-battle-api-mongodb-1-prg4s         1/1     Running     0          19m
pet-battle-tournament-1-7m2zp          1/1     Running     0          5m31s
pet-battle-tournament-1-deploy         0/1     Completed   0          5m35s
pet-battle-tournament-mongodb-1-66gdq  1/1     Running     0          5m35s
pet-battle-tournament-mongodb-1-deploy 0/1     Completed   0          5m38s
```

Figure 14.5: PetBattle pods

The Tournament service will take several minutes to deploy and stabilize. This is because its dependent infrastructure chart is deploying operator subscriptions for Keycloak, Infinispan, and Grafana. Navigate to the OpenShift web console and you should now be able to explore the PetBattle application suite as shown in *Figure* 14.6. Browse to the PetBattle frontend to play with the applications.

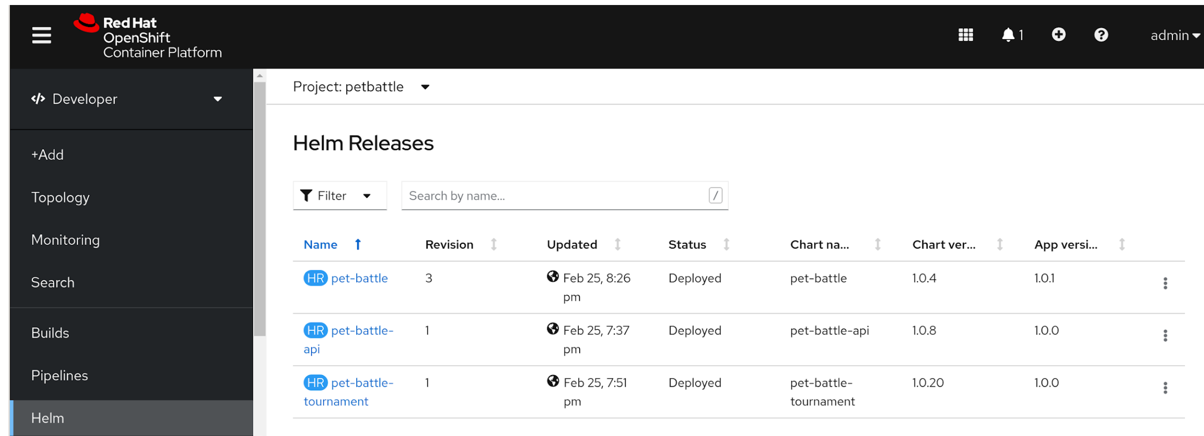


Figure 14.6: PetBattle Helm charts deployed in the OpenShift Developer view

You have now been shown how to install PetBattle Helm charts using the command line—some may say the hard way! We are now going to demonstrate some of the integrated features of Helm in OpenShift—some may say the easier way! We can create a `HelmChartRepository` Custom Resource object that points to our PetBattle Helm chart repository; think of it as `helm repo add` for OpenShift. Run this command to install the chart repository:

```
cat <<EOF | oc apply -f -
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: petbattle-charts
spec:
  name: petbattle
  connectionConfig:
    url: https://petbattle.github.io/helm-charts
EOF
```

With this in place, we can browse to the Developer view in OpenShift and select Add Helm Charts, and see a menu and a form-driven approach to installing our Helm charts—just select a chart and install it:

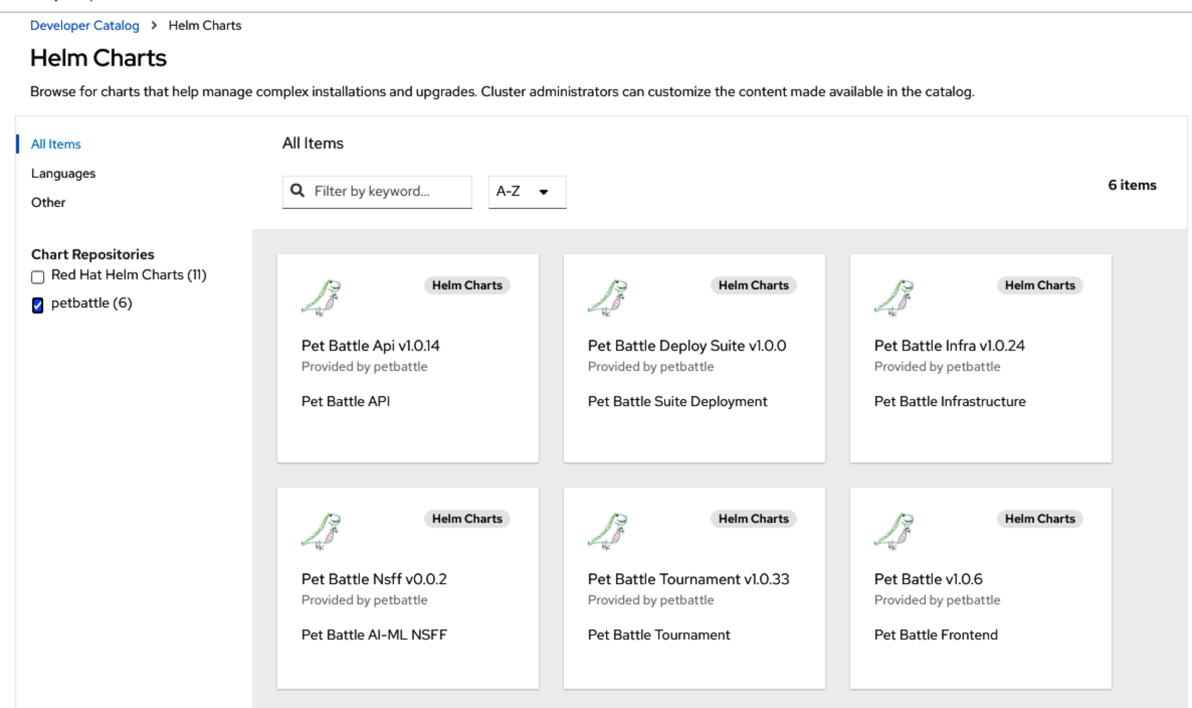


Figure 14.7: Adding PetBattle via the HelmChartRepository view in OpenShift

This can provide a great developer experience for teams sharing services with each other. A backend team can produce a new app to the repository and a downstream team can deploy it to their developer environment in a single click. In fact, if you add a Helm values schema file, OpenShift will build a **What You See Is What You Get** (**WYSIWYG**) form for easy configuration of the values file.

Argo CD

When we established our foundations in *Section 2*, *Establishing the Foundation*, we bootstrapped all of our builds, deployment, and tooling using Helm and Argo CD. We made some opinionated choices when running that bootstrap automation and it's worth discussing some of the trade-offs we made in a bit more detail. We followed our call to action when establishing our technical foundation and planned out what worked for us as the PetBattle product team and reviewed and discussed what was working and not working so well.

It turned out that for our development team, bootstrapping all of the CI/CD tools became an extremely important task. We had been given an arbitrary (but necessary) constraint that our development OpenShift cluster needed to be rebuilt from scratch every two weeks. So we needed to be confident that our CI and CD could stand up quickly and repeatedly. By following our everything-as-code practice, all of our OpenShift infrastructure definitions, CI/CD tooling, and pipeline definitions are stored in Git. The declared CI/CD tooling state is continuously synced to our development cluster by Argo CD, so updating the SonarQube Helm chart version, for example, is as easy as changing one line and pushing it to Git. The change is synchronized and rolled out to our cluster a minute later.

Being able to effectively lifecycle-manage all of the supporting tools involved with building your applications takes effort and attention to detail, but it is worth it in the long run, as you will have built a system that can handle change easily and repeatedly. We have optimized our application lifecycle around the cost of change, making the cost (in man hours) as small as possible. Human time is our biggest resource cost after all!

The versions of all our tooling are checked into Git using **Semantic Versioning**[4] (**SemVer**). There is a good lesson to be learned here in terms of version control; nearly all modern open-source software uses this pattern for it. Often you can be surprised with the resulting deployment when a chart or operator references the *latest* images from external sources. The *latest* tag is a moving target and is updated often. Referencing tags for your versions in Git is like walking a tightrope, there is a balancing act to be performed between wanting to easily accept your toolchain and wanting to update it for bugs and security fixes—hence the use of tags and knowing with confidence that a specific version is working. Normally in SemVer `MAJOR.MINOR` versions are tags that move with small bug fixes and security patches. `MAJOR.MINOR.PATCH` versions are not tags and yet they specify a fixed version (ideally!). Choose a strategy that does not incur too much technical debt that strands the team on old and unsupported versions forever. This is balanced with not having to constantly update version numbers all the time. Of course, if you have optimized for a small cost of change through automation, this problem of changing versions becomes much less of an issue!

We have chosen a *push* (CI) and *pull* (CD) model for our software delivery lifecycle. The job of building images and artifacts (Helm charts and configuration), as well as unit and integration testing, is part of a *push* CI model. On every code commit, a build pipeline trigger (Tekton or Jenkins) fires. It is the job of the Argo CD controller to keep what we have deployed in our OpenShift cluster in sync with the declared application state in our Git repositories. This is a GitOps pull model for CI. The key thing here is that Git is the single source of truth and everything can be recreated from this source.

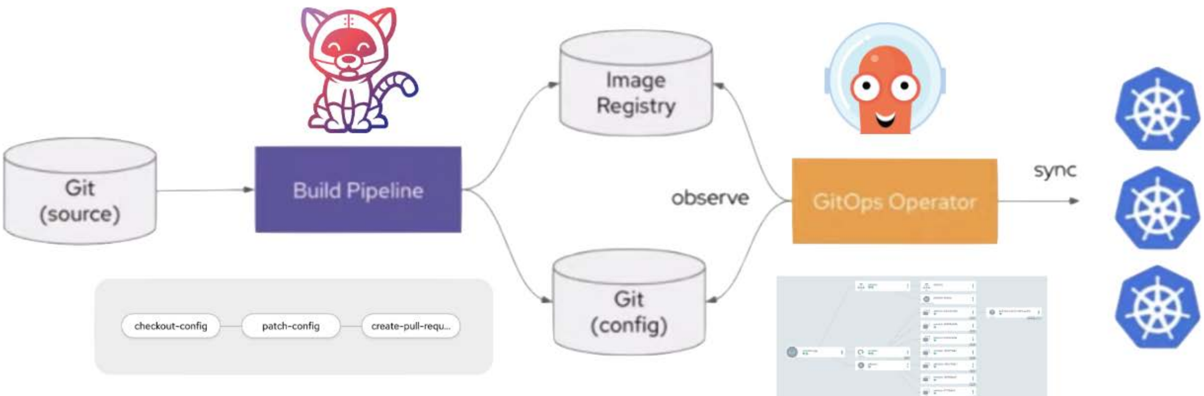


Figure 14.8: A GitOps push and pull model for continuous delivery

4 https://semver.org/

The main benefit we see with this approach is that it is developer-centric. Any change in the code base triggers a pipeline build and deployment. This gives the team fast feedback for any breakages, since automated tests are always run against the new code. The pull CD model decouples the synchronous nature of a build and testing pipeline. Built artifacts (container images and configuration) can be built once, then tagged and promoted through a lifecycle, all of which is controlled from Git. This is great for auditability and discovering who changed what and when. We can easily trace code committed and pushed with builds, tests, and deployments. It is also a flexible approach in that not all artifacts need to be built per se. Configuration can be changed and deployed using the same model. The model is also very flexible in its ability to support different development workflow models. For example, Gitflow and Trunk-based development can easily be catered for, depending on how the team chooses to work.

Trunk-Based Development and Environments

When designing our initial pipelines, we mapped out the basic build, bake, deploy, integration testing, tag, and promotion stages. In *Figure* 14.9, we can see the MultiBranchPipeline Plugin, branches, and namespaces.

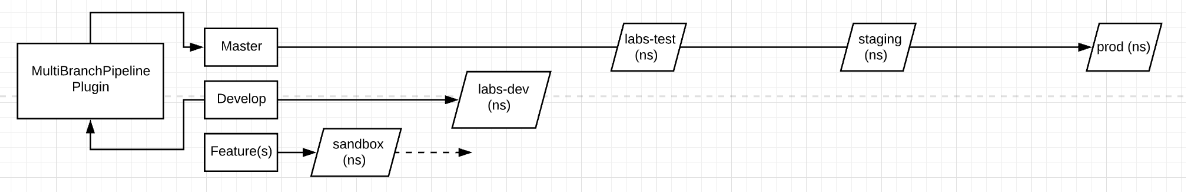


Figure 14.9: Branches and namespaces

This helped us to clarify where the responsibilities lie between our Git branches, our continuous integration tasks, continuous delivery tasks, and which OpenShift projects these would occur in.

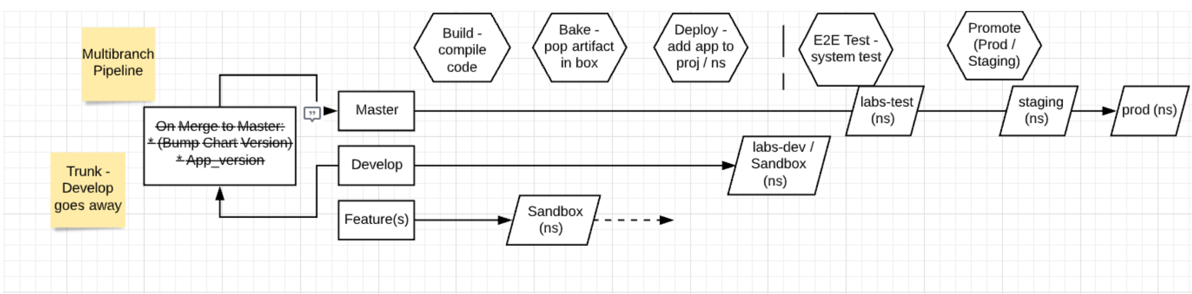


Figure 14.10: Branches and activities modeling

Because we are following trunk-based development,[5] our main/master branch is built, tagged, and promoted through the full lifecycle, that is, images are built, unit and functionally tested, and deployed into `labs-test` with end-to-end testing prior to deployment within the `labs-staging` project. For any short-lived feature branches or pull requests, we decided to only unit test, build, and deploy these sources into our `labs-dev` OpenShift project. That way we can customize which pipeline tasks happen on specific code branches. There is a trade-off between the time and resources used for every code commit in our pipelines, and this must be adjusted according to what goes into our pipelines to help improve the overall product quality.

The Anatomy of the App-of-Apps Pattern

We choose to use Helm; remember, at its most basic, Helm is just templating language for packaging our Kubernetes-based application resources. Each PetBattle application has its own Git repository and Helm chart, making it easier to code independently of other apps. This inner *Helm chart per application* box is depicted in *Figure 14.11*. A developer can get the same experience and end result installing an application chart using a `helm install` as our fully automated pipeline. This is important from a useability perspective. Argo CD has great support for all sorts of packaging formats that suit Kubernetes deployments, Kustomize, Helm, as well as just raw YAML files. Because Helm is a templating language, we can mutate the Helm chart templates and their generated Kubernetes objects with various values.

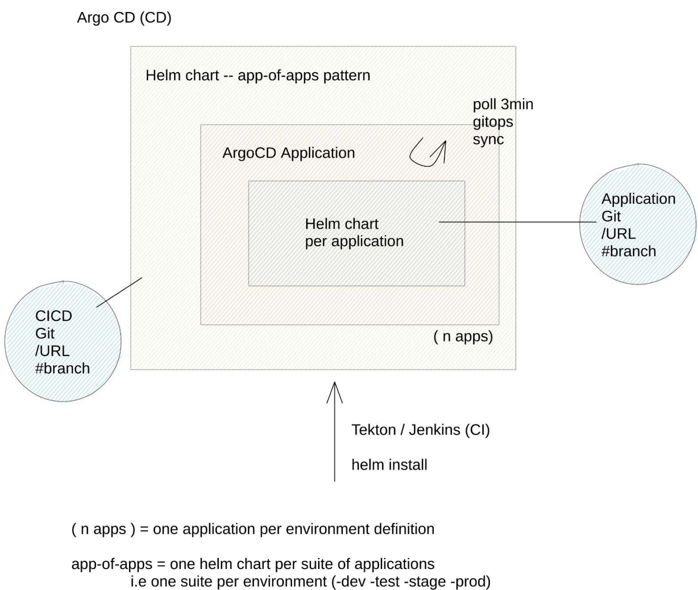


Figure 14.11: Application packaging, Helm, and Argo CD with the app-of-apps pattern

5 https://trunkbaseddevelopment.com

One strict view of GitOps is that mutating a state is not as *pure* as just checking in the filled-in templates with the values themselves. Kustomize, for example, has no templating and follows this approach. We use Kustomize for deploying our CI/CD automation with Argo CD because we think it fits that use case better. This means that we are less likely to have a large number of CI/CD environments for our PetBattle product—at the moment there is just one.

The trade-off here is that while we use GitOps to synchronize the Helm chart itself, the supply of application values may come from multiple places, so you have to be careful to understand where overriding values and precedence occurs, as follows:

- **Lowest precedence**: `values.yaml` provided with the chart (or its sub chart dependencies)—these are kept in sync by the Argo CD controller.
- **Higher precedence**: Override values specified when Argo CD creates the Helm application. These have higher precedence than value files, such as `helm template --set` on the command line. These can be specified in a template or a trigger, depending on how the pipeline is run.

We deploy each of our applications using an Argo CD application definition. We use one Argo CD application definition for every environment in which we wish to deploy the application. This is the red box depicted in *Figure 14.11*. We make use of Argo CD with the app-of-apps pattern[6] to bundle these all up; some might call this an application suite! In PetBattle we generate the app-of-apps definitions using a Helm chart. This is the third, outer green box in *Figure 14.11*. The configuration for this outer box is kept in a separate Git repository to our application.

The app-of-apps pattern is where we declaratively specify one Argo CD app that consists only of other apps. In our case, this is the `pet-battle-suite` application. We have chosen to put all of our applications that are built from the main/master under this `pet-battle-suite` umbrella. We have a PetBattle suite for *testing* and *stage* environments. *Figure 14.12* shows the app-of-apps for the stage environment:

6 https://argoproj.github.io/argo-cd/operator-manual/cluster-bootstrapping/#app-of-apps-pattern

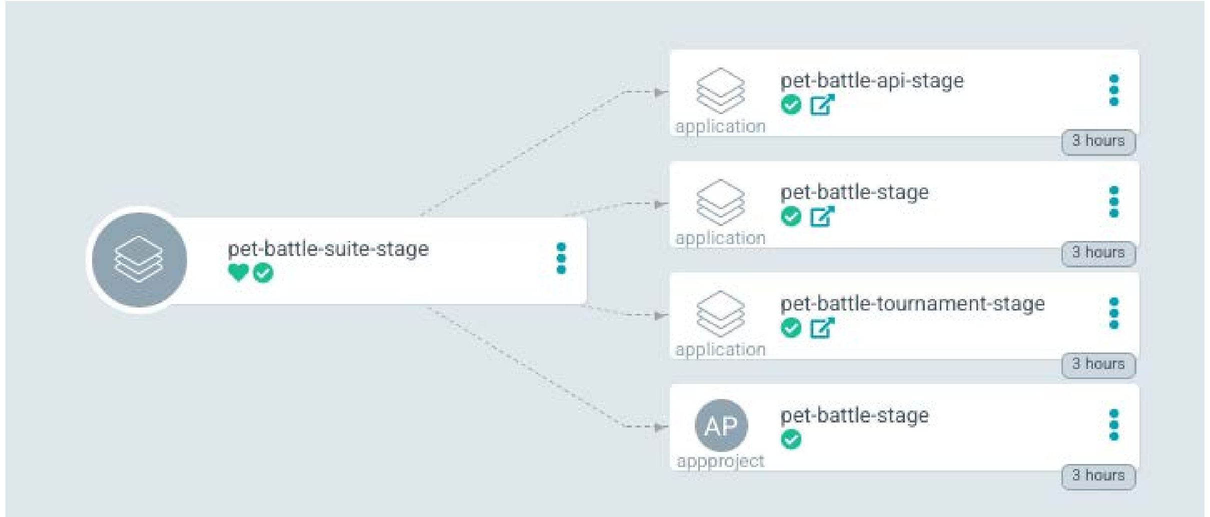


Figure 14.12: Argo CD, a deployed application suite

In Git, we model different branches using the following patterns:

- **HEAD/main/master**: An app-of-apps pattern (full suite of apps deployed and synced to all our environments, `labs-test`, `labs-staging`)
- **Branches/PRs**: A single Helm chart-deployed application in our `labs-dev` namespace only

The Argo CD sync policy for our applications is set to *automated* + *prune*, so that child apps are automatically created, synced, and deleted when the manifest is changed. You can change or disable this if you need to. We also configure a webhook against the CI/CD Git repository so that any changes trigger Argo CD to sync all applications; this avoids having to wait for the three-minute sync cycle when CI/CD code changes.

The Git revision can be set to a specific Git commit **Secure Hash Algorithm** (**SHA**) for each child application. A Git SHA is a unique 40-character code computed for every commit to the repository, and is therefore not movable, unlike a tag. This ensures that even if the child app's repository changes, the app will only change when the parent app changes that revision. Alternatively, you can set it to HEAD/master/main or a branch name to keep in sync with that particular branch. It's a good idea to use Git commit SHAs to manage your application versions the closer you are to production environments. Pinning to an exact version for production ensures easier traceability when things go wrong. The structure here is flexible to suit your product team's needs.

Build It – CI/CD for PetBattle

Let's get our hands dirty now by getting down into the weeds with some more techie stuff. How do we go from code to running in production in a repeatable and safe way? If you remember all the way back in *Section 2, Establishing the Foundation*, we learned about Derek the DevOps dinosaur and the obstacles we put him through to test his fearsomeness. We will now do the same thing with our PetBattle apps, beginning with the frontend.

The Big Picture

Before we touch a line of code, we always like to keep an eye on the Big Picture. This helps us frame what the tool is and why we are using it, and to scaffold out our pipeline from a very high level. Let's build the details of how we manage the PetBattle source code in this same way. As a subtle reminder, the PetBattle frontend is an Angular application. It was built using Node.js v12 and is deployed to a Red Hat Nginx image.

Continuing our Big Picture from the foundation, let's add the steps we will consider implementing for the PetBattle frontend so we can get it ready for a high frequency of change being pushed through it.

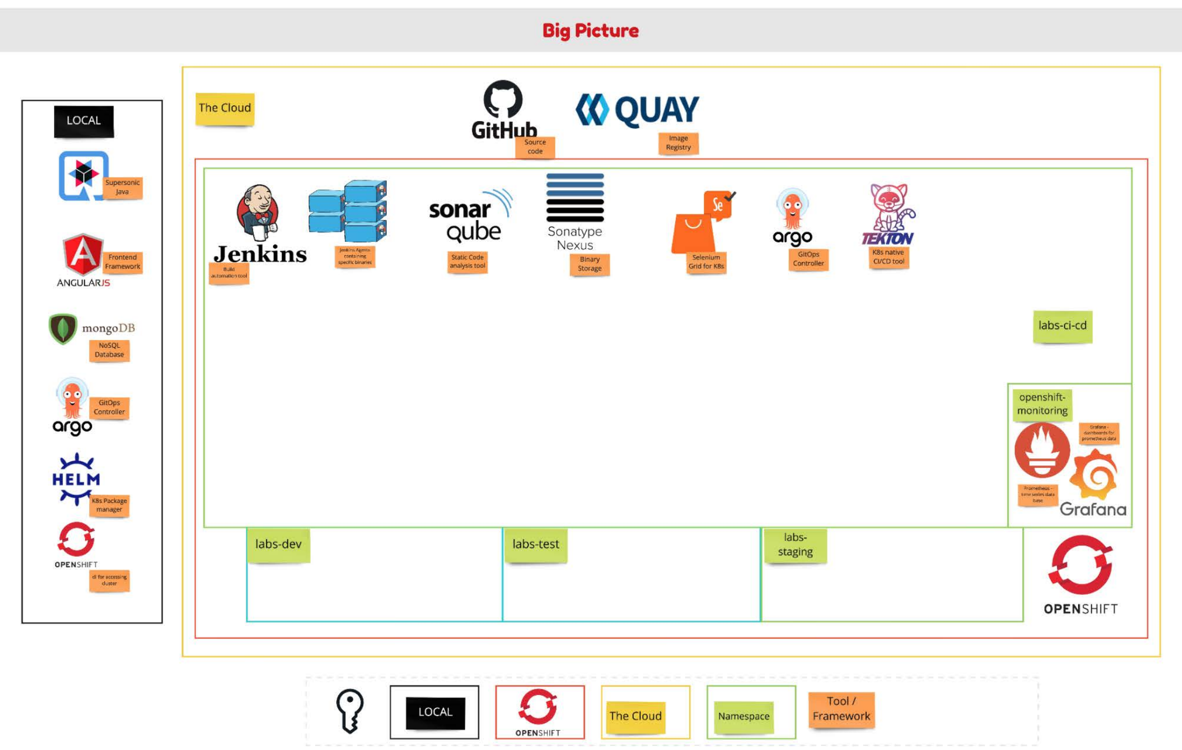


Figure 14.13: The Big Picture including the tools the team thinks they will use

As a quick reminder, our Big Picture from *Section 2*, *Establishing the Foundation*, identified all the tools we might use as depicted in *Figure* 14.13, which include:

- Jenkins: To automate the building and testing of our software
- Nexus: To host our binaries and Helm charts
- Argo CD: To manage our deployments
- SonarQube: To assess our code quality
- Zalenium: For automated browser testing

Now that the tools are in place, let's think about the stages our code should move through for being deployed. A team should start small—what is the minimum amount of automation we need to get code compiling and deployed? It's very important for teams to start small with a basic end-to-end flow of their code; otherwise, things become messy quite quickly, leading to unnecessary complexity and potentially not delivering anything. It's also important because the feedback loop we are creating needs to be fast. We don't want a brilliant, complex process that thinks of everything but takes hours to run! It's not the kind of feedback loop we're trying to create.

We always use three simple stages: Build > Bake > Deploy. A good pattern for engineers to take is to keep to an abstract definition of their pipeline so they can get greater reuse of the pattern across any of their apps, irrespective of the technology they use. Each stage should have a well-defined interface with input and output. Reusing a pipeline definition in this way can lower the context switch when moving between backend and frontend. With this in mind, we can define the stages of our build in the following manner.

The Build

Input: The code base

Output: A "compiled" and unit-tested software artifact

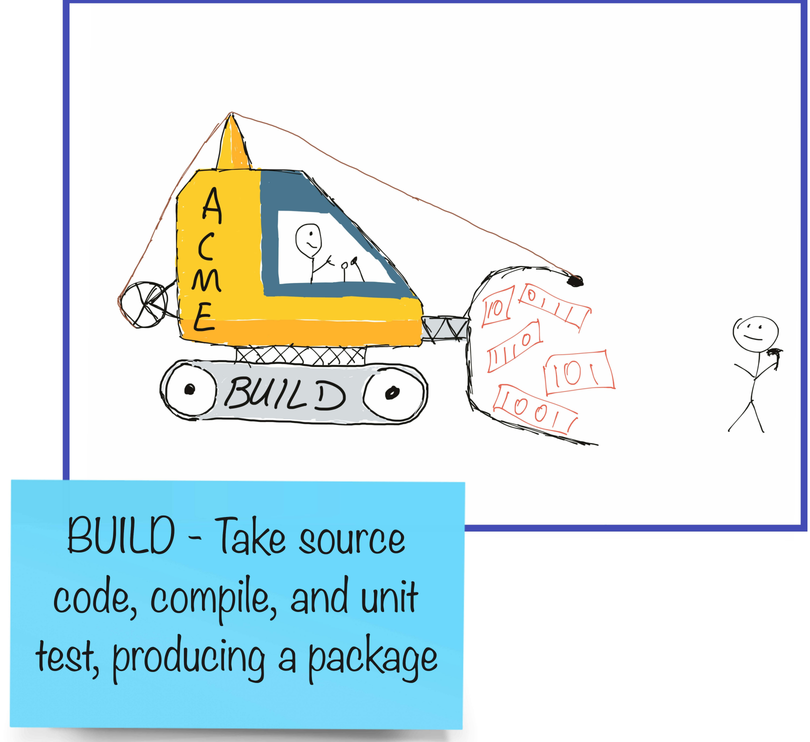


Figure 14.14: The BUILD component from the Big Picture

The build should always take our source code, compile it, and run some unit tests before producing some kind of artifact that will be stored in Nexus. By defining the interface of the build process as lightly as this, it means we can substitute the implementation of what that looks like in each technology or application type. So, for example, when building the PetBattle frontend Angular app, we will use the **Node Package Manager** (**npm**) to complete these steps within the Build stage, but a Java application would likely use Gradle or Maven to achieve the same effect. The hardest work will happen in this stage and it is usually what has the highest dependency on the framework or language that's being used. We will see this in later stages; the technology originally being used becomes less important, so higher reuse of code can occur.

The Bake

Input: A "compiled" software artifact

Output: A tagged container image

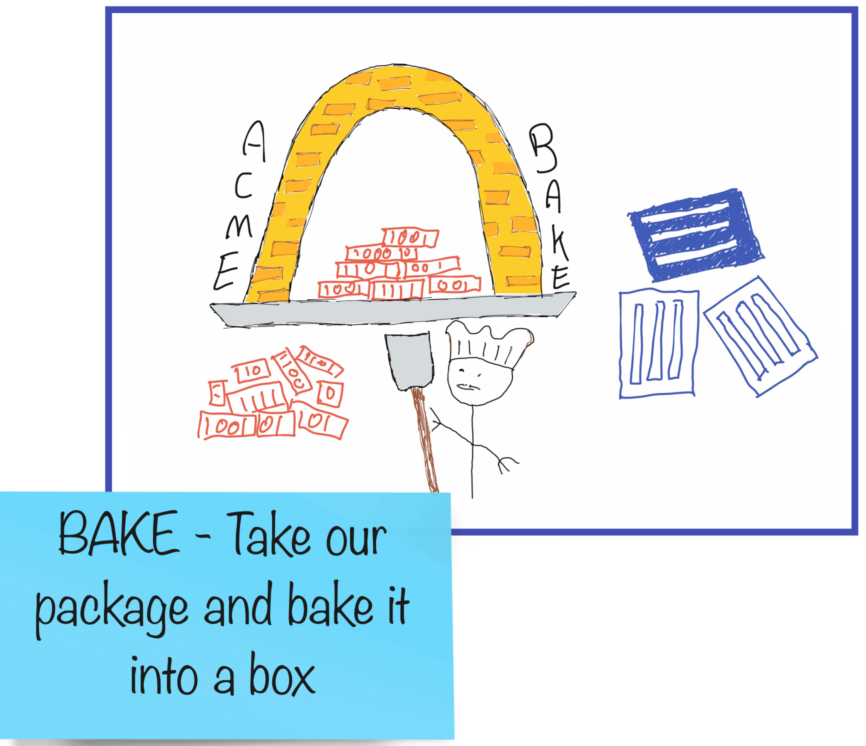


Figure 14.15: The BAKE component from the Big Picture

This is the act of taking our software artifact that was created as an output in the previous step and packaging it into a box, that is, a Linux Container Image. This image is then tagged and stored in a container registry, either one built into OpenShift or an external one. In OpenShift there are many different ways we can achieve this, such as using source-2-image, binary build, or providing a Containerfile/Dockerfile.

The Deploy

Input: A tagged image

Output: A running app in a given environment

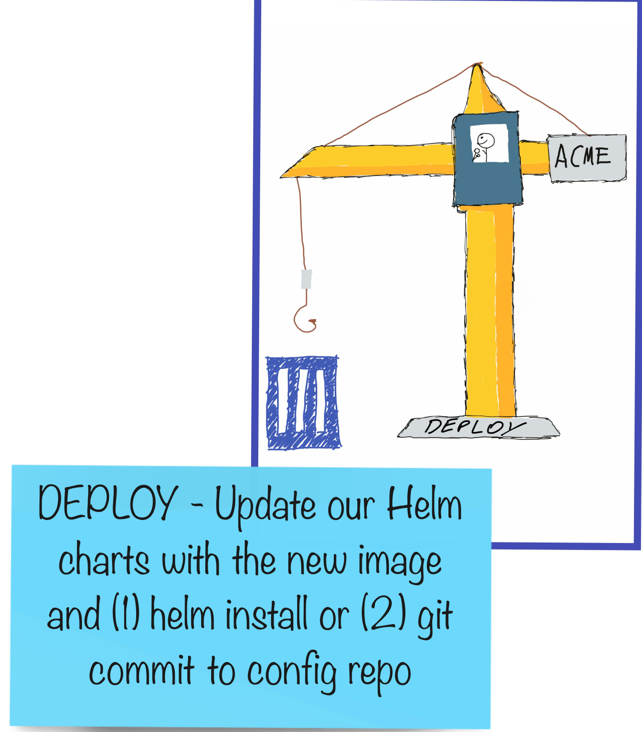


Figure 14.16: The DEPLOY component from the Big Picture

Take the image that has just been pushed to the registry and deploy it along with any other services or configuration required for it to run. Our applications will be packaged as Helm charts, so the deployment will likely have to patch the image referenced in our app's chart. We want our pipeline to support multiple workflows. For feature development, we can just `helm install` into the development namespace. But for release candidates, we should be committing new release information to Git for it to trigger the rollout of changes. The implementation of this workflow is the responsibility of the steps, the lower level of what is being executed. The abstract view of a Deploy should result in a *verified* app deployed on our cluster (and ultimately promoted all the way to production).

The team captures these stages for the applications they're building by adding some nice doodles to their Big Picture. Next, they begin thinking about promoting the application across the environment from test to production. When building applications in containers, we want to ensure the app can run in any environment, so controlling application configuration separately is vital. The team will not want to rebuild the application to target different environments either, so once an image is baked and deployed it needs to be verified before promotion. Let's explore these stages further.

System Test

Input: The app name and version under test

Output: A successful test report and verified app

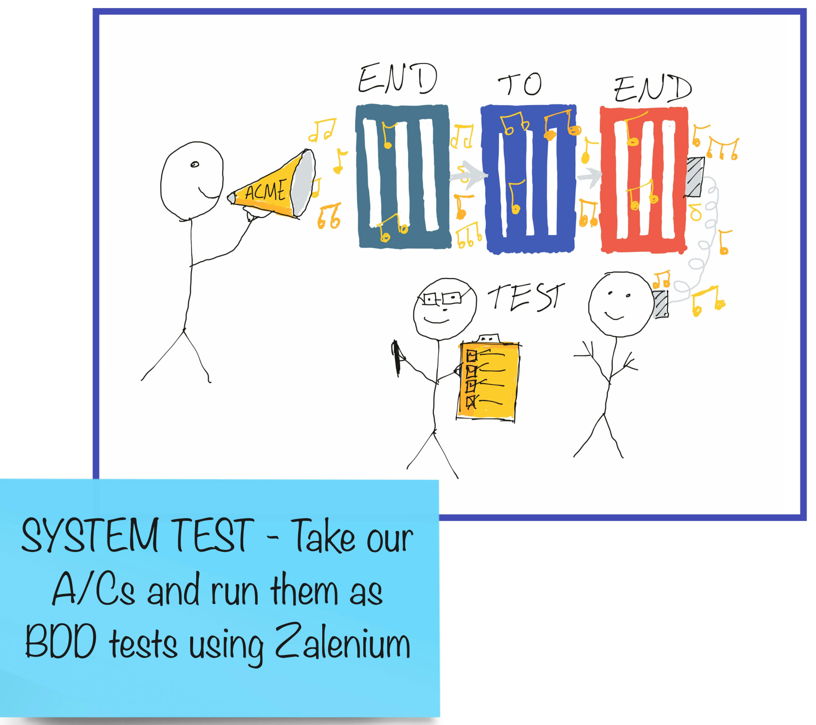


Figure 14.17: The SYSTEM TEST component from the Big Picture

Drive the user behavior within the application via the frontend by verifying whether the app is behaving as expected. If all the connected parts that make up the application are behaving as expected (the microservices, authentication, and frontend) then the app can be signed off and will not need to be rebuilt. Our system test cases for PetBattle will be the acceptance criteria the team has agreed upon. Because of this, we can sign off the application as ready for real-world users. Any component that has changed in the stack should trigger this stage; it is not just the responsibility of the frontend.

Promote

Input: A verified image name and version

Output: Running app in production environment

Figure 14.18: The PROMOTE component from the Big Picture

With the application working as expected (based on our passing system test cases), we can now promote the images that make up our app to the new environment, along with their configuration. Of course, in the world of GitOps, this is not a manual rollout of a new deployment but committing the new version and any custom configuration to our configuration repositories, where they will be picked up by Argo CD and deployed.

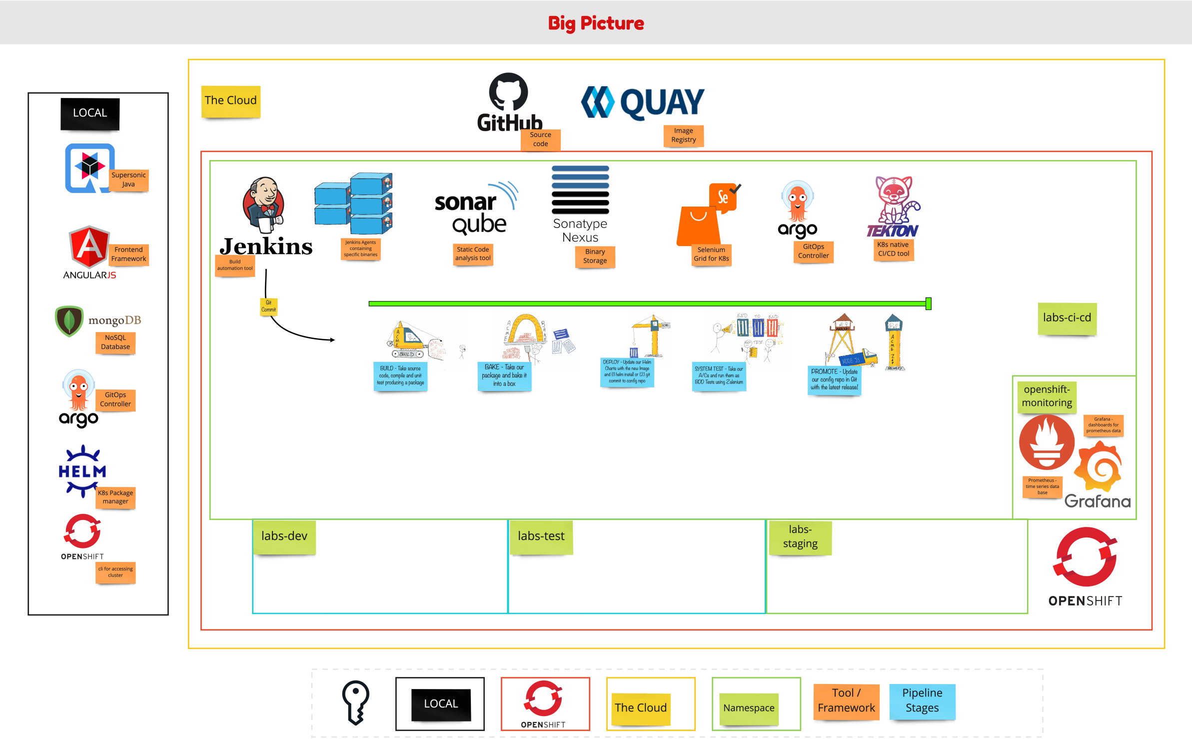


Figure 14.19: The Big Picture including all the stages of the pipeline in place

In *Figure 14.19*, we can see the Big Picture with the stages of the pipeline drawn in. Now that the team knows the stages that their software will pass through on the way across the cluster, they can fill in the lower-level details, the steps. At this stage, the team is looking to see how they can build common pipeline steps, irrespective of the technology they're using. This will provide greater reuse across their software stack but, more importantly, reduce the cognitive load for engineers writing software in multiple technologies. For this, it's a good idea to put on the Big Picture the technology being used. In PetBattle's case, it is Angular and Quarkus (Node.js and Maven for the build tools). They use a new color sticky to write the steps that each service will go through in order to fulfill the interface defined at each stage.

In *Figure 14.20*, we detail what these steps could look like for the Build stage of our pipeline. First, we install the application dependencies. Following this, we test, lint, and compile the code. Finally, we store the successful artifacts in the Nexus repository to use in the next stage, the Bake.

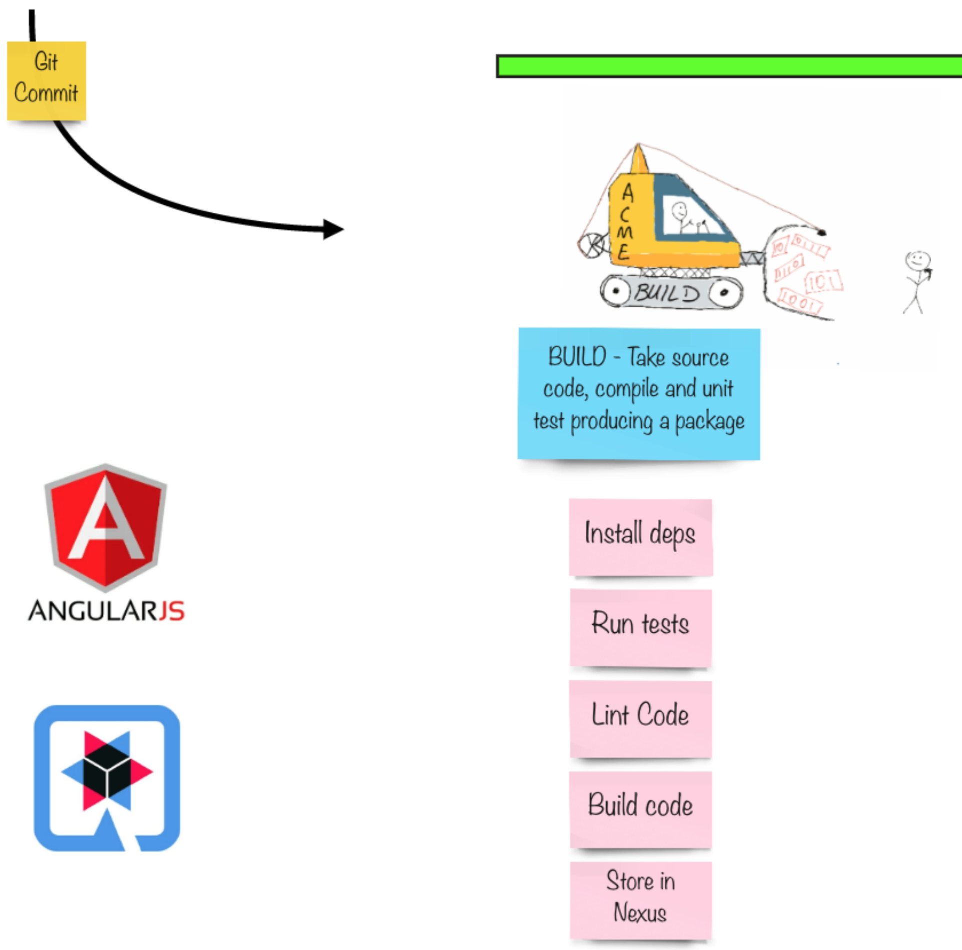


Figure 14.20: The Build stage and the breakdown of its steps

The team continues to flesh out the steps across all the stages. Finally, they add some example containers deployed to each namespace at each stage to give a view of all the components deployed for the PetBattle system to work. This is detailed in *Figure* 14.21:

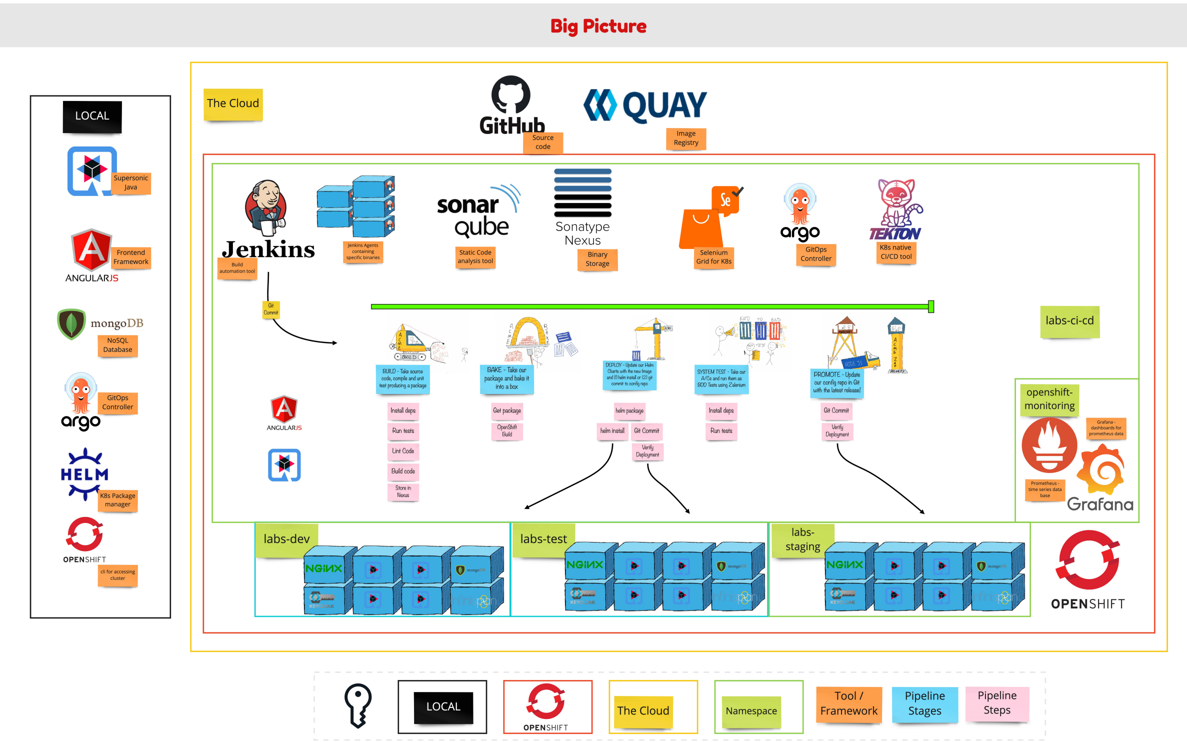


Figure 14.21: The complete Big Picture for our software delivery process

The Big Picture is a helpful practice for getting team alignment on what's in our toolchain and how we use it. It can be a great thing to play back to non-technical team members too, giving them an idea of the complexity and usefulness of being able to repeatedly build and test code. As with all our practices, it's also never done; when a new tool enters our toolchain or we add a new stage in our pipeline, we add it to the Big Picture first. It is the living and breathing documentation of our software delivery process. With the Big Picture complete for now, let's move on to implementing the components it describes.

Choose Your Own Adventure

We understand that there are many ways to do anything and in software development, there are usually hundreds or even more. With this in mind, we hope to meet you, dear reader, where you are now. By this, we mean that the next section will identify two ways of doing the same thing, so take the approach that better fits your own context.

Jenkins is the build tool of choice for lots of companies and developers alike. It has been around for some time and it has its set of quirks for sure. It was never intended to be deployed as a container when it was first conceived. In order to keep things current and have an eye on the future, we've decided to write the code for the Big Picture using both Tekton and Jenkins. Both can easily be tweaked for both frontend and backend development, but for the purposes of this book we will use Jenkins to automate the parts of the Big Picture for our Angular application. For the API, written in Java (Quarkus), we will use Tekton, and go through setting up the same things in a more Kubernetes native way. Both paths are available for the avid reader to play with and get working, but we'll split the narrative this way for illustrative purposes.

So, like you would in a *choose your own adventure* book, you can now pick the path that you would like to follow next. If you're not interested in Jenkins automation, then skip ahead to the Tekton section directly. The code for both options is available in the Git repositories for the book.

Before attempting the pieces in this chapter, make sure to have completed the bootstrap steps in *Chapter 7, Open Technical Practices – The Midpoint*, under the *Implementing GitOps – Let's Build the Big Picture With Some Real Working Code!* section. These steps deploy the CI/CD tooling into your cluster using GitOps. The main tools we are going to use in the next sections include Jenkins, Argo CD, and Tekton.

Jenkins–The Frontend

Jenkins is our trusty friend who will do the hard crunching of code—compiling, testing, and so on—on our behalf. In order to get the best out of all the tools in our kit bag, there are a few items we need to configure first. This includes, among other things, managing secrets and adding webhooks to trigger our Jenkins automation as soon as a developer commits their code.

Connect Argo CD to Git

Let's talk about GitOps. We want our Git repositories to be the single source of truth and the Argo CD controller to analyze the differences between what is currently deployed to our cluster and what is stored in our Git repositories. Argo CD can do things based on the difference it sees between the desired state (in Git) and the actual state (in the cluster) such as automatically synchronizing them or sending a notification to say that these two states are not as expected. For example, in Git we may have set version 123 of our application but the cluster currently has version 122 deployed.

To create this connectivity between our configuration repository and Argo CD, we need to create an Argo CD app-of-apps to point to the repository. The app-of-apps pattern is a neat way to describe all elements of a system. Imagine we have an app, named `App-1`, which is our full system. This `App-1` is made up of independently deployable services such as `App-1a`, `App-1b`, `App-1c`, and so on. For PetBattle, we have the whole system that is all of our frontend, APIs, and other services. We also have one of these for our staging and test environments; this allows us to think of our app-of-apps as a suite of applications.

If we clone the `ubiquitous-journey`[7] project that we set up in *Chapter 7*, *Open Technical Practices – The Midpoint*, to bootstrap our cluster, there is another set of charts in here for our application stacks located in *applications/deployments*. When applied, these definitions will create our Argo CD application Custom Resource pointing to our Helm charts that will be created by the running builds on either Tekton or Jenkins.

The values files (`values-applications-stage.yaml`) contain the Helm chart version and application version that will be updated by Jenkins on successful builds. We want Argo CD to monitor these values when applying changes to the cluster. These values files also contain our overrides to the base Helm chart for specific environments, for example, the config map that the frontend is configured with to communicate with the services it requires to work properly (`tournament-svc`, `cats-svc`, and so on). The following snippet shows the definition of this. These values will differ between development, testing, and staging, so this pattern gives us the ability to version control the configuration we want the application to use on startup.

```
pet_battle_stage:
    name: pet-battle-stage
    enabled: true
    source: *helm_repo
    chart_name: pet-battle
    sync_policy_automated: true
    destination: labs-staging
    source_ref: 1.0.6
    values:
```

7 https://github.com/petbattle/ubiquitous-journey

```
        fullnameOverride: pet-battle
        image_repository: quay.io
        image_name: pet-battle
        image_namespace: petbattle
        config_map: '{ "catsUrl": "https://pet-battle-api-labs-staging.apps.
hivec.sandbox1405.opentlc.com", "tournamentsUrl": "https://pet-battle-
tournament-labs-staging.apps.hivec.sandbox1405.opentlc.com", "matomoUrl":
"https://matomo-labs-ci-cd.apps.hivec.sandbox1405.opentlc.com/", "keycloak":
{ "url": "https://keycloak-labs-staging.apps.hivec.sandbox1405.opentlc.
com/auth/", "realm": "pbrealm", "clientId": "pbclient", "redirectUri":
"https://pet-battle-labs-staging.apps.hivec.sandbox1405.opentlc.com/*",
"enableLogging": true } }'
        image_version: "master"
      project:
        name: pet-battle-stage
        enabled: true
```

So, when we deploy an Argo CD application pointing to this Git repository, it will find additional apps and so create our app-of-apps pattern. The structure of the repository is trimmed, but you can see that the chart is very basic, having just two templates for creating a project in Argo CD and the application definitions to put inside the project.

```
ubiquitous-journey/applications
├── README.md
├── alerting
│   └── ....
├── build
│   ├── ...
└── deployment
    ├── Chart.yaml
    ├── argo-app-of-apps-stage.yaml
    ├── argo-app-of-apps-test.yaml
    ├── templates
    │   ├── _helpers.tpl
    │   ├── argoapplicationdeploy.yaml
    │   └── argocd-project.yaml
    ├── values-applications-stage.yaml
    └── values-applications-test.yaml
```

We could go to the Argo CD UI and connect it to this repository manually, or use the Argo CD CLI to create the Argo CD application Custom Resource, but let's just run this handy one-liner to connect things up for both our staging and test app-of-apps:

```
# from the root of ubiquitous-journey
$ cd applications/deployment

# install an app-of-apps for each test and staging
$ helm upgrade --install pet-battle-suite-stage -f \
argo-app-of-apps-stage.yaml \
--namespace labs-ci-cd .

$ helm upgrade --install pet-battle-suite-test -f \
argo-app-of-apps-test.yaml \
--namespace labs-ci-cd .
```

With these in place, we should see Argo CD create the app-of-apps definitions in the UI, but it will be unable to sync with the child applications. This is because we have not built them yet! Once they are available, Argo CD will kick in and sync them up for us.

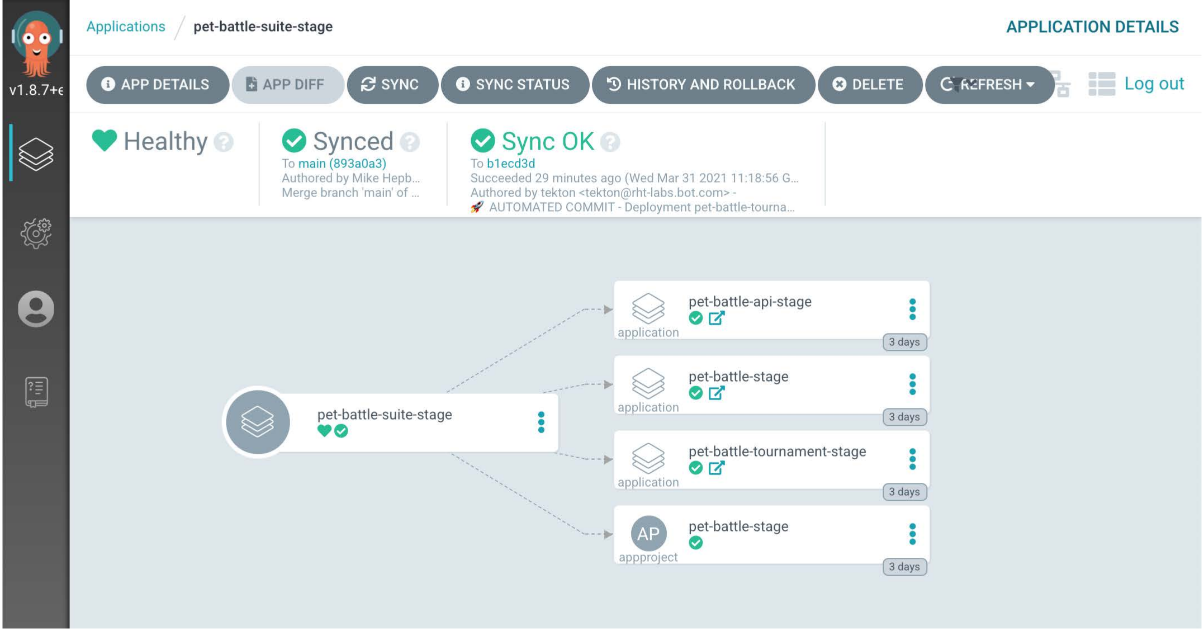


Figure 14.22: Argo CD sync of the pet-battle application suite for the staging environment

To extend this app-of-apps pattern now is very simple. We only need to connect Git to Argo CD this one time. If, after the next few Sprints, the PetBattle team realizes they need to add a new component or service, they can simply extend the values YAML, that is, `values-applications-stage.yaml` or `values-applications-test.yaml` for their staging or test environment, with a reference to the new component chart location and version. For example, for `cool-new-svc`:

```
cool_new_svc_stage:
    name: cool-new-svc-stage
    enabled: true
    source: *helm_repo
    chart_name: cool-new-svc
    sync_policy_automated: true
    destination: labs-staging
    source_ref: 1.0.1 # version of the helm chart
    values:
      fullnameOverride: cool-new-svc
      image_repository: quay.io
      image_name: pet-battle
      image_namespace: petbattle
      image_version: "2.1.3" # version of the application image
    project:
      name: pet-battle-stage
      enabled: true
```

Secrets in Our Pipeline

Jenkins is going to be responsible for compiling our code, pushing the image to a registry, and writing values to Git. This means Jenkins is going to need some secrets! In our case, we're using Quay.io for hosting our images so Jenkins will need the access to be able to push our packaged Container Images to this repository, which requires authentication. If you're following along with forks of the PetBattle repositories and want to create your own running `pet-battle` instance, go ahead and sign up for a free account on https://quay.io/. You can log in with GitHub, Google, or your Red Hat account.

Quay.io

When on Quay, create three new repositories, one for each of the application components that we will be building. You can mark them as public, as private repositories cost money.

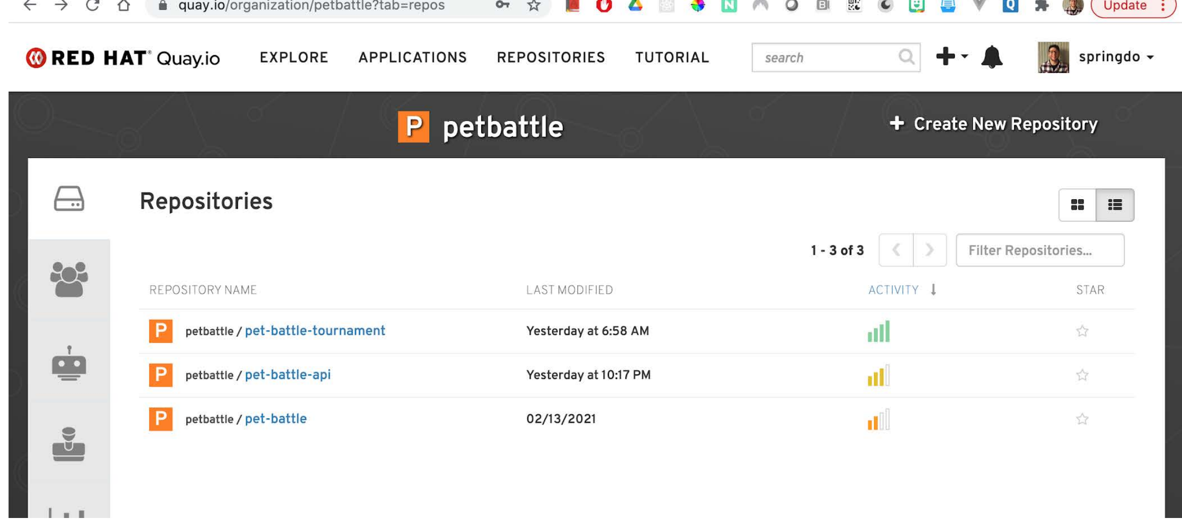


Figure 14.23: PetBattle images in Quay.io

These repositories serve as empty image stores for us to push our images to from within the pipeline. But we need to provide Jenkins with the correct access to be able to push them, so go ahead and hit the robot icon on the UI to create a new service account that Jenkins can use. Give it a sensible name and description for readability.

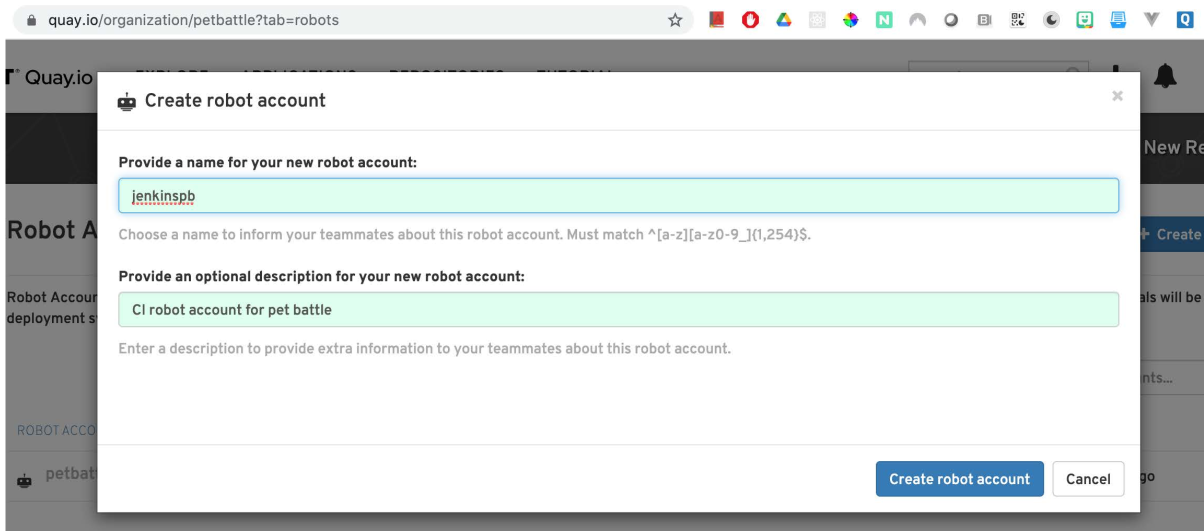


Figure 14.24: Robots in Quay.io

We are going to mark all the repositories we created previously as **Write** by this robot. Hit **Add permissions**:

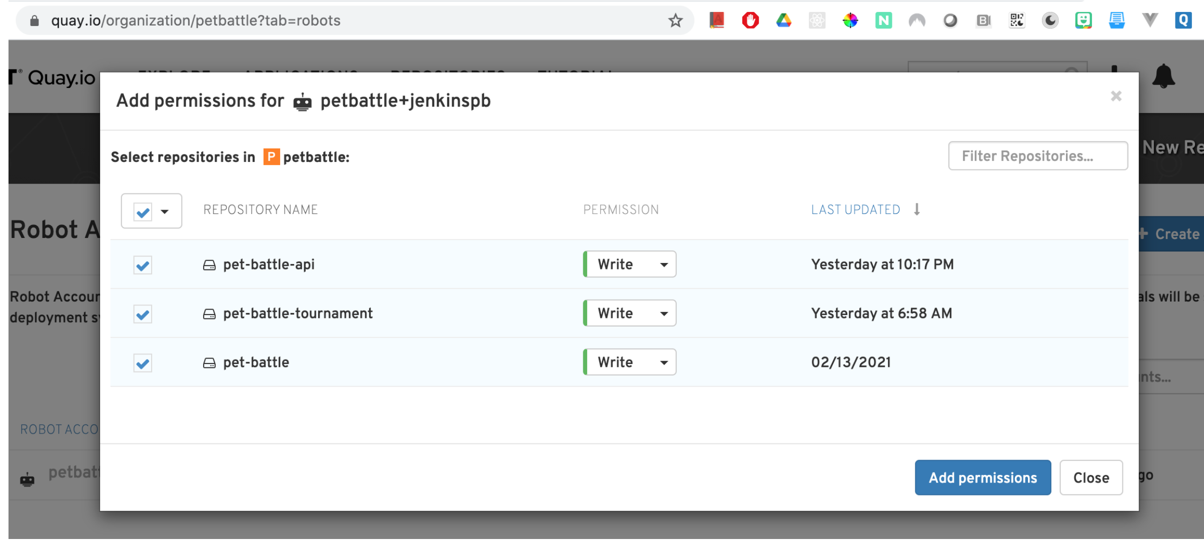


Figure 14.25: Robot RBAC in Quay.io

Now that the repositories and robot account have been created, we can download the secret to be used in our pipelines! Hit the cog on the side of the secret name and select **View Credentials**.

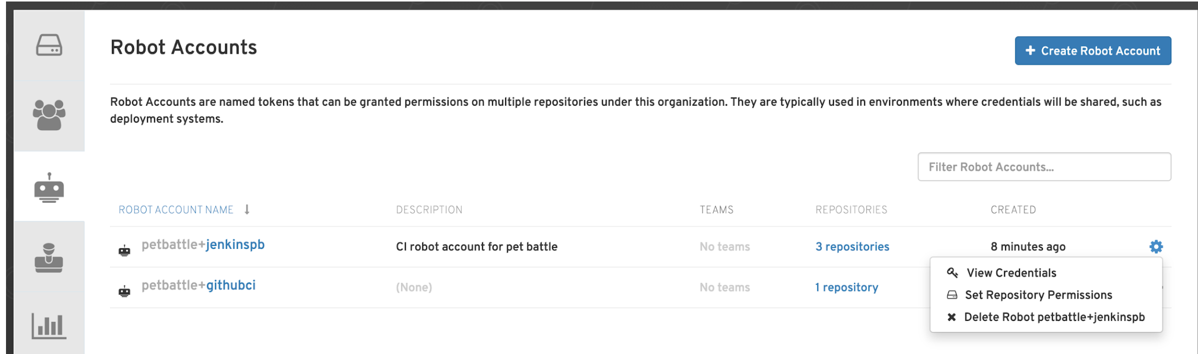


Figure 14.26: How to view the robot credentials in Quay.io

On the page that pops up, download the Kubernetes YAML and store it in your fork of `pet-battle`.

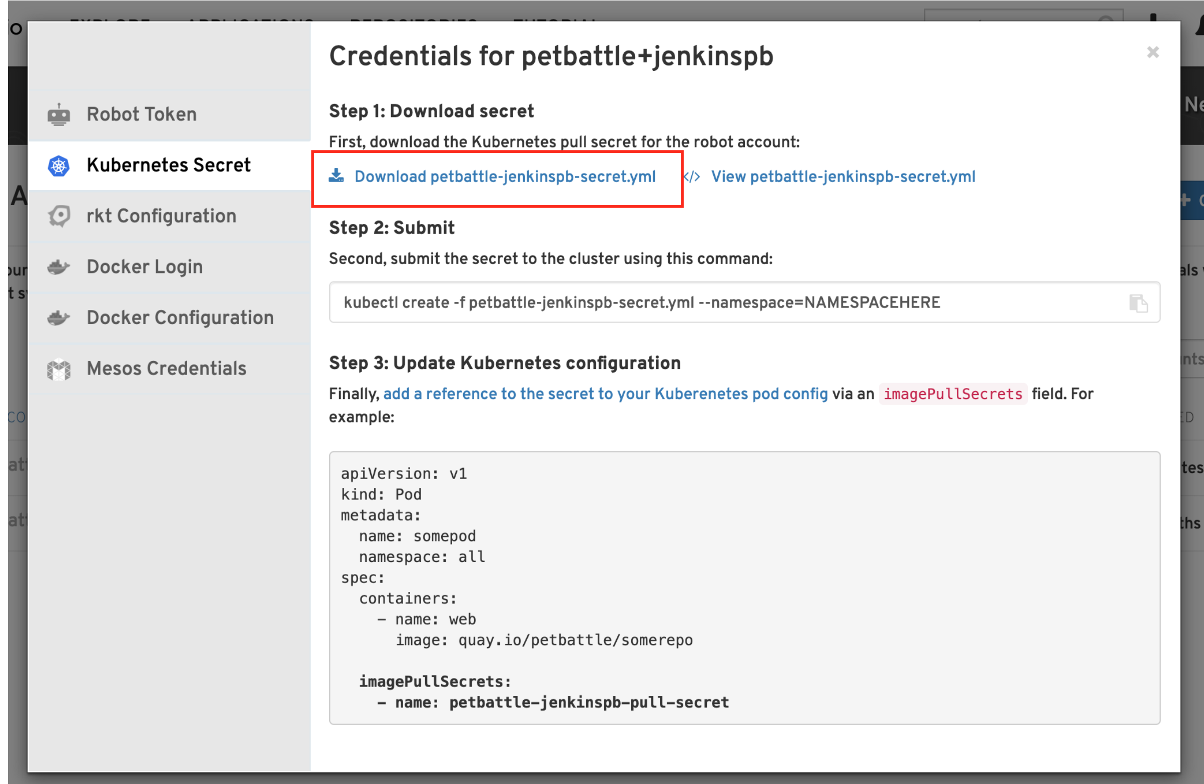


Figure 14.27: Downloading the Kubernetes secret

We can apply it to our cluster (ensure you are logged in first):

```
$ oc apply -n labs-ci-cd -f petbattle-jenkinspb-secret.yml
```

GitHub

A secret is also required for Jenkins to be able to push updates to our Helm values files stored in Git. The values files for our applications will contain the properties we want to pass to our templates, such as ConfigMap variables, or locations of images, such as Quay.io. Our values files for the deployment of our applications will also hold a reference to the image version (that is, the SemVer of our app, such as 1.0.1) to be deployed by patching our DeploymentConfigs. We don't want to manually update this but have a robot (Jenkins) update this when there has been a successful build. Therefore, this secret will be needed to write these changes in versions to our configured repositories, which are being pointed out by Argo CD. We track version changes across all our environments in this way because, after all, if it's not in Git, it's not real.

To create a secret for GitHub, simply go to the Developer Settings view. While logged into GitHub, that's **Settings > Developer Settings > Personal access tokens** or https://github.com/settings/tokens for the lazy. Create a new **Personal Access Token** (**PAT**); this can be used to authenticate and push code to the repository. Give it a sensible name and allow it to have repository access.

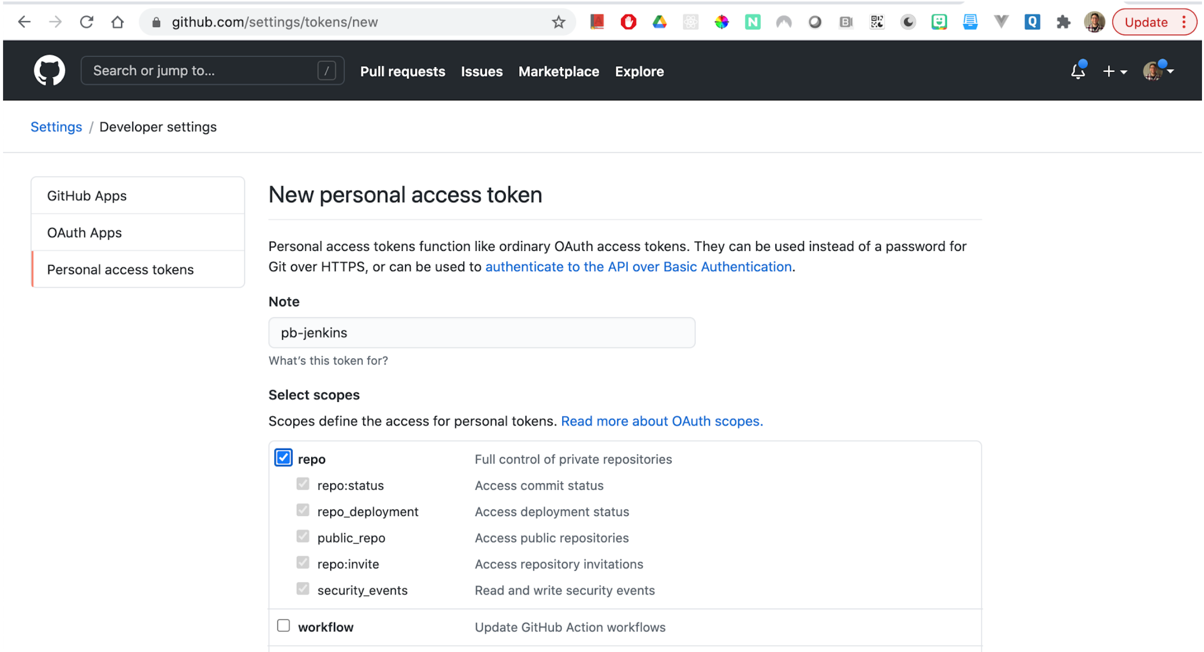


Figure 14.28: GitHub personal access token permissions

Save the token's value, as you won't be able to access it again without generating a new one. With the token in place, we can create a secret in Jenkins by adding it to a `basic-auth` secret. In order for Jenkins, which is running in the same namespace as where this secret will be created, to be able to consume the value of the secret, we can apply a special annotation, `credential.sync.jenkins.openshift.io: "true"`. This little piece of magic will allow any credentials to be updated in Jenkins by only updating the secret!

Update the secret with your values for `GITHUB_TOKEN` and `GITHUB_USERNAME` if you're following along in your own fork and apply them to the cluster:

```
$ cat <<EOF | oc apply -f-
apiVersion: v1
stringData:
  password: GITHUB_TOKEN
  username: GITHUB_USERNAME
kind: Secret
metadata:
  labels:
    credential.sync.jenkins.openshift.io: "true"
  name: git-auth
  namespace: labs-ci-cd
type: kubernetes.io/basic-auth
EOF
```

SealedSecrets

You might be thinking that these secrets should probably be stored somewhere safe—and you're right! If you want to explore the idea of storing the secrets in Git so they too are *GitOpsy* (yes, I did just invent a word there), then we could use SealedSecrets by Bitnami. It provides a controller for encrypting secrets, allowing us to store them as plain text. This means we can commit them to Git! Through the magic of the SealedSecret Custom Resource, it decrypts the SealedSecret, and creates a regular Kubernetes secret on your behalf. We've written our Jenkins Helm chart to accept SealedSecrets for this very reason!

You can deploy SealedSecrets to the cluster by enabling it in the Ubiquitous Journey Git project. Open up `bootstrap/values-bootstrap.yaml`. It's as simple as changing the `enabled` flag to `true` and, of course, Git committing the changes. This will resync with Argo CD and create an instance of Bitnami SealedSecrets in your cluster, by default in the `labs-ci-cd` namespace. Because this is a new component we're adding to our tooling, we should of course also update our Big Picture with the tool and a sentence to describe how we use it.

```
sealed-secrets:
  # Disabled by default
  enabled: true
  nameOverride: sealed-secrets
```

Once the controller has been created, we can seal our secrets by following these few steps:

1. Install kubeseal using the instructions found on their GitHub releases page: https://github.com/bitnami-labs/sealed-secrets/releases.
2. Log in to the cluster where SealedSecrets is deployed and take note of the namespace (in our case this defaults to labs-ci-cd).
3. Process your existing secret using the kubeseal command-line utility. It is important to set the correct namespace otherwise the secret will not be unsealed. In this case, we're going to seal it as super-dooper-secret. It should look something like this:

```
# create secret file from step 3
$ cat << EOF > /tmp/super-dooper.yaml
---
apiVersion: v1
kind: Secret
metadata:
  name: super-dooper
  labels:
    credential.sync.jenkins.openshift.io: "true"
type: "kubernetes.io/basic-auth"
stringData:
  password: "myGitHubToken"
  username:  "donal"
EOF

# encrypt the secret
$ kubeseal < /tmp/super-dooper.yaml > /tmp/sealed-super-dooper.yaml \
  -n labs-ci-cd \
  --controller-namespace labs-ci-cd \
  --controller-name sealed-secrets \
  -o yaml
```

4. You can now apply that secret straight to the cluster for validation, but you *should* add it to the cluster using Argo CD by committing it to Git. If it's not in Git, it's not real. Here, we can see what the SealedSecret looks like before it's applied to the cluster. As you can see, it's a very large, encrypted string for each variable we sealed:

```
# have a look at the sealed secret
$ cat /tmp/sealed-super-dooper.yaml

apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  creationTimestamp: null
  name: super-dooper
  namespace: labs-ci-cd
spec:
  encryptedData:
    password: AgC6NyZa2to2MtKbXYxJBCOfxmnSQ4PJgV8KGdDRawWstj24FIEm5YCyH6n/
BXq9DEPIJL4IshLb2+/kONKHMhKy0CW5iGVadi13GcvO7lxZpVLeVr4T3nc/AqDwPrZ2KdzAI-
62h/7O4o2htRWrYnKqzmUqdESzMxWCK9dl7HZyArGadvwrH98iR48avsaNWJRvhMEDD6EM-
jS5yQ2vJYFMcMzOVyMtbD4f8e3jK7OO+vqoXsHtiuHk4HB63BZZqreiDcFgZMGaD6Bo6FyMSs/
tbkBjttiRvP5zZJ5fqC8IEgbZeuwhJ1eVOeKs/2xGBUMoEiYo6cKaUOqV9k130K2wcdX-
gN8B25phkRK9DpO23LoF/7/uLwNn01pCcxAxm1/2kvX24uPLtirmg1rQ03E9qrnlvyky-
J+9G3QBNtIlsiuoYmEYogZCSRZX29Cm0GWLolYPhlhMDDN6VQI6ktKCH6ubMcbh888Gn2K-
F8NzpQvV5wN9mQVFMR8+wNVkLGsaN+EEdgAc2CmiajIXur3zu4Menq3iWzJcWHdyT-
NlROpJeFH9qyfJLzbkWinPyzyBZEXeiZVKZ/ZAYEvXpyHAUngbnNnUO8HBwsLHb//
uYEzWRufIJezCy9PYxUVSBNIdfPybuCSeb87Bgry/+5D5aUjrqLuKJUhsLWIL3waHyvQswU-
jCQlcgFA7OZ9lwMqkDUYy9SnYatIZ98kf1Z6DA==
    username: AgDY4NgxKug07A+jZ63h0Rdisfm6o7kVaKaiaPek9ZOiHsox1A0P4k-
lYaK/7cTEyOCpFVC/2nx00TX6F2KbA1GsRHkjnU/79nOkYWqsWWTU32c/0Re8sSEIPX7aVgR/
sMXYeWyRediRogA23xFcFzIFSvw4fZ2XpeX0BZNPbMdwZv2b+j/cjW8Po75B5gqbjwhMy-
H36QUApnjmoWmutLONVgAnHVM2rBr1Kx4wgxyy+hdmj+6ZkgMBckd53lMVX0unRVW93I-
j2eDcxTwN+HvVY7nBDmxVHuYAt6t31+DXpqBew10kNDxd8Xw2MpUFDb3JpMwIVtTnt-
mgeoyCHmo7nCYzQkGhwdrEYzoLVQBq+jf0Wmu3YRpEzZbegdTU3QfS1J7XM+86pAF6g-
cgbmrhpguGkU+PwnzPMxGNkq445oEPpvRemftjyFf7A8C+bZ90lrvVzZsfOue8WdXKm-
66vZoYuMPqA2o2HQV0IraaNGYPt9FmiAuXqWhzKsSVsbURXUUOaZIPAyX1z5V1reRz+gs/
cGHYKbmUua7XOFQr32siANI1IkRPi9cT+9iP9GGdq5RzZL75cJGFV8BorZ3CMADGC+skrFKO-
ExFvSrvofBnODB/xnPuirzsnQPcxtdvIz+sCv4M8qG2j0ASH1DBLLF7vMP9rLBgA1sPtzqX-
0CBakjuOjYDqpbXaKqHrM6kdTuBvO7tTDpAYA==
  template:
    metadata:
      creationTimestamp: null
```

```
    labels:
      credential.sync.jenkins.openshift.io: "true"
    name: super-dooper
    namespace: labs-ci-cd
  type: kubernetes.io/basic-auth

# apply it to the cluster
$ cat /tmp/sealed-super-dooper.yaml | oc apply -f- -n labs-ci-cd

sealedsecret.bitnami.com/super-dooper configured
```

5. To *GitOpsify* (yes, again I did just make that up), open up your Jenkins configuration in `ubiquitous-journey/values-tooling.yaml`. Set your values on Jenkins `sealed_secrets` as follows using the output of the secret generation step to add the encrypted information to each key. The example here is trimmed for readability:

```
- name: jenkins
  enabled: true
  source: https://github.com/redhat-cop/helm-charts.git
  ...
  values:
    ...
    sealed_secrets:
      - name: super-dooper
        password: AgAD+uOI5aCI9YKU2NYt2p7as.....
        username: AgCmeFkNTa0tOvXdI+lEjdJmV5u7FVUcn86SFxiUAF6y.....
```

6. If you've already manually applied the secret in *Step 4*, delete it by running `cat /tmp/sealed-super-dooper.yaml | oc delete -f- -n labs-ci-cd`. Then `Git commit` these changes so they are available to Jenkins and, more importantly, stored in Git. In Argo CD, we should see that the SealedSecret generated a regular secret.

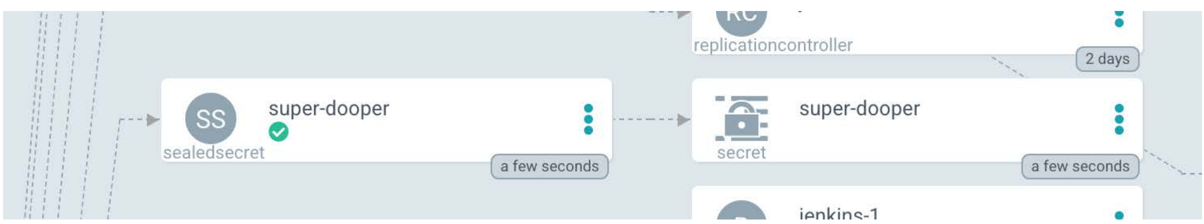


Figure 14.29: SealedSecrets from Argo CD

7. In Jenkins, we should see all that the synchronized secrets using the `magic` annotation (`credential.sync.jenkins.openshift.io: "true"`) have become available.

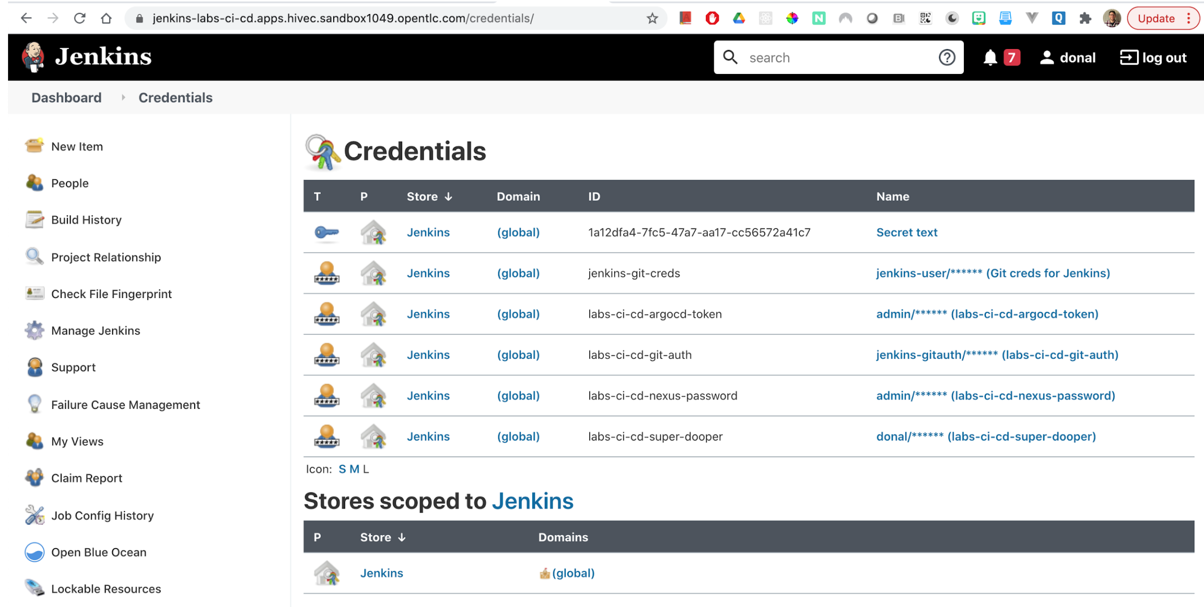


Figure 14.30: Jenkins secrets automatically loaded from Kubernetes

For simplicity here, we will continue without having sealed the secrets; the topic of secrets and GitOps has been included only for illustrative purposes.

The Anatomy of a Jenkinsfile

Some of you may be familiar with a `Jenkinsfile`, but for those who are not, let's take a look at the anatomy of one. A `Jenkinsfile` is just a simple **domain-specific language** (**DSL**) that Jenkins knows how to interpret and build our pipelines from. It's the blueprint for the tasks we want to carry out and the order in which they come. Historically, people manually configured Jenkins by creating jobs in the UI, but as people started to do these things at scale, repeatability and maintenance became an issue. I remember when on my first project, one of the developers' code kept failing the build, so he just edited the job to remove that step. Because we had no authorization or traceability around who could change what on Jenkins, this change was not noticed for some time, until things began to break because of it. Fast-forward to now, and we have the `Jenkinsfile`. It is an *everything-as-code* practice that defines the sequence of things we want our pipeline to execute in order.

The `Jenkinsfile` is made up of a pipeline definition with a collection of blocks, as the following is from our PetBattle frontend. If you're curious as to where this file is, you will find it at the root of the project in Git. *Figure 14.31* is trimmed a little for simplicity:

```
Jenkinsfile
1    pipeline {
2      agent {
3        label "master"
4      }
5  >   environment {…
27     }
28     // The options directive is for configuration that applies to the whole job.
29 >   options {…
34     }
35
36     stages {
37       stage('Perpare Environment') {
38         failFast true
39         parallel {
40 >         stage("Release Build") {…
60           }
61 >         stage("Sandbox Build") {…
86           }
87         }
88       }
89
90       stage("Build (Compile App)") {
91         agent { label "jenkins-agent-npm" }
92         steps {
93           echo '### Running build ###'
94           sh '''
95               npm run build
96           '''
97         }
98           // Post can be used both on individual stages and for the entire build.
99         post {
100 >          always {…
110          }
111        }
112      }
113
114 >    stage("Bake (OpenShift Build)") {…
150      }
151 >    stage("Deploy - Helm Package") {…
182      }      You, 4 days ago • Jenkinsfile for pet battle
183 >    stage("Deploy - App") {…
241      }
242    }
```

Figure 14.31: Anatomy of a Jenkinsfile

The key aspects of a `Jenkinsfile` DSL are:

- `pipeline {}` is how all declarative Jenkins pipelines begin.
- `environment {}` defines the environment variables to be used across all Build stages. Global variables can be defined here.
- `options {}` contains specific job specs you want to run globally across jobs; for example, setting the terminal color or the default timeout.
- `stages {}` encapsulates the collection of blocks our pipeline will go through, that is, the `stage`.
- `stage {}`: All jobs must have at least one stage. This is the logical part of the build that will be executed, such as `bake-image`, and contains the steps, agents, and other stage-specific configurations.
- `agent {}` specifies the node that the build should be run on, for example, `jenkins-agent-npm`.
- `steps {}`: Each stage has one or more steps involved. These could be executing shell commands, scripts, Git checkout, and so on.
- `post {}` is used to specify the post-build actions. Jenkins' declarative pipeline syntax provides very useful callbacks for `success`, `failure`, and `always`, which are useful for controlling the job flow or processing reports after a command is executed.
- `when {}` is used for flow control. It can be used at the stage level, as well as to stop the pipeline from entering that stage; for example, when branch is master, deploy to test environment.
- `parallel {}` is used to execute some blocks simultaneously. By default, Jenkins executes each stage sequentially. If things can be done in parallel, then they should, as it will accelerate the feedback loop for the development team.

For us, we are creating the components in the Big Picture, which are Build > Bake > Deploy.

The Build should always take the source code, compile it, run some linting (static code checking) and testing before producing a package, and store it in Nexus. We should produce test reports and have them interpreted by Jenkins when deciding to fail the build or not. We are building an Angular application, but Jenkins does not know how to execute `npm` or other JavaScript-based commands, so we need to tell it to use the agent that contains the `npm` binary. This is where the agents that we bootstrapped to the Jenkins deployment will come in handy. Each agent that is built extends the base agent image with the binary we need (that is, `npm`) and is pushed to the cluster. This ImageStream is then labeled `role=jenkins-slave` to make it automatically discoverable by Jenkins if they are running in the same namespace. This means that for us to use this, we just need to configure our Jenkins stage to use `agent { label "jenkins-agent-npm" }`.

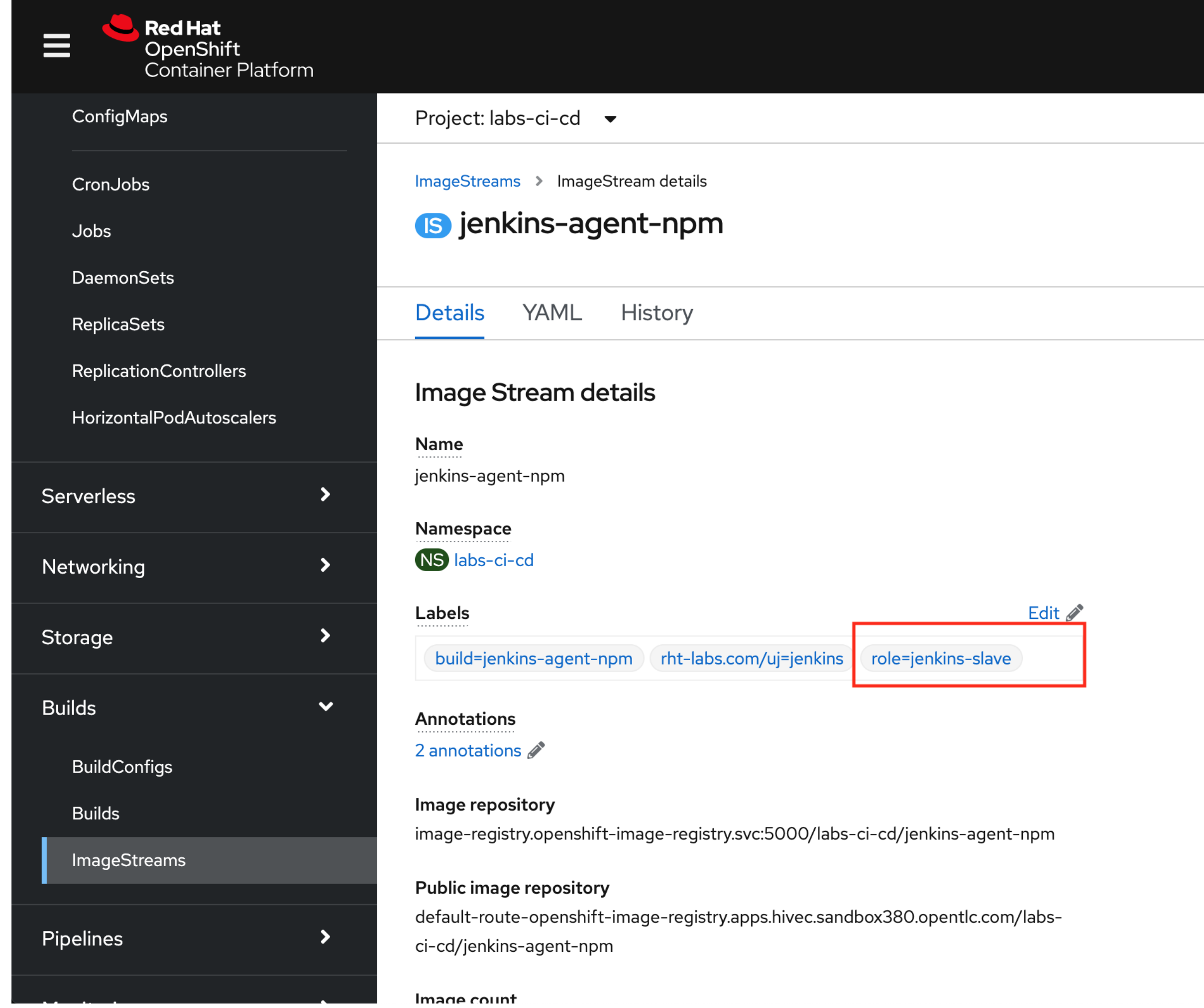


Figure 14.32: Jenkins agent discovery magic

The Build stage will use this agent and execute some steps. The first thing is to capture the app's version to be used throughout the pipeline by reading the app's manifest (`pom.xml` for Java or `package.json` for Node). This version is then used on all generated artifacts, including our image and Helm chart version, and should follow SemVer (for example, `<major>.<minor>.<patch> = 1.0.1`). We will then pull our dependencies, run our tests, lint, and build our code before publishing the results to Jenkins and the package to Nexus.

This will display in the Jenkins declarative pipeline like so:

```
stage("Build (Compile App)") {
  agent { label "jenkins-agent-npm" }
  steps {
    script {
        env.VERSION = sh(returnStdout: true, script: "npm run version
--silent").trim()
        env.PACKAGE = "${APP_NAME}-${VERSION}.tar.gz"
    }
    sh 'printenv'
    echo '### Install deps ###'
    // sh 'npm install'
    sh 'npm ci --registry http://sonatype-nexus-service:${SONATYPE_NEXUS_
SERVICE_SERVICE_PORT}/repository/labs-npm'
    echo '### Running linter ###'
    sh 'npm run lint'
    echo '### Running tests ###'
    sh 'npm run test:ci'
    echo '### Running build ###'
    sh '''
        npm run build
    '''
    echo '### Packaging App for Nexus ###'
    sh '''
        tar -zcvf ${PACKAGE} dist Dockerfile nginx.conf
        curl -v -f -u ${NEXUS_CREDS} --upload-file ${PACKAGE}
http://${SONATYPE_NEXUS_SERVICE_SERVICE_HOST}:${SONATYPE_NEXUS_SERVICE_
SERVICE_PORT}/repository/${NEXUS_REPO_NAME}/${APP_NAME}/${PACKAGE}
    '''
  }
  post {
    always {
      junit 'junit.xml'
      publishHTML target: [
        allowMissing: true,
        alwaysLinkToLastBuild: false,
        keepAll: false,
        reportDir: 'reports/lcov-report',
        reportFiles: 'index.html',
        reportName: 'Code Coverage'
      ]
    }
  }
}
```

Our Bake will always take the output of the previous step, in this case, the package stored in Nexus, and pop it into a container. In our case, we will be running an OpenShift build. This will result in the package being added to the base container and pushed to a repository. If we are executing a sandbox build, say some new feature on a branch, then we are not concerned with pushing the image externally—so we can use the internal registry for OpenShift. If this build is a release candidate then we'll push into Quay.io (our external registry for storing images). The breakdown of the steps for a Bake is found in the Git repository that accompanies this book: https://github.com/petbattle/pet-battle/blob/master/Jenkinsfile.

From a bird's-eye view, the idea is to get the package from Nexus and then create an OpenShift `BuildConfig` with a binary build and pass the package to it. You should then see the build execute in the OpenShift cluster.

```
130
131 >   stage(" Bake (OpenShift Build)") {…
167     }
168
169 >   stage(" Deploy - Helm Package") {…
200     }
201
202     stage(" Deploy - App") {
203       failFast true
204       parallel {
205 >       stage(" Sandbox - Helm Install"){…
221         }
222 >       stage(" TestEnv - ArgoCD Git Commit") {…
258         }
259       }
260     }
261   }
262 }        You, 10 months ago • Headless e2e and Jenkinsfile merge
```

Figure 14.33: Jenkins stages outline for Bake and Deploy

The deployment will take the application that has just been packaged up with its dependencies and deploy it to our cluster. Initially, we will push the application to our `labs-test` environment. We want to package the application and its Kubernetes resources as a Helm chart, so for the deployment we will patch the version of the application referenced in the values file with the latest release. For this reason, our Deploy stage is broken down into two parts.

The first one patches the Helm chart with the new image information, as well as any repository configuration, such as where to find the image we just Baked! This is then stored in Nexus, which can be used as a Helm chart repository.

Secondly, it will install this Helm chart. Depending on what branch we're on, this behavior of how the application will be deployed differs. If we're building on `master` or `main`, it is a release candidate, so there is no more oc applying some configuration—this is GitOps land! Instead, we can commit the latest changes to our Argo CD config repository (Ubiquitous Journey). The commits on this repository should be mostly automated if we're doing this the right way. Managing our apps this way makes rollback easy—all we have to do is Git revert!

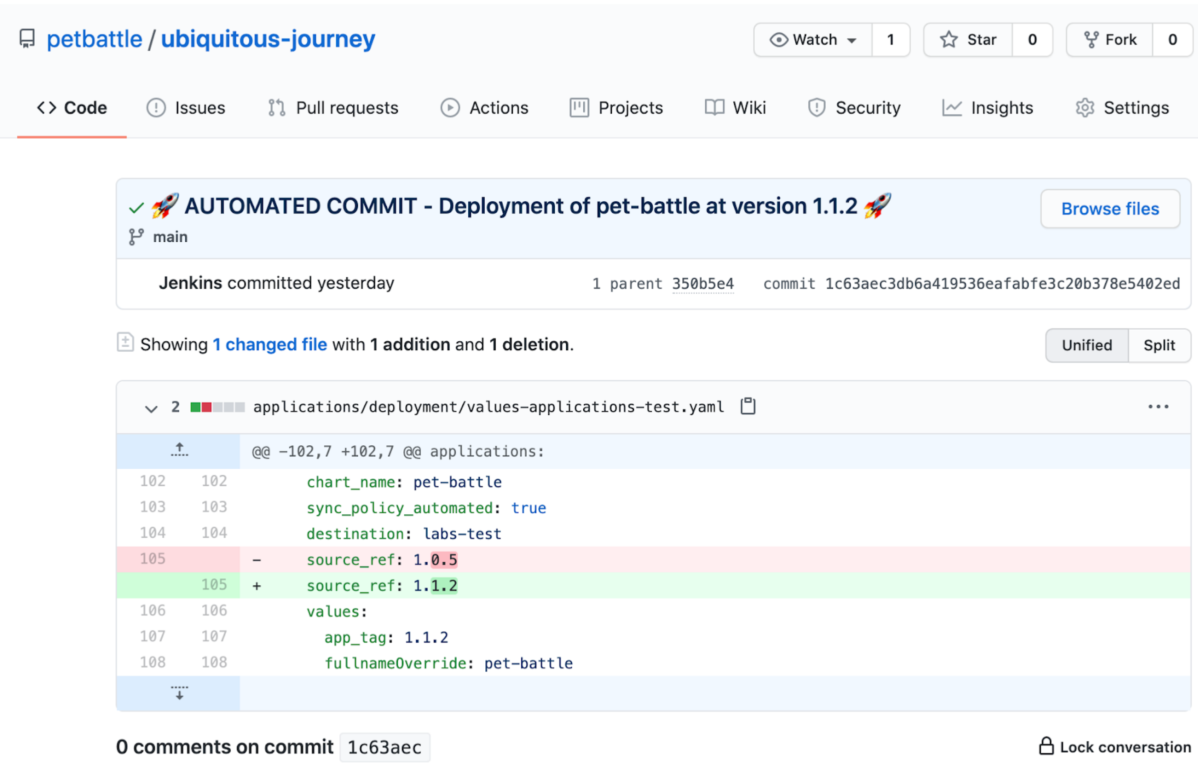


Figure 14.34: Jenkins automated commit of the new version from a pipeline run

Branching

Our pipeline is designed to work on *multibranch*, creating new pipeline instances for every branch that is committed to in Git. It is intended to have slightly different behavior on each branch. In our world, anything that gets merged to `master` or `main` is deemed to be a release candidate. This means that when a developer is ready to merge their code, they would amend the `package.json` version (or `pom.xml` version for Java projects) with the new release they want to try and get all the way through the pipeline to production. We could automate the version management, but because our workflow has always been easier, a developer will do this management, as they are best placed to decide whether it's a patch, a minor, or a major release.

This means that anything not on the `main` or `master` branch is deemed to be a sandbox execution of the pipeline. If something is a sandbox build, it is there to provide fast feedback to the developers of the current state of development in that feature. It can also act as a warning to other engineers that something is not ready to be merged if it's failing. The sandbox builds should be thought of as ephemeral—we're not interested in keeping them hanging around—hence we make some key changes to the pipeline to accommodate this:

1. **Internal registry**: If our built image is pushed to our external repository, it will become clogged up and messy with unnecessary images. Every time a developer commits to any branch it would create new images, so it can introduce a cleanup headache; hence we use the internal registry, which automatically prunes old images for us. We only use the external registry when we know a release could go all the way to production.

2. **Helm install**: For our deployments, we're not interested in bringing in a heavyweight tool like Argo CD to manage the development/sandbox deployments. It's unnecessary, so we just use Jenkins to execute a Helm install instead. This will verify that our app can deploy as we expect. We use Argo CD and GitOps to manage the deployments in test and staging environments, but any lower environments we should also treat as ephemeral (as we should test and staging too).

This approach allows us to support many different types of Git workflow. We can support GitHub Flow, Gitflow, and Trunk, all via the same consistent approach to the pipelines.

Webhooks

Before we actually trigger Jenkins to build things for us, it's important to add a few webhooks to make our development faster. We need two, one for the Argo CD config repo and one for Jenkins in our source code repository.

When we commit a new change to the Git repositories that Argo CD is watching for, it polls. The poll time is configurable, but who can be bothered to wait at all? Argo CD allows you to configure a webhook to tell it to initiate a sync when a change has been made.

This is particularly important if we want things to happen after Argo CD has worked its magic, such as in a system test. Our pipeline in Jenkins runs synchronously, but Argo CD is asynchronous and therefore anything we can do to reduce the wait between these behaviors is critical.

On GitHub, we can configure the webhook for Ubiquitous Journey to trigger Argo CD whenever the repository updates. On GitHub, add the webhook with the address of our Argo CD server followed by `/api/webhook`.

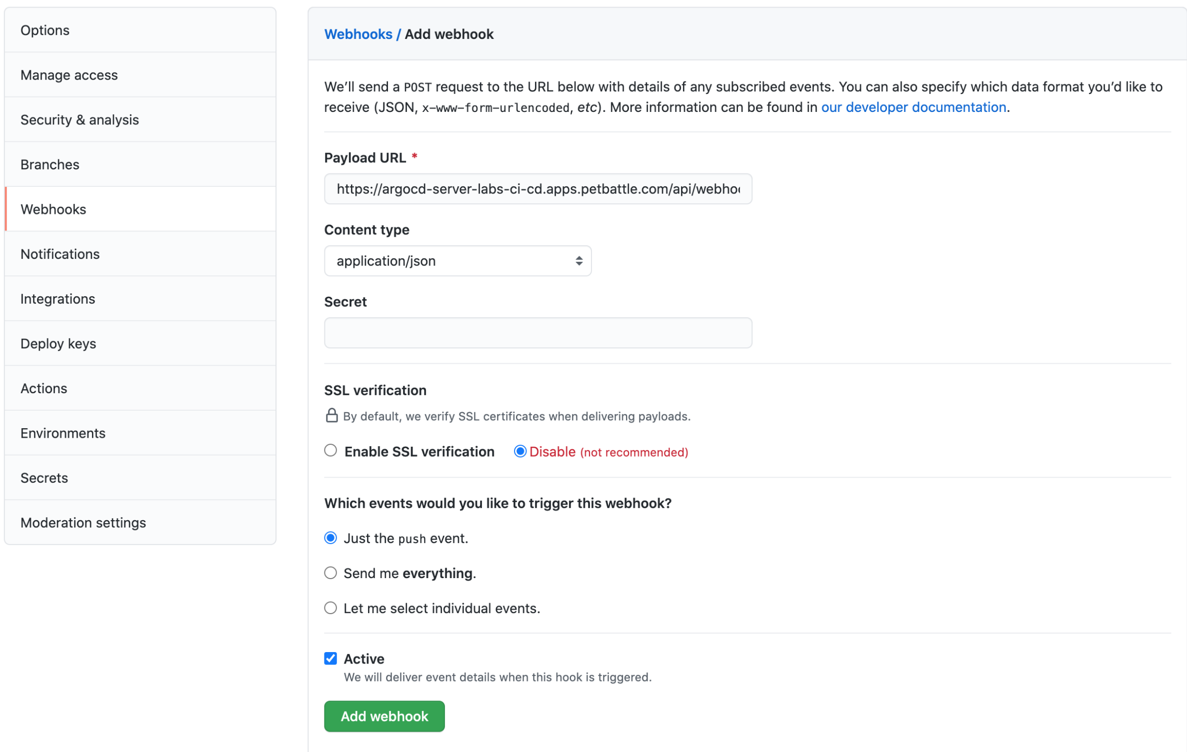


Figure 14.35: Webhook to trigger Argo CD on Git commit

Jenkins

Every time we commit to our source code repository, we want Jenkins to run a build. We're using the multibranch plugin for Jenkins, so this means that when we commit to the repository, the webhook will trigger a branch scan, which should bring back any new feature branches to build pipelines or create builds for any new code commits on any branch.

Configuring the Jenkins webhook for the `pet-battle` frontend is simple. On GitHub's *Hooks* page, add the URL to our Jenkins instance in the following form, where the trigger token is the name of our GitHub project. As a convention, I tend to use the name of the Git project as the token, so the same would apply for the backend if you were building it using Jenkins too:
`JENKINS_URL/multibranch-webhook-trigger/invoke?token=[Trigger token]`

For example, the frontend application's webhook URL would look something like this:

https://jenkins-labs-ci-cd.apps.petbattle.com/multibranch-webhook-trigger/invoke?token=pet-battle

Bringing It All Together

We have now gone through what a Jenkins file is and what it does for us. We've spoken about branching and what we mean for a build to be a release candidate (that is, a version bump and on master/main). We've touched on deploying using Helm and GitOps to commit our change and have Argo CD roll out the change for us...but how do we connect Jenkins up to all this magic?

As with all these things, there are several ways. We *could* open up Jenkins and hit **New Job > Multibranch Pipeline**, configure it to point to our Git repository, and set it to be triggered by a webhook, but that feels like the old way of doing things. I want a repeatable process so I don't have to do this step each time. Enter our `seed-multibranch-pipelines` job! Some of you may have noticed that Jenkins was configured to point to our organization's GitHub for PetBattle when we deployed the Helm chart from Ubiquitous Journey. We set some environment variables on the image (in `ubiquitous-journey/values-tooling.yaml`) to point to our GitHub organization as follows:

```
- name: GITHUB_ACCOUNT
value: 'petbattle'
- name: GITHUB_ORG
value: 'true'
```

If you're following along with a fork of the Ubiquitous Journey and want to see the pipeline run end to end, update both ARGOCD_CONFIG_REPO to point to your fork and QUAY_ACCOUNT to resolve to your user on Quay.io.

These are used by the `seed-multibranch-pipelines` job that is baked into the Jenkins image to scan the organization for repositories that contain a `Jenkinsfile` and are not archived. If it finds any, it will automatically scaffold out multibranch Jenkins jobs for us. In our case, we have a `Jenkinsfile` for both the Cats API and the PetBattle frontend, so jobs are created for us without having to configure anything! If you're following along and not using GitHub but GitLab, you can set `GITLAB_*` environment variables to achieve the same effect.

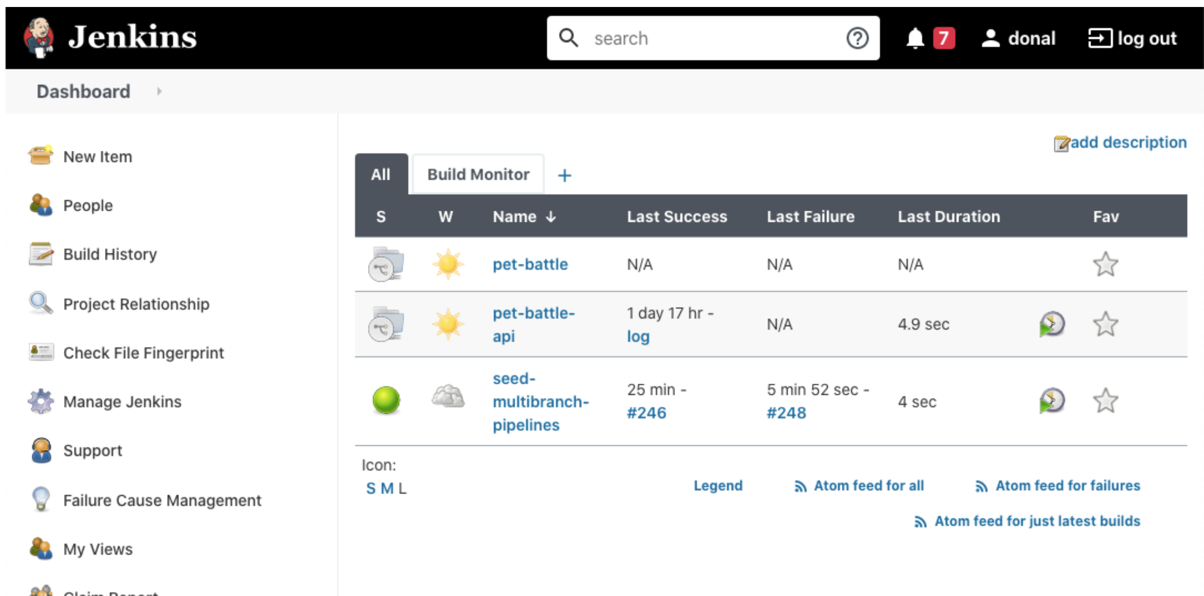


Figure 14.36: Jenkins seed to scaffold out our Jenkins jobs

If you open Jenkins and drill down into the `pet-battle` folder for the frontend code base, you should see builds; for example, a Git branch called `cool-new-cat` and the `master` with pipeline executions for each of them. Opening the Blue Ocean view, we get a much better understanding of the flow control we built, as previously discussed.

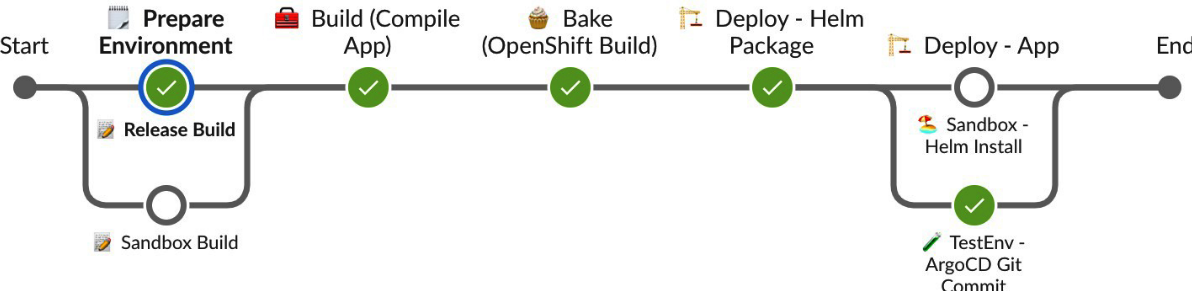


Figure 14.37: Jenkins release candidate pipeline

For the master branch, which we deem to be a release candidate, the artifacts that are built could go all the way. If we are updating our application, we bump the manifest version along with any changes we're bringing in and Git commit, which should trigger the build. From this point, our build environment is configured, and the pipeline should execute. We target an external repository and the image that's built will be pushed to Quay.io for portability across multiple clusters. Our Helm chart's values are patched and pushed to Nexus for storage. If we need to update our Helm chart itself, for example, to add some new configuration to the chart or add a new Kubernetes resource, we should of course bump the chart version too. For our deployment, we patch the Argo CD config repository (Ubiquitous Journey) with the new release information, and it should sync automatically for us, deploying our application to the `labs-test` namespace! We then run a verify step to check that the version being rolled out matches the new version (based on the labels) and has been successful.

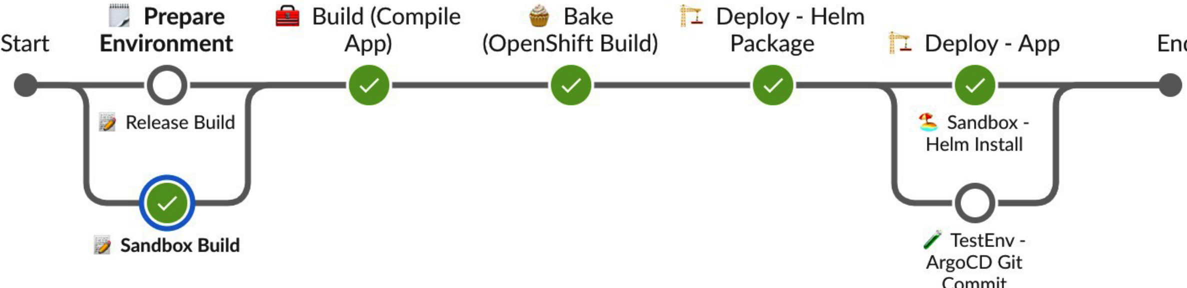


Figure 14.38: Jenkins feature development pipeline

For our feature branches, the idea is much the same, but without the need for an external repository. Our charts are also manipulated to override the name to include the branch. This means that on each commit to a feature branch, we get a new application deployed containing the branch name in the route. So, for our `cool-new-cat` branch, the application is deployed as `cool-new-cat-pet-battle` and is available in the developmental environment.

The remaining stages that were added to the Big Picture, System Test and Promote, will be covered in the next chapter, when we look in more detail at the testing for PetBattle.

What's Next for Jenkinsfile

Jenkins has been around for quite some time. It is not the most container-native approach to building software but there is a rich ecosystem surrounding it. It's lasted a long time because people like it! Hopefully, this gives you a taste of what can be done with Jenkins for our PetBattle applications, but it's by no means the end. There are a few plot holes in the story, as some of you may have noticed. For example, once a build has been successfully deployed to the test environment, how do I promote it onward? Should I do more testing? Well, the answer will come in the next chapter when we explore system tests and extend our pipeline further to include promoting images. At the end of a successful pipeline execution, the values file in our repository is not updated; we should be thinking about writing the successful build artifact details back to the repository, so it's always got a sensible default set to what is currently deployed.

The stages we have written here are fairly massive and do have some `bash` and other logic inside of them. If you were to build a non-frontend application, for example, you would want to build something in Golang. For the most part, the only thing that needs to change is the Build stage, as the act of putting something in a box, and how we package our Helm chart and deploy the app, remains the same. Once the app, in any language or framework, is in a container, then how we ship it remains the same. This means there is a high potential for reusing the code in the Bake and Deploy stages again and again, thus lowering the bar for adopting new technologies on a platform such as OpenShift. But be careful – copying and pasting the same steps across many jobs in a large estate of apps can lead to one mistake being copied around. Changes to the pipeline can become costly too, as you have to update each `Jenkinsfile` in each repository.

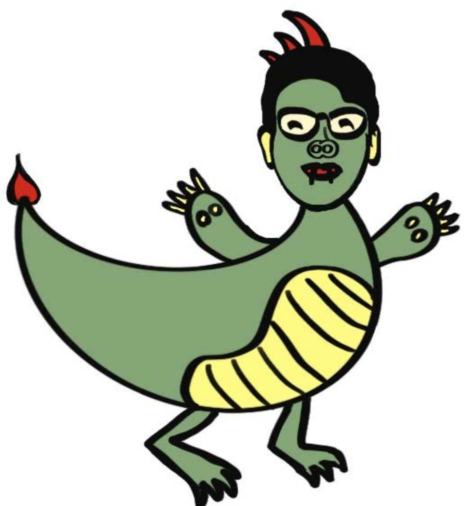


Figure 14.39: A lurking dragon, watch out!

Jenkins does tackle these problems with the use of shared libraries, and more recently the **Jenkins Templating Engine** (**JTE**). The JTE tackles the problem by enforcing pipeline approaches from a governance point of view. While this might seem like a great way to standardize across an enterprise—here be dragons!

Applying a standard pipeline without justification or the ability for teams to pull requests and make changes for their own specific use case is the same as having Dev and Ops in separate rooms. We've worked with plenty of customers who have tried approaches like this and ultimately it makes them go slower, rather than faster. The teams putting the pipelines in place think they're helping and providing a great service, but when things go wrong, they are the bottleneck to fixing it. For some teams, the hammer approach might not be applicable for their use case and so the pipeline becomes something in the way for them to go faster.

Tekton is another way for us to get greater pipeline reusability and also honor more of our GitOps landscape. Let's explore it now for our Java microservices.

Tekton–The Backend

Tekton[8] is an open source cloud-native CI/CD tool that forms the basis for OpenShift Pipelines.[9]

Tekton Basics

There are many similarities between what Jenkins does and what Tekton does. For example, both can be used to store pipeline definitions as code in a Git repository. Tekton is deployed as an operator in our cluster and allows users to define in YAML `Pipeline` and `Task` definitions. Tekton Hub[10] is a repository for sharing these YAML resources among the community, giving great reusability to standard workflows.

8 https://tekton.dev

9 https://docs.openshift.com/container-platform/4.7/cicd/pipelines/understanding-openshift-pipelines.html

10 https://hub.tekton.dev

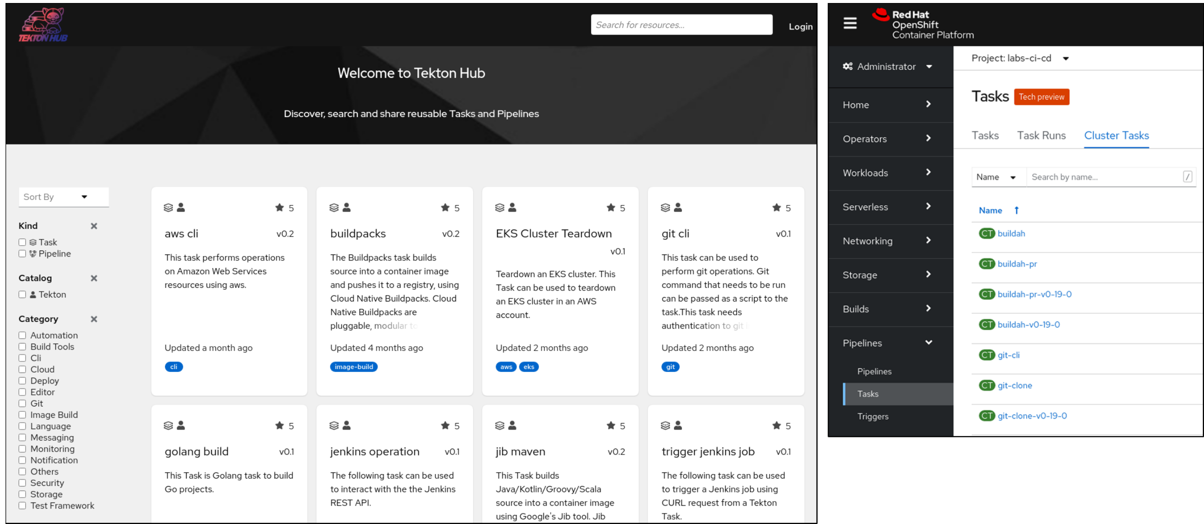


Figure 14.40: Tekton Hub and OpenShift Cluster tasks

OpenShift also makes these available globally as `ClusterTasks`. To write a pipeline you can wire together these task definitions. OpenShift provides a guided `Pipeline` builder UI for just this task. You link various tasks together and define parameters and outputs as specified in each task definition.

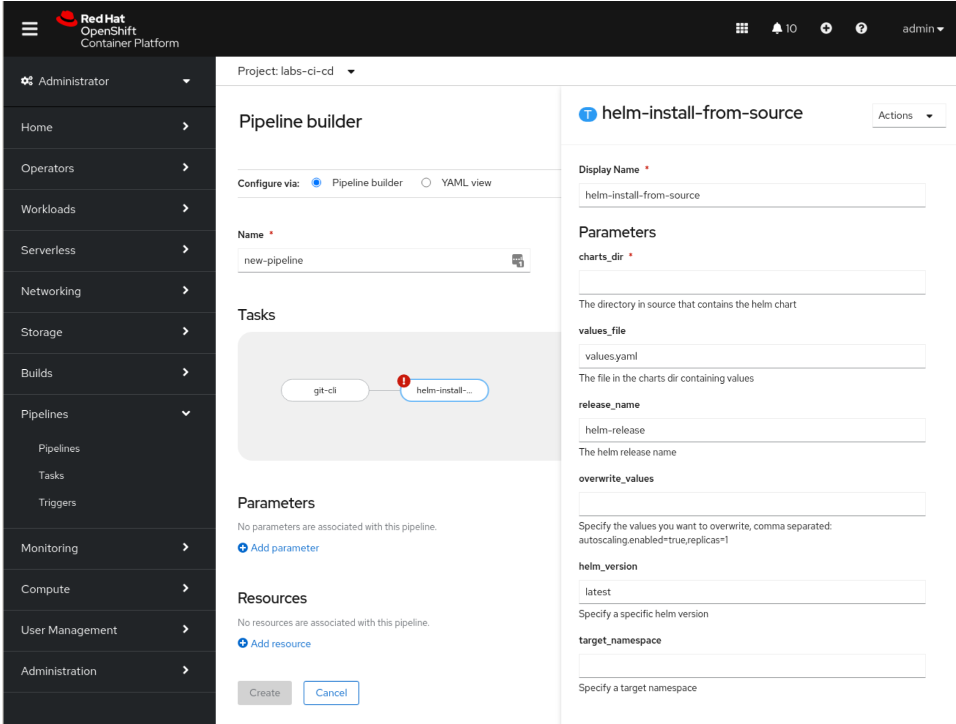


Figure 14.41: OpenShift Pipeline builder UI

There are numerous task activities in our pipeline definitions that require persistent storage. When building our backend PetBattle API and Tournament applications using Maven, we pull our Java dependencies via our Nexus repository manager. To speed up this process, we can perform the same caching we might do on our laptops and store these locally between builds in a `.m2/repository` folder and share this between builds. We also use persistent storage for built artifacts so they can be shared between different steps in our pipeline. Another use case is to mount Kubernetes secrets into our pipelines:

```
# maven pipeline
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: maven-pipeline
  labels:
    petbattle.app/uj: ubiquitous-journey
spec:
  workspaces:
    - name: shared-workspace
    - name: maven-settings
    - name: argocd-env-secret
    - name: maven-m2
    - name: git-auth-secret


# binding the workspace in the PipelineRun object
  resourcetemplates:
    - apiVersion: tekton.dev/v1beta1
      kind: PipelineRun
      metadata:
      ...
        workspaces:
        - name: shared-workspace
          persistentVolumeClaim:
            claimName: build-images
        - name: maven-settings
          persistentVolumeClaim:
            claimName: maven-source
        - name: argocd-env-secret
          secret:
            secretName: argocd-token
        - name: maven-m2
          persistentVolumeClaim:
```

```
          claimName: maven-m2
      - name: git-auth-secret
        secret:
          secretName: git-auth
```

In Tekton, we link these Kubernetes objects with the named `workspaces` when we create what is called the `PipelineRun`, a piece of code that represents one run of a pipeline. Similarly, the execution of a single task is a `TaskRun`. Each `workspace` is then made available for the tasks in that `PipelineRun` as shown.

Reusable Pipelines

There are some choices to be made before you start writing and designing your Tekton pipeline. The first is to choose whether you write a pipeline for each application, or whether you write reusable pipelines that can be used for applications that are similar.

In PetBattle, we started with one pipeline per application; this is similar to having a `Jenkinsfile` in each application Git repository. Both the API and Tournament PetBattle applications are built using Java, Quarkus, and Maven, so it makes sense to consolidate the pipeline code and write a reusable parameterized pipeline for these two applications because they will always have similar tasks. We use our `maven-pipeline` in PetBattle to do this.

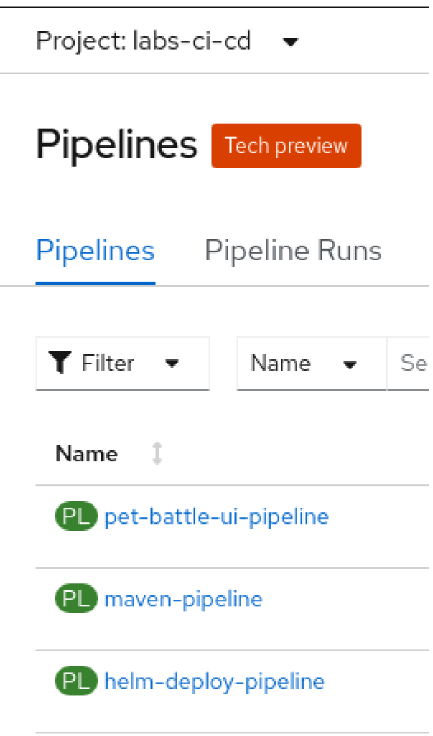


Figure 14.42: PetBattle's Tekton pipelines

Of course, you could keep the reuse to the Task level only but we share common tasks across the PetBattle UI, API, and Tournament applications. Ultimately, the development team has to balance the benefits of maintaining one pipeline over application pipeline autonomy. There is no one-size-fits-all answer.

Build, Bake, Deploy with Tekton

The next step is to start designing what we put into our pipeline. This is a very iterative process! In our Big Picture, we talked about the Build, Bake, and Deploy process, so it makes sense to add pipeline task steps that follow this methodology.

Tasks

- T git-clone (fetch-app-repository)
- T git-clone (fetch-cicd-repository)
- T maven (code-analysis)
- T sonarqube-quality-gate-check (quality-gate-check)
- T maven (maven-run)
- T openshift-kustomize (kustomize-build-config)
- T openshift-client (oc-start-build)
- T helm-upload-chart (upload-chart)
- T openshift-client (oc-tag-image-test)
- T openshift-client (oc-tag-image-stage)
- T helm-install-from-chartrepo (helm-install-apps-dev)
- T git-commit-argo-versions (git-commit-test)
- T helm-install-from-source (helm-argocd-apps-test)
- T argocd-sync-and-wait (argocd-sync-application-test)
- T git-commit-argo-versions (git-commit-stage)
- T helm-install-from-source (helm-argocd-apps-stage)
- T argocd-sync-and-wait (argocd-sync-application-stage)

Figure 14.43: The list of task definitions used by PetBattle's Tekton Pipelines

The `maven-pipeline` starts by cloning the application and CI/CD (Ubiquitous Journey) repositories into the shared `workspace`. We check the code quality by invoking Maven to build and test the application, with quality reports being uploaded to our SonarQube image.

We check that the quality gate in SonarQube has passed and then invoke Maven to package our application. Tekton offers us useful constructs to retry a task step if it fails by specifying the number of `retries` as well as the ordering of task steps using the `runAfter` task name list.

```
    - name: quality-gate-check
      retries: 1
      taskRef:
        name: sonarqube-quality-gate-check
      workspaces:
        - name: output
          workspace: shared-workspace
      params:
      - name: WORK_DIRECTORY
        value: "$(params.APPLICATION_NAME)/$(params.GIT_BRANCH)"
      runAfter:
      - save-test-results
```

In Java Quarkus, the packaging format could be a fat JAR, an exploded fast JAR, or a native GraalVM-based image. There are various trade-offs with each of these formats.[11] However, we are using the exploded fast JAR in PetBattle, which allows us to trade off between faster build times or faster startup times. This is the end of the Build stage. We have moved the unit testing left in our pipeline, so we get fast feedback on any code quality issues before we move on to the Bake and Deploy pipeline phases.

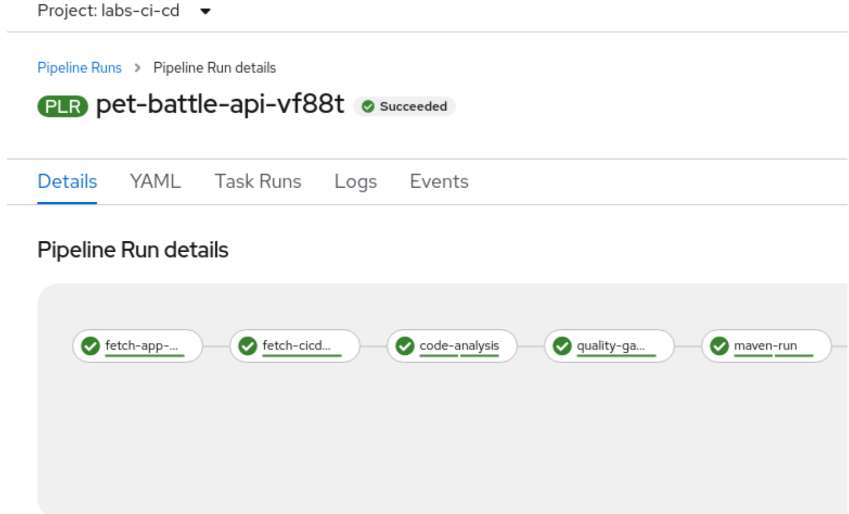


Figure 14.44: The view of a PipelineRun in OpenShift showing the tasks being executed

11 https://quarkus.io/guides/maven-tooling

The Bake stage is next. We use a standard OpenShift BuildConfig object, which is loaded using Kustomize, as we do not package that with our Helm chart. We perform a binary build using the `oc start build` command on the packaged application. We decided not to upload the built application package to Nexus because we want to work with container images as our unit of deployment. If we were building libraries that needed to support our services, then they should be captured in Nexus at this stage. It is worth pointing out that we could also push the image to an external registry at this point in time so it can be easily shared between OpenShift clusters.

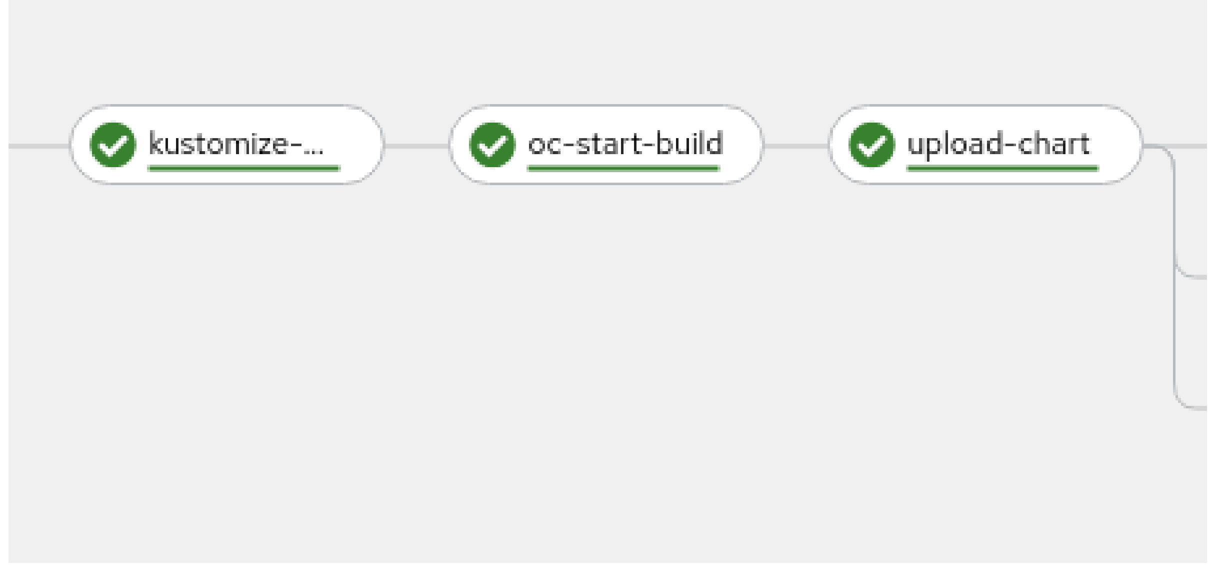


Figure 14.45: Bake part of the pipeline

The next step is to lint and package the application Helm chart. The versioned chart is then uploaded to Nexus. If we were on an application branch, the next pipeline step would be a `helm install` into the `labs-dev` project. We can make use of `when` statements in our Tekton pipeline to configure such behavior:

```
- name: helm-install-apps-dev # branches only deploy to dev
      when:
        - Input: "$(params.GIT_BRANCH)"
          Operator: notin
          Values: ["master","trunk","main"]
      taskRef:
        name: helm-install-from-chartrepo
```

When on trunk/HEAD, the ImageStream is versioned and tagged into the namespaces we are going to deploy our application to (`labs-test`, `labs-staging`). Because we are practicing GitOps, the applications are deployed using Argo CD and Git. The Argo CD app-of-apps values files are updated with the new chart and image versions. This is checked into source code by the pipeline and `git commit` is executed. Argo CD is configured to automatically sync our applications in `labs-test` and `labs-staging`, and the last step of the pipeline is to make sure the sync task was successful.

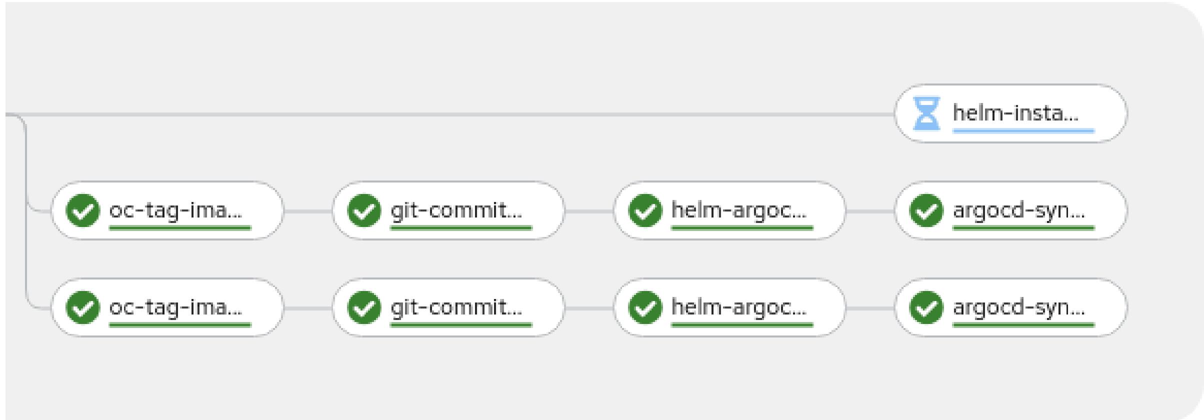


Figure 14.46: Deploy part of the pipeline

There is a lot of pipeline information available to the developer in the OpenShift web console and all of the pipeline task logs can be easily seen.

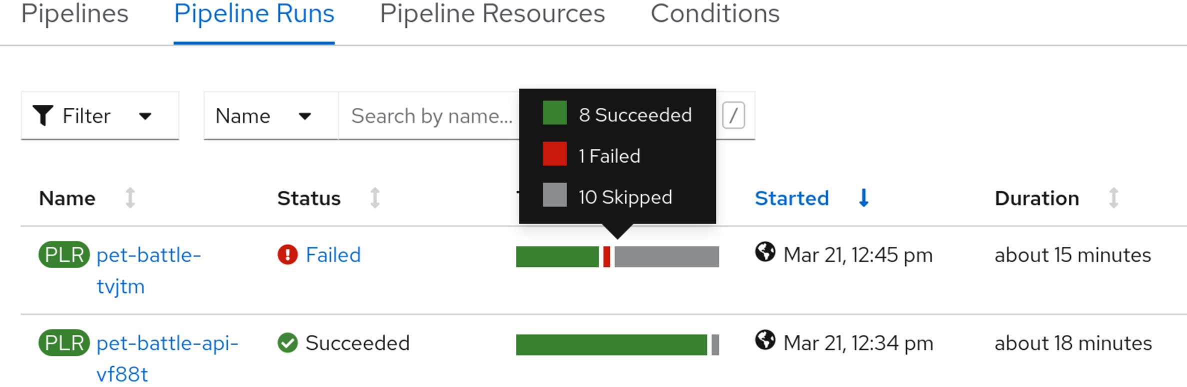


Figure 14.47: Tekton pipeline progress and status hover

Tekton also has a great command-line tool called `tkn`, which can be used to perform all of the pipeline actions available in the OpenShift console, such as viewing logs, starting pipeline runs, and defining Tekton objects.

```
$ tkn pr list -n labs-ci-cd
```

```
NAME                           STARTED    DURATION     STATUS
pet-battle-tournament-44vg5    1 days ago 27 minutes   Failed
pet-battle-9d9c7               2 days ago 19 minutes   Succeeded
pet-battle-api-jcgn2           2 days ago 21 minutes   Succeeded
pet-battle-kfch9               2 days ago 5 minutes    Failed
pet-battle-tournament-br5xd    2 days ago 24 minutes   Failed
pet-battle-api-5l4sd           2 days ago 23 minutes   Failed
```

Let's now take a look at how we can trigger a build.

Triggers and Webhooks

On every developer push to Git, we wish to trigger a build. This ensures we get the fastest feedback for all of our code changes. In Tekton this is achieved by using an `EventListener` pod object. When created, a pod is deployed, exposing our defined trigger actions.

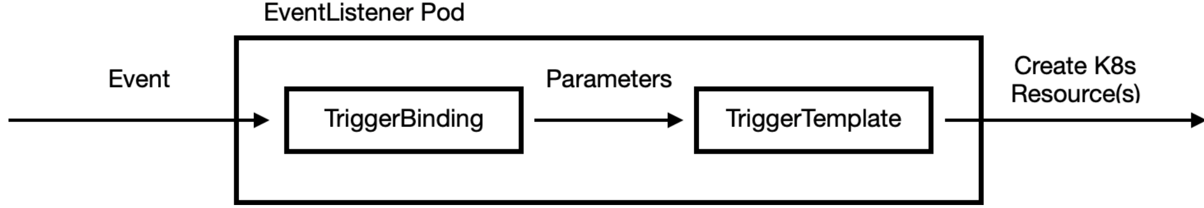


Figure 14.48: Tekton Triggers flow

Tekton Triggers work by having `EventListener` objects receive incoming webhook notifications, processing them using an interceptor, and creating Kubernetes resources from templates if the interceptor allows it, with the extraction of fields from the body of the webhook (there's an assumption that the body is a JSON file):

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: EventListener
metadata:
  name: github-webhook
  labels:
    app: github
spec:
  serviceAccountName: pipeline
  triggers:
    - name: pet-battle-api-webhook-all-branches ...
    - name: pet-battle-api-webhook-pr ...
    - name: pet-battle-tournament-webhook-all-branches ...
    - name: pet-battle-tournament-webhook-pr ...
    - name: pet-battle-webhook-all-branches ...
    - name: pet-battle-webhook-pr ...
```

In OpenShift, we expose the EventListener webhook endpoint as a route so that it can be wired into Git. Different types of **source control managers** (**SCMs**) define different webhook payloads. We are using GitHub, so it is those webhook payloads[12] that we need to use to help define the parameters used to create the TriggerBinding and pass to our TriggerTemplate. The TriggerTemplate then defines the Tekton resources to create. In our case, this is a PipelineRun or TaskRun definition.

```
triggers:
    - name: pet-battle-api-webhook-all-branches
      interceptors: # fixme add secret.ref
        - cel:
            filter: >-
              (header.match('X-GitHub-Event', 'push') &&
                body.repository.full_name ==
                                        'petbattle/pet-battle-api')
            overlays:
            - key: truncated_sha
              expression: "body.head_commit.id.truncate(7)"
            - key: branch_name
              expression: "body.ref.split('/')[2]"
            - key: app_of_apps_key
              expression: "body.repository.name.replace('-','_',-1)"
      bindings:
      - kind: TriggerBinding
        ref: github-trigger-binding
      template:
        ref: pet-battle-api-maven-trigger-template
```

Tekton uses an expression language, known as the **Common Expression Language** (**CEL**),[13] to parse and filter requests based on JSON bodies and request headers. This is necessary because of the differing webhook payloads and potentially different Git workflows. For example, we are using GitHub and treat a pull request differently from changes to our main/HEAD. One customization we make that you can see above is to define the Argo CD app-of-apps key in the trigger binding based on the Git repository name. This allows us to check the synchronization of just the one application that changed and not the whole application suite during the Deploy phase of our pipeline. While triggering seems complex, the flexibility is required when dealing with all the various Git SCMs and workflows that are available to development teams.

12 https://docs.github.com/en/developers/webhooks-and-events/webhook-events-and-payloads

13 https://github.com/google/cel-go

There are some convenience templates loaded into the labs-ci-cd project by Ubiquitous Journey that can be used to manually trigger a PipelineRun—this is handy if you have not configured the GitHub webhook yet.

```
$ oc -n labs-ci-cd process pet-battle-api | oc -n labs-ci-cd create -f-
$ oc -n labs-ci-cd process pet-battle | oc -n labs-ci-cd create -f-
$ oc -n labs-ci-cd process pet-battle-tournament | oc -n labs-ci-cd create -f-
```

You can manually add webhooks to your GitHub projects[14] that point to the EventListener route exposed in the labs-ci-cd project.

```
$ oc -n labs-ci-cd get route webhook \
     -o custom-columns=ROUTE:.spec.host --no-headers
```

Otherwise, check out the PetBattle Ubiquitous Journey documentation for Tekton tasks that can be run to automatically add these webhooks to your Git repositories.

GitOps our Pipelines

Our pipeline, task, trigger, workspace, and volume definitions are themselves applied to our labs-ci-cd project using GitOps. The idea here is to minimize how hard it is to adapt our pipelines. We may want to add some more security checks into our pipeline steps, for example. If there are testing failures, or even service failures in production, then we need to adapt our pipelines to cater for further quality controls or testing steps. Adding new tools or modifying task steps becomes nothing more than pushing the pipeline as code definitions to Git.

```
# PetBattle Tekton objects
  - name: tekton-pipelines
    destination: labs-ci-cd
    enabled: true
    source: https://github.com/petbattle/ubiquitous-journey.git
    source_path: tekton
    source_ref: main
    sync_policy: *sync_policy_true
    no_helm: true
```

Within our Tekton source folder, we use Kustomize to apply all of the YAML files that define our Tekton objects. These pipeline objects are kept in sync by Argo CD.

14 https://docs.github.com/en/developers/webhooks-and-events/creating-webhooks

Which One Should I Use?

The CI/CD tooling landscape is massive[15] and also extremely vibrant and healthy. The CNCF landscape for tools in this category has no less than 36 products and projects today. In trying to answer the question of which one you should use; it is best to consider multiple factors:

- Does your team have previous skills in a certain tooling or language? For example, pipelines as code in Jenkins use the Groovy language, so if your team has Groovy or JavaScript skills, this could be a good choice.
- Does the tool integrate with the platform easily? Most of the tools in CNCF have good integration with Kubernetes already and have a cloud-native pedigree. That does not mean that all tools are the same in terms of deployment, platform integration, or lifecycle management–some may be **Software as a Service** (**SaaS**)-only offerings with agents, whereas some can be deployed per team using namespace isolation on your cluster. Others, such as Argo CD and Tekton, can be deployed at cluster scope using the operator pattern, and have their lifecycle managed via the **Operator Lifecycle Manager** (**OLM**). Tekton has great web console integration with OpenShift because of the OpenShift Pipelines operator.
- Tool deployment model: Jenkins and Argo CD both have a client-server model for deployment. This can be problematic at larger scales, such as when looking after thousands of pipelines or hundreds of applications. It may be necessary to use multiple deployments to scale across teams and clusters. Argo CD and Tekton extend Kubernetes using CRDs and operator patterns, so deployment is more Kubernetes-native in its scaling model.
- Enterprise support: Most, but not all, the tools have vendor support. This is important for enterprise organizations that need a relationship with a vendor to cover certification, training, security fixes, and product lifecycles.

15 https://landscape.cncf.io/card-mode?category=continuous-integration-delivery&grouping=category

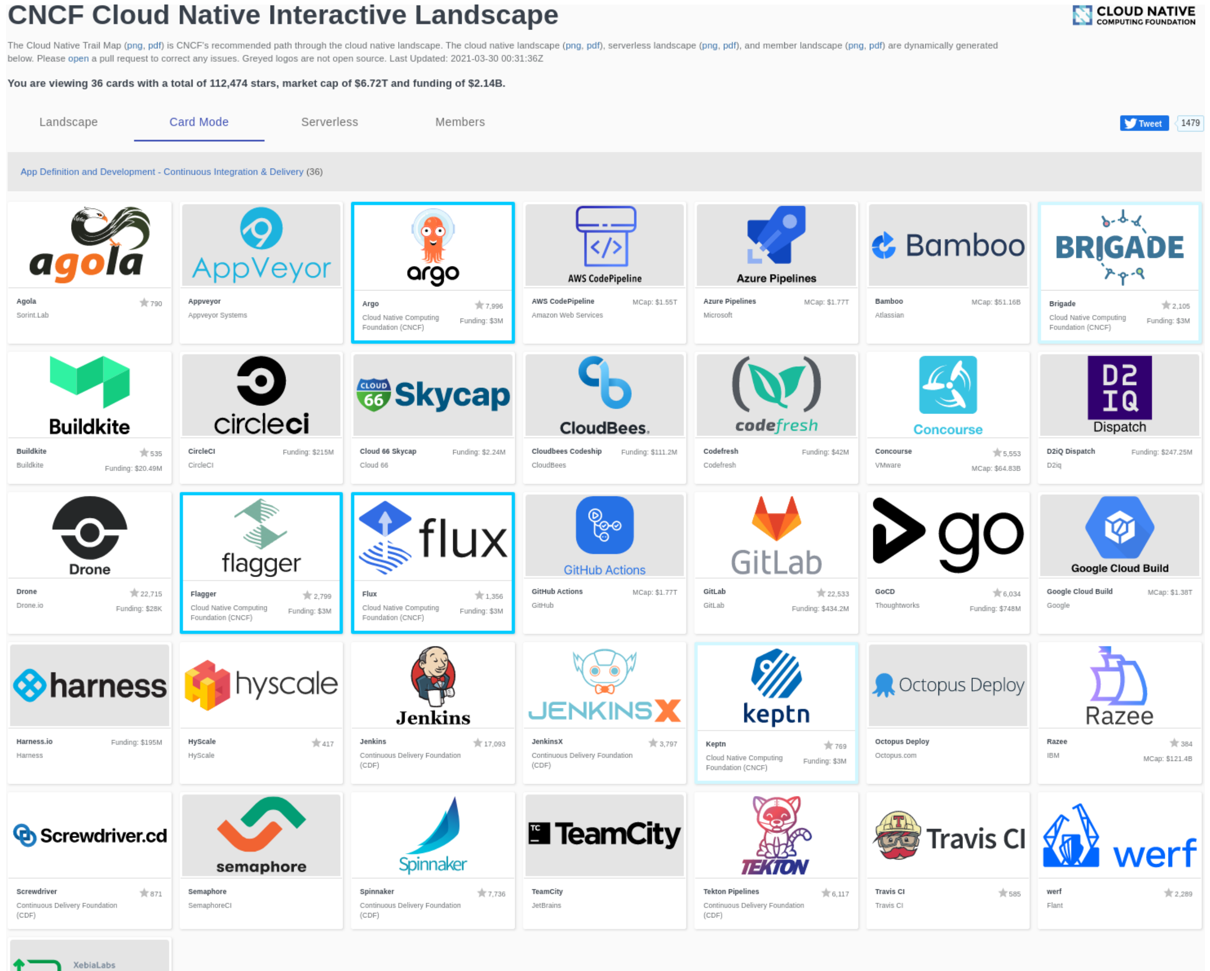


Figure 14.49: CNCF CI/CD tooling landscape

- Active open-source communities: A vibrant upstream community is important as a place to collaborate and share code and knowledge. Rapid development of features and plugins often occurs in a community based on real user problems and requests.
- One tool for both CI and CD or use different tools? As we have shown with PetBattle, sometimes it makes sense for CI to be a push-type model, and CD to be a pull-type model using different tools.

- Extensibility model: This is important for the ecosystem around the tooling. Jenkins has a great plugin model that allows lots of different extensions to the core. Tekton has a similar model, but it is different in that users have the ability to use any container in a task. It is important to weigh up these extensions as they offer a lot of value on top of the core tool itself. A good example is that Tekton does not manage test dashboards and results as well as Jenkins and its plugins do, so we might lean on Allure to do this. Reporting and dashboarding extensions are important to make the feedback loop as short as possible during CI/CD.

Once you have considered a few of these ideals, hopefully you will agree on the right set of tools for your product and team. A measure of design and planning is required to answer the question of where the various steps in your continuous deployment happen and what application packaging approach should be used (templated or not templated, for example). By now, we have instilled an experiment-driven approach to answering these types of questions, where it is not one or the other tool, but about choosing the right tool for the job at hand!

Conclusion

In this chapter we introduced how we use Git as the single source of truth. We covered taking our source code and packaging it using either

Tekton or Jenkins. In the next chapter, we will focus on testing, introducing a new component to our app using Knative, running A/B tests, and capturing user metrics using some of the advanced deployment capabilities within OpenShift.

15

Run It

There is a saying that *your code has no value until it runs in production*. The sentiment here is that until your customers use your software, it's of limited value for your business or organization. It is certainly a broad generalization! However, it does speak to the essential nature of software that its utility is directly related to being able to run it for whatever purposes it was ultimately written for. To reach production with the quality of service that our customers expect, all of the code must be put through its paces.

In this chapter, we are going to explore how the PetBattle team tests their software so they have greater confidence in its ability to run as expected in production. Testing is multifaceted, as we discussed in *Chapter 7*, *Open Technical Practices – The Midpoint*, and we are going to cover in some detail the types and scope of testing, from unit tests to end-to-end testing, through to security checks and more.

When the hobbyist version of the application went live, the PetBattle founders soon discovered that malicious content was being uploaded to the site. As part of this chapter, we'll look at a modern-day solution to this problem using a trained AI-ML model.

In the last section of this chapter, we explore some common cloud deployment patterns and demonstrate A/B testing and experimentation, for gaining insight into how we can safely measure and learn the impact of deploying new features in production.

The Not Safe For Families (NSFF) Component

As we mentioned earlier, one of the major issues that we faced when running the first generation of PetBattle was online trolls uploading inappropriate images to the system. This added to the operational overhead of running the platform because the PetBattle founders would have to search MongoDB for the offending images and remove them by hand—very tedious!

Ever innovating, the team decided to try and come up with an automated solution to this problem. One approach we decided to investigate was to use **artificial intelligence (AI)** to perform image classification on the uploaded images and incorporate this into the platform.

The field of AI in itself is a fascinating area of expertise that we won't even slightly go into here, other than to say that we are using a pre-trained image classification model served by the open source TensorFlow machine learning platform.

Great, but how do we go about running this on OpenShift?

The plan is to:

1. Generate or obtain a pre-trained image classification model.
2. Build containers containing the TensorFlow serving component that can serve up the model and make predictions based on our uploaded images.
3. Deploy and run the container on OpenShift in a "scale to zero" deployment model, aka Serverless.

Why Serverless?

When deploying a container on a Kubernetes-based platform, such as OpenShift, Kubernetes takes on the responsibility of managing the running container and, by default, restarting it if it terminates due to an error. Basically, there's always a container running. This is all good and fine for containers that are constantly receiving and processing traffic, but it's a waste of system resources constantly running a container that receives traffic either occasionally or in bursts.

What we'd like to achieve is to deploy a container and have it start up only when needed, that is, during incoming requests. Once active, we want it to process the incoming requests and then, after a period of no traffic, shut down gracefully until further incoming requests are received. We'd also like the container instances to scale up in the event of a surge of incoming requests.

It is possible to automate the scaling up and down of the number of container instances running on the platform using the Kubernetes Horizontal Pod Autoscaler; however, this does not scale to zero. We could also use something like the `oc scale` command, but this requires a fair amount of scripting and component integration. Thankfully, the Kubernetes community thought about this and came up with a solution called Knative.[1]

Knative has two major components, **Knative Serving** and **Knative Eventing**. Serving is used to spin up (and down) containers depending on HTTP traffic. Knative Eventing is somewhat equivalent but is focused on spinning up containers based on events and addresses broader use cases. For the purposes of this book, we are going to focus on using Knative Serving. However, we will also give an example of how Knative Eventing could be used.

Generating or Obtaining a Pre-trained Model

We had been experimenting with image classification for a while. We started using some of the components from the Open Data Hub community (https://opendatahub.io/) and trained out models on top of pre-existing open source models that were available. We eventually generated a trained data model that could classify images that we deemed NSFF based on an implementation of Yahoo's Open NSFW Classifier,[2] which was rewritten with TensorFlow. While it was not perfect, it was a good enough model to start with.

A common pattern in the data science community is to serve up trained data models using tools such as Seldon,[3] which are part of Open Data Hub. For our purposes though, a simple object storage tool was all that was required. So, we chose MinIO,[4] a Kubernetes native object store. We decided we could scale that out later if needed, using more advanced storage mechanisms, for example, OpenShift Container Storage or AWS S3.

1 https://knative.dev/

2 https://github.com/yahoo/open_nsfw

3 https://www.seldon.io/

4 https://min.io/

We loaded the trained data model into MinIO and it looked as follows:

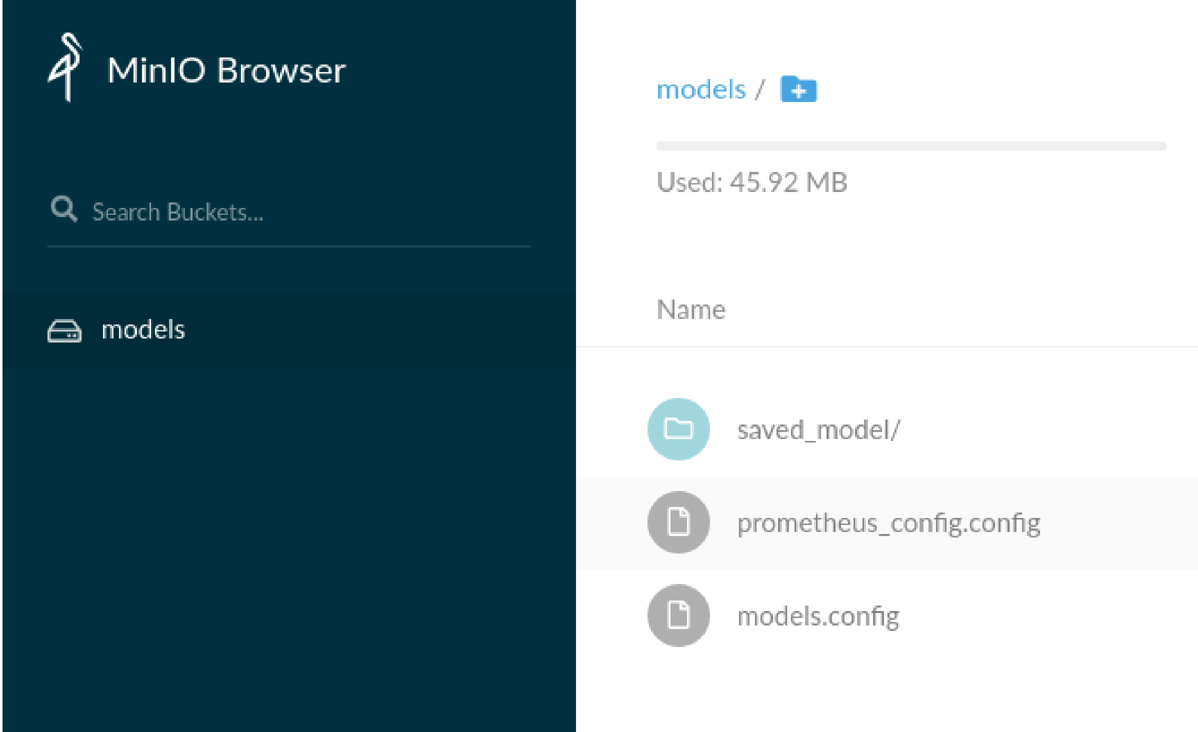


Figure 15.1: TensorFlow data model saved in MinIO

The saved model is something we can serve up using TensorFlow Serving,[5] which basically gives us an API endpoint to call our saved model with. There is an open source TensorFlow serving image we can deploy and it's a matter of configuring that to find our saved model in our S3 storage location.

We have glossed over the large portion of engineering that goes into making AI, ML, and Ops pipelines not because it is not an interesting subject, but mainly because it would require a whole other book to do it justice! If this subject is close to your heart, then take a look at the Open Data Hub project.[6] This is an open source project based on Kubeflow,[7] providing tools and techniques for building and running AI and ML workloads on OpenShift.

5 https://www.tensorflow.org/tfx/guide/serving

6 http://opendatahub.io/

7 https://www.kubeflow.org/

The OpenShift Serverless Operator

Before we start deploying our application software for the NSFF service, we need to add the OpenShift Serverless Operator[8] to our PetBattle Bootstrap. The operator is installed at the cluster scope so that any project that wants to use the Knative components Knative Serving and Knative Eventing may do so.

Let's use GitOps, ArgoCD, and Kustomize to configure and install the serverless operator. First, we can test out the configuration with ArgoCD. Log in using ArgoCD from the command line. Add the Git repository that contains the Knative serverless operator YAML subscription and create the application:

```
# Login to ArgoCD
$ argocd login $(oc get route argocd-server --template='{{ .spec.host }}' \
-n labs-ci-cd):443 --sso --insecure

# Add our repository
$ argocd repo add \
  https://github.com/rht-labs/refactored-adventure.git

# Create the Knative Operator - this may have already been created for you
but here is how to do it on the command line.
# Create the Knative Operator
$ argocd app create knative\
  --repo https://github.com/rht-labs/refactored-adventure.git \
  --path knative/base \
  --dest-server https://kubernetes.default.svc \
  --dest-namespace openshift-serverless \
  --revision master \
  --sync-policy automated
```

8 https://github.com/openshift-knative/serverless-operator

Once installed, you should be able to see this installed successfully in the `openshift-serverless` namespace:

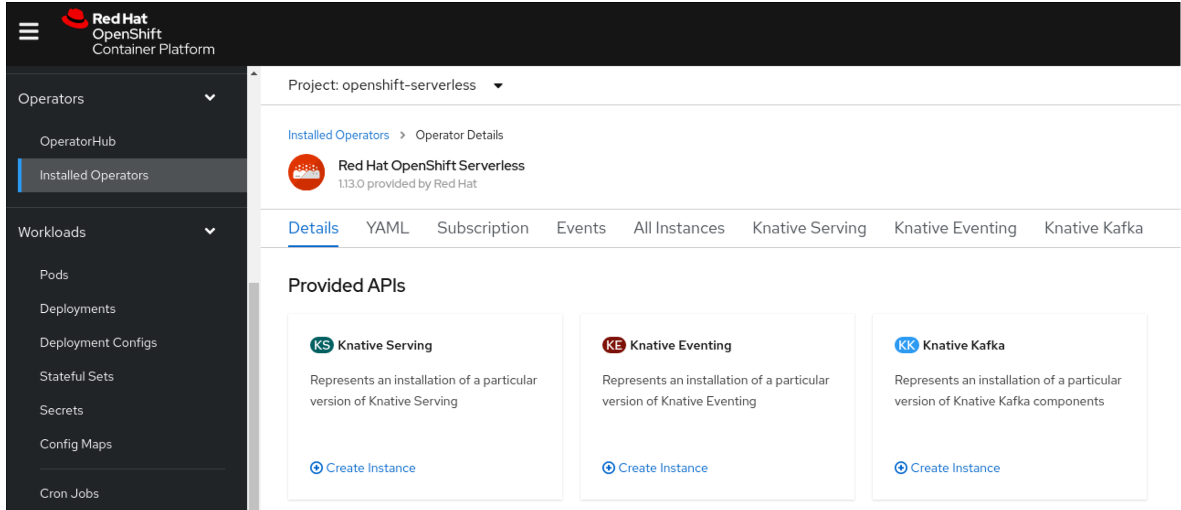


Figure 15.2: The OpenShift Serverless Operator (Knative)

We can also put this in our PetBattle UJ bootstrap from *Chapter 7*, *Open Technical Practices – The Midpoint*, so that we don't need to run these commands manually. Add the following to our `values-tooling.yaml` and check it into Git:

```
# Knative stanza in values-tooling.yaml
- name: knative
    enabled: true
    destination: openshift-serverless
    source: https://github.com/rht-labs/refactored-adventure
    source_path: knative/base
    source_ref: master
    sync_policy: *sync_policy_true
    no_helm: true
```

The operator is now ready for us to use to deploy our Knative service.

Deploying Knative Serving Services

There are a few ways in which to create Knative Serving services. We can create the Knative service definition and install that into our cluster. We have packaged this up as a Helm chart for easy installation:

```
$ helm upgrade --install pet-battle-nsff \
petbattle/pet-battle-nsff \
--version=0.0.2 \
--namespace petbattle
```

It may take a minute or so for the containers to start up and load the model data into MinIO; they may restart a few times while doing this. The output of the `oc get pods` command should look like this once successful – the MinIO S3 pod and its completed data load and a TensorFlow Knative service pod:

```
$ oc get pods --namespace petbattle

NAME                                   READY   STATUS    RESTARTS  AGE
Minio-pet-battle-nsff-594fc7759-j7lwv  1/1     Running     0       88s
minio-pet-battle-nsff-dataload-7x8jg   0/1     Completed   2       88s
minio-pet-battle-nsff-gfjhz            0/1     Completed   3       88s
tensorflowserving-pet...-7f79956d9qfp  2/2     Running     2       85s
```

After a couple of minutes, the Knative Serving TensorFlow pod will terminate because it is not yet being called. This is what's called Serverless scale to zero, that is, when there are no calling workloads there is no need to run the service. An equivalent service can also be created using the Knative command-line tool **kn**, which can be downloaded and installed from the OpenShift[9] console. This is useful if you want to create a new service or are developing a service from scratch:

```
$ kn service create tensorflowserving-pb-nsff --namespace petbattle \
  --image=docker.io/tensorflow/serving:latest \
  --cmd "tensorflow_model_server" \
  --arg "--model_config_file=s3://models/models.config" \
  --arg "--monitoring_config_file=s3://models/prometheus_config.config" \
  --arg "--rest_api_port=8501" \
  --env S3_LOCATION=minio-pet-battle-nsff:9000 \
  --env AWS_ACCESS_KEY_ID=minio \
  --env AWS_SECRET_ACCESS_KEY=minio123 \
  --env AWS_REGION=us-east-1 \
  --env S3_REGION=us-east-1 \
  --env S3_ENDPOINT=minio-pet-battle-nsff:9000 \
  --env S3_USE_HTTPS="0" \
  --env S3_VERIFY_SSL="0" \
  --env AWS_LOG_LEVEL="3" \
  --port 8501 \
  --autoscale-window "120s"
```

9 https://docs.openshift.com/container-platform/4.7/serverless/serverless-getting-started.html

Here, we use command-line arguments and environment variables to tell the TensorFlow serving image how to run. The `--image` field specifies the container image and version we wish to run – in this case, the latest TensorFlow serving image. The `--cmd` field specifies the binary in the image we wish to run, for example, the model server command `tensorflow_model_server`. The `--arg` and `--env` variables specify the configuration. The trained model is served from the `S3 minio` service so we specify how to access the S3 endpoint. There are many configurations available to Knative Serving, such as autoscaling global defaults, metrics, and tracing. The `--autoscale-window` defines the amount of data that the autoscaler takes into account when scaling, so in this case, if there has been no traffic for two minutes, scale the pod to 0.

The Knative website[10] goes into a lot more detail about the serving resources that are created when using Knative Serving and the configuration of these. To find the URL for our service, we can use this command:

```
$ kn route list
```

This gives us the HTTP URL endpoint to test our service with. It is worth noting that we can have multiple revisions of a service and that within the route mentioned previously, we can load balance traffic across multiple revisions. The following diagram depicts how this works in practice:

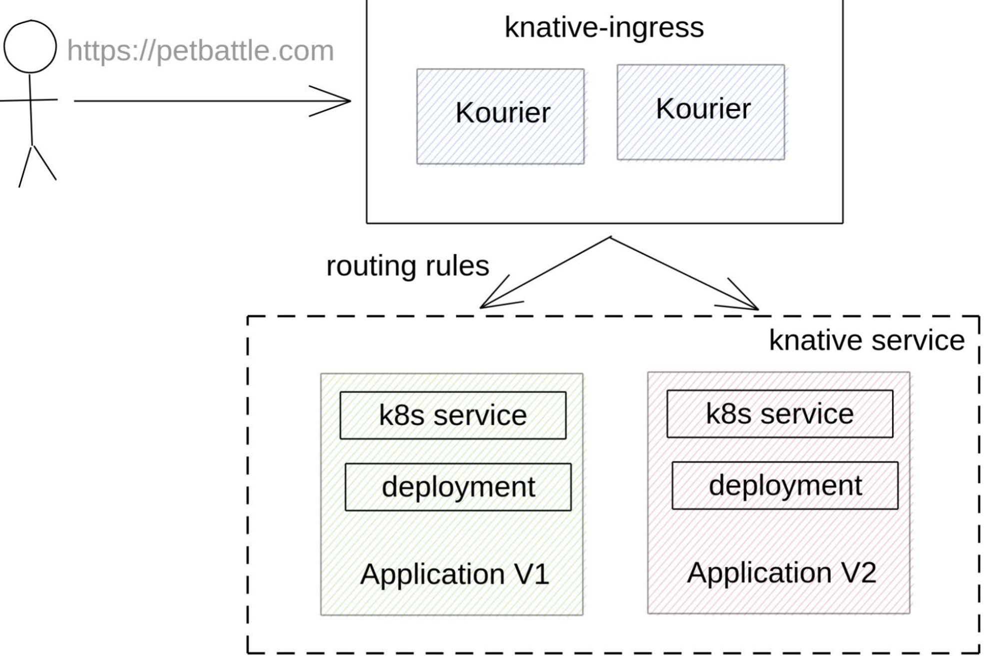


Figure 15.3: Knative routing for multiple application revisions

10 https://knative.dev/docs/serving/

Kourier[11] is a lightweight ingress router based on an Envoy gateway. Using Knative service configuration, a user can specify routing rules that Knative Serving applies. This can be very useful when we're experimenting with different AI models or wanting to do A/B, Blue/Green, or Canary-type deployments, for example.[12]

Invoking the NSFF Component

A simple HTTP GET request on the route is all that's required to invoke the component. The pod spins up and services the request usually within a couple of seconds and then spins down after a period of time (the `--autoscale-window` specified in the `kn` command-line argument, that is, 120 seconds). Using the output from the `kn list route` command, let's check if the AI TensorFlow model is available. The `state` should read `AVAILABLE`:

```
$ curl <url from kn route list>/v1/models/test_model
# For example

$ curl http://tensorflowserving-pet-battle-nsff-labs-
dev.apps.hivec.sandbox882.opentlc.com/v1/models/test_model
{
 "model_version_status": [
  {
   "version": "1",
   "state": "AVAILABLE",
   "status": {
    "error_code": "OK",
    "error_message": ""
   }
  }
 ]
}
```

11 https://developers.redhat.com/blog/2020/06/30/kourier-a-lightweight-knative-serving-ingress/

12 https://medium.com/@kamesh_sampath/serverless-blue-green-and-canary-with-knative-kn-ad49e8b6aa54

We should also see a pod spinning up to serve the request, using:

```
$ oc get pods \
-l serving.knative.dev/configuration=tensorflowserving-pet-battle-nsff \
--namespace petbattle

NAME                          READY   STATUS    RESTARTS   AGE
tensorflowserving-pet…        2/2     Running   0          21s
```

It then scales down to 0 after two minutes.

We want to test that our NSFF service works by sending it some images. We have two test sample images that have been encoded so they can be uploaded to the NSFF service.

• Not Safe for Families - Boxing

Figure 15.4: NSFF test images

Let's download these images for testing:

```
$ wget https://raw.githubusercontent.com/petbattle/pet-battle-nsff/
main/requests/tfserving/nsff-negative.json
$ wget https://raw.githubusercontent.com/petbattle/pet-battle-nsff/
main/requests/tfserving/nsff-positive.json
```

Now submit these to our NSFF service using a simple `curl` command:

```
$ HOST=$(kn service describe tensorflowserving-pb-nsff -o url)/v1/models/
test_model:predict

# Daisy Cat - Safe for Families
curl -s -k -H 'Content-Type: application/json \
  -H 'cache-control: no-cache' \
  -H 'Accept: application/json' \
  -X POST --data-binary '@nsff-negative.json' $HOST

{ "predictions": [[0.992712617, 0.00728740077]] }
# Not Safe For Families - Boxing
curl -s -k -H 'Content-Type: application/json' \
  -H 'cache-control: no-cache' \
  -H 'Accept: application/json' \
  -X POST --data-binary '@nsff-positive.json' $HOST

{ "predictions": [[0.30739361, 0.69260639]] }
```

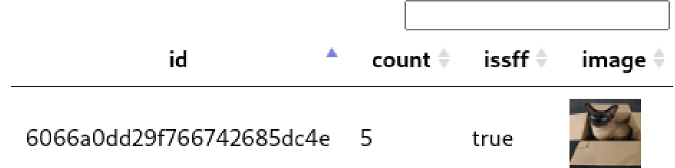
The response from our model is a `predictions` array containing two numbers. The first is a measure of **Safe for Families**, the second is a measure of **Not Safe for Families**, and they add up to 1.

So, we can see that Daisy Cat has a very high safe for families rating (0.993) compared to our wrestlers (0.014) and we can use this in our PetBattle API to determine whether any given image is safe to display. By arbitrary testing, we have set a limit of >=0.6 for images we think are safe to view in the PetBattle UI.

We can redeploy our PetBattle API service to call out to the NSFF service by setting the `nssf.enabled` feature flag to `true` and using the hostname from the Knative service from a bash shell using the command line:

```
$ HOST=$(kn service describe tensorflowserving-pet-battle-nsff -o url)
$ helm upgrade --install pet-battle-api petbattle/pet-battle-api \
--version=1.0.8 \
--set nsff.enabled=true \
--set nsff.apiHost=${HOST##http://} \
--set nsff.apiPort=80 --namespace petbattle
```

If we now upload these test images to PetBattle via the UI and check the API server, we can see that the boxing picture has a **false** value for the `ISSFF` (Is Safe for Families) flag and Daisy Cat has a **true** value:



id	count	issff	image
6066a0dd29f766742685dc4e	5	true	

Figure 15.5: PetBattle API saved images with the ISSFF flag

The API code will not return any pictures to the PetBattle UI that are deemed NSFF. For example, the API code to return all pets in the PetBattle database is filtered by the `ISSFF` flag being set to `true`:

```
@GET
@Operation(operationId = "list",
           summary = "get all cats",
           description = "This operation retrieves all cats from the
                          database that are safe for work",
           deprecated = false, hidden = false)
public Uni<List<Cat>> list() {
  return Cat.find(ISSFF, true).list();
}
```

Now that we have the API up and running it's time to test it and see if it performs as we expect.

Let's Talk about Testing

In our experience with working with many developer teams, nothing can dampen the mood of many developers quite like a discussion on the subject of testing that their software does what it's supposed to do.

For many developers, testing is the equivalent of getting a dental checkup—few like doing it but all of us need to do it a lot more. It's a set of bridges that have to be crossed (often under duress) before our precious, handcrafted, artisan-designed piece of software excellence is accepted into the nirvana that is production. Testers are seen as the other team that ensures that we've dotted our I's and crossed our T's, and they don't appreciate how hard we've suffered for our art. Basically, we write it, *throw it over the fence to test*, and they check it.

If you're reading the preceding paragraph and mentally going *yep*, *yep*, *that's us*, *that's us*, *that's how we roll*, we've got really bad news for you. *You're basically doing it wrong*. It may have made sense when big bang software releases happened every 6-12 months, but in more agile organizations with faster, more frequent releases into production, this approach is considered cumbersome and archaic. There are always exceptions to this, for example, critical control systems, highly regulated environments, and so on, but for the majority of enterprise developers, this isn't the case.

The quality of software is the responsibility of the delivery team, from the Product Owner writing user stories to the engineers writing code and the associated tests. As a wise delivery manager once said, "testing is an activity, not a role." In effective software delivery teams, testing is a continuous activity that spans the entire software development life cycle. There are a number of principles that we try to adhere to when it comes to testing:

1. Automate as much as possible, but not so much that there's no human oversight. There's always value in having users interact with the application under test, in particular when it comes to end-to-end, acceptance, and exploratory testing.
2. Testing code is as important as production code—both need to be kept up to date and removed/deprecated when not adding value.
3. One meaningful test can be worth more than hundreds of scripted test cases.

In *Chapter 7*, *Open Technical Practices – The Midpoint*, we introduced the idea of the Automation Test Pyramid. For each of the different types of tests defined in the pyramid, there are several testing tools and frameworks we use across our PetBattle application.

Generally speaking, we have chosen to use what are considered the default test tools for each of the application technology stacks as these are the simplest to use, are the best supported, have good user documentation, and are generally easy to adopt if people are new to them:

<table>
<tr><th>Application</th><th>Technology</th><th>Unit Test</th><th>Service Test</th><th>E2E Test</th></tr>
<tr><td>API</td><td>Java/Quarkus</td><td>JUnit 5</td><td>REST Assured</td><td rowspan="3">Protractor and Selenium Grid</td></tr>
<tr><td>Tournament</td><td>Java/Quarkus</td><td>JUnit 5</td><td>REST Assured
Test Containers</td></tr>
<tr><td>UI</td><td>Angular</td><td>Jest</td><td>Jest</td></tr>
</table>

Table 15.1: Test Frameworks in use

Let's take a look at some of these tests in more detail.

Unit Testing with JUnit

In both the API and Tournament applications, we have different examples of standard unit tests. Quarkus testing[13] has great support for the standard unit test framework JUnit.[14] The anatomy of all unit tests using this framework is very similar. Let's take a look at the API application `CatResourceTest.java`[15] as an example:

```
@QuarkusTest
class CatResourceTest {

  private static final Logger LOGGER = LoggerFactory
                                        .getLogger("CatResourceTest");

  @Test
  void testCat() {
    PanacheMock.mock(Cat.class);
    Mockito.when(Cat.count())
      .thenReturn(Uni.createFrom().item(23l));
    Assertions.assertEquals(23, Cat.count().await().indefinitely());
```

13 https://quarkus.io/guides/getting-started-testing

14 https://junit.org/junit5/

15 https://github.com/petbattle/pet-battle-api/blob/master/src/test/java/app/battle/CatResourceTest.java

In Java, we use annotations to make our Java class objects (POJOs) into tests. We use the `@QuarkusTest` annotation to bring in the JUnit framework for this class and we can think of the class as a test suite that contains lots of individual tests. Each method is a single test that is annotated with `@Test`. For this unit test, we don't have a database running, so we use mocks[16] for the `Cat.class`. A mock is a fake object. It does not connect to a real database, and we can use it to test the behavior of the `Cat` class. In this case, we are asserting in our test that when we call the method `Cat.count()`, which corresponds to the number of likes of our pet image in PetBattle, we receive back the expected number (23). We use the `Uni` and `await()` functions because we are using the reactive programming model in our Quarkus application.[17]

We run these unit tests as part of the automated continuous deployment pipeline and visualize and report on the tests' success and history using our CI/CD tools, including Jenkins, Tekton, and a test report tool such as Allure.[18]

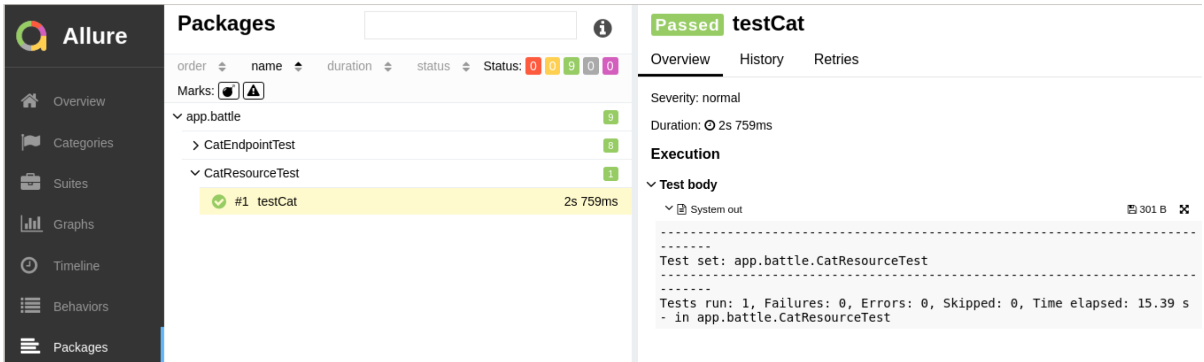


Figure 15.6: Visualization of tests using Allure

In the next section, we'll continue with service and component testing with REST Assured and Jest.

Service and Component Testing with REST Assured and Jest

Jest and REST Assured are **Behavior-Driven Development** (**BDD**) frameworks for JavaScript and Java. We covered BDD in *Chapter 7*, *Open Technical Practices – The Midpoint*. These frameworks make it super easy for developers to write tests where the syntax is obvious and easy to follow.

16 https://quarkus.io/guides/mongodb-panache

17 https://quarkus.io/guides/getting-started-reactive#mutiny

18 https://github.com/allure-framework

We are going to cover the basics of component testing the PetBattle user interface[19] using Jest. The user interface is made of several components. The first one you see when landing on the application is the home page. For the home page component, the test class[20] is called `home.component.spec.ts`:

```
describe('HomeComponent', () => {
  let component: HomeComponent;
  let fixture: ComponentFixture<HomeComponent>;

  beforeEach(async () => {...
  });

  beforeEach(() => {...
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

Each test has a similar anatomy:

- `describe()`: The name of the test suite and test specification argument
- `beforeEach()`: Runs the function passed as an argument before running each test
- `it()`: Defines a single test with a required expectation and a function with logic and assertions
- `expect()`: Creates an expectation for the test result, normally with a matching function such as `toEqual()`

So in this case, the unit test will expect the `HomeComponent` to be created correctly when the test is run.

19 https://angular.io/guide/testing

20 https://github.com/petbattle/pet-battle/blob/master/src/app/home/home.component.spec.ts

Similarly, within the API application, REST Assured is a testing tool that allows us to write tests using the familiar `Given, When, Then` syntax from *Chapter 7*, *Open Technical Practices – The Midpoint*. Let's examine one of the service API tests in the test suite `CatResourceTest.java`[21]:

```
@Test
@Story("Test pet create")
void testCatCreate() {
    CatInstance catInstance = new CatInstance();

    RestAssured.given()
            .contentType(ContentType.JSON)
            .body(catInstance.cat)
            .log().all()
            .when().post("/cats")
            .then()
            .log().all()
            .statusCode(201)
            .body(is(notNullValue()));
}
```

In this test, we are creating a `Cat` object. The `Cat` class is the data object in PetBattle that contains the pet's uploaded image, along with its PetBattle vote count, and is stored in MongoDB. In the test, given the `Cat` object, we use an `HTTP POST` to the `/cats` endpoint and expect a return status code of (`201`), which is `CREATED`. We also test the HTTP response body is not empty. It should contain the ID of the newly created `Cat`:

```
@QuarkusTest
@QuarkusTestResource(MongoTestResource.class)
@Epic("PetBattle")
@Feature("PetEndpointTest")
class CatEndpointTest {
```

21 https://github.com/petbattle/pet-battle-api/blob/master/src/test/java/app/battle/CatResourceTest.java

In this service test, we make use of the `@QuarkusTestResource` annotation to create and start an embedded MongoDB for testing against. So, this test is a bit more sophisticated than the basic unit test that was using mocks only. We also track the execution of these service tests using our test report tool:

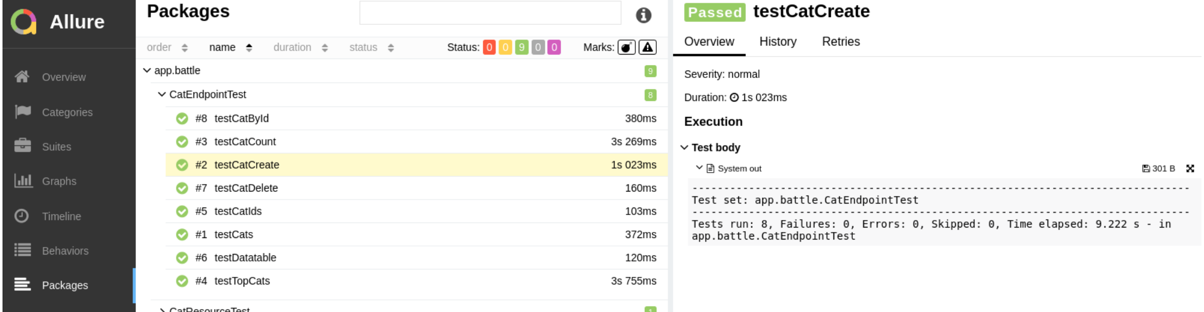


Figure 15.7: Visualization of service tests using Allure

Now we have seen what unit tests look like, let's move up the test pyramid to have a look at service-level testing.

Service Testing with Testcontainers

Integration testing is always substantially harder than unit testing as more components have to be either stood up or simulated/mocked. The next level of testing in our test pyramid is integration testing using a Java framework called **Testcontainers**.[22] Testcontainers allows us to easily create and start components such as MongoDB, **KeyCloak**, and **Infinispan** and perform tests using those components. The following classes instantiate and manage the containers and inject them into the testing life cycle of the Quarkus framework:

```
$ ls src/test/java/com/petbattle/containers/
InfinispanTestContainer.java KeycloakTestContainer.java   MongoTestContainer.
java
```

Within the integration test code at `ITPetBattleAPITest.java`, we just inject the previously created containers and use them as resources during the test:

```
$ head src/test/java/com/petbattle/integration/ITPetBattleAPITest.java

package com.petbattle.integration;
...
```

22 https://www.testcontainers.org/

```
@QuarkusTest
@DisplayName("API Test Cases")
@QuarkusTestResource(MongoTestContainer.class)
@QuarkusTestResource(InfinispanTestContainer.class)
@QuarkusTestResource(KeycloakTestContainer.class)
public class ITPetBattleAPITest {
```

This is a great example of how containers can be used as part of a testing phase. The containers are spun up, the tests are run, and the containers are removed. The only real prerequisite is that the Docker daemon is run on the machine running the tests. To run the integration tests use the command `mvn clean verify -Pintegration`.

End-to-End Testing

Our application is made up of a frontend written in Angular, which makes calls for data to two APIs. One is for tournaments and the other is for cats. We can think of the interplay between these components as the system as a whole. Any time a change is made to either of these individual applications, it should require revalidating the whole system. The end-to-end automated testing is performed primarily in the user interface but exercises the underlying services layer.

There are loads of tools to do testing from the user interface level. Some of the more popular ones are things like Selenium and Cypress, which are used to drive a web application and simulate user behavior. There are pros and cons to each – Selenium is just browser automation so you need to bring your own test frameworks, whereas Cypress is an all-in-one testing framework. Selenium Grid, when running on Kubernetes, allows us to test against multiple browsers in parallel by dynamically provisioning the browser on each test execution, meaning we don't have browsers waiting idly for us to use them.

For our end-to-end testing, we're using Protractor from the Angular team. We already deployed an instance of Selenium Grid built for Kubernetes by the Zalando team (called Zalenium https://opensource.zalando.com/zalenium/) when we deployed our tooling. Zalenium is pretty handy as it allows us to play back previous test runs and watch them live. In your cluster, if you get the route for Zalenium (`oc get routes -n labs-ci-cd`) and append `/grid/admin/live`, you can follow the tests as they execute or go to **/dashboard** to watch the historical test executions.

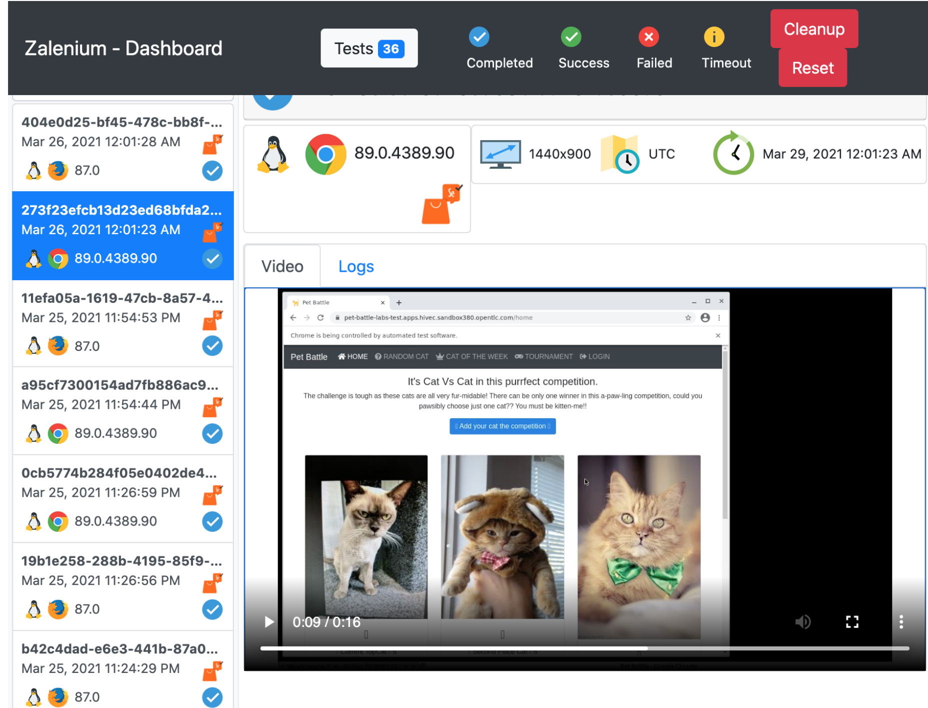


Figure 15.8: Zalenium dashboard showing test history and video playback

Our `system-tests` project (https://github.com/petbattle/system-tests) has all of the system tests that we should execute after any change is pushed to the frontend or backend services. The tests are written using Cucumber-style BDD. In fact, we should be able to connect the BDD to the acceptance criteria from our PetBattle Sprint items.

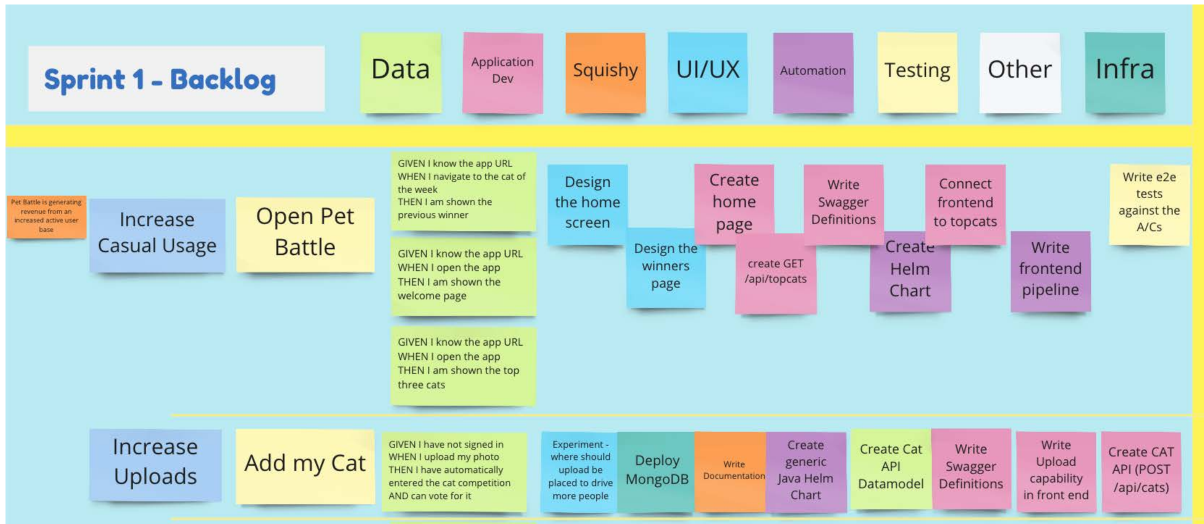


Figure 15.9: Example of BDD written as acceptance criteria on a Sprint board for PetBattle

Here's a test for a tournament feature written in the `Given`, `When`, `Then` syntax:

```
Feature: Tournament features

  Scenario: Should only be prompted to login on navigating to the
            tournament
    Given I am on the home page
    When  I move to the tournament page
    Then  I should be redirected to keycloak
```

The `system-test` project has its own Jenkinsfile, so it's already connected to Jenkins via our seed job. We won't go through the contents of this Jenkinsfile in detail. Suffice to say, the pipeline has two stages, as per our Big Picture, one to run the tests and the other to promote the app if the tests have passed. Explore the code for this in the accompanying Git repo https://github.com/petbattle/system-tests. To extend our Jenkinsfile for `pet-battle` to trigger our system test job, we just need to add another stage to trigger the job. We could use the Jenkins `post{}` block, but we only want to trigger the system tests if we're on `master` or `main` and producing a release candidate.

```
stage("🥾 Trigger System Tests") {
  options {
    skipDefaultCheckout(true)
  }
  agent { label "master" }
  when {
    expression { GIT_BRANCH.startsWith("master") || GIT_BRANCH.startsWith("main") }
  }
  steps {
      echo "TODO - Run tests"
      build job: "system-tests/${SYSTEM_TEST_BRANCH}",
            parameters: [[$class: 'StringParameterValue', name: 'APP_NAME', value: "${APP_NAME}" ],
                         [$class: 'StringParameterValue', name: 'CHART_VERSION', value: "${CHART_VERSION}"]
                         [$class: 'StringParameterValue', name: 'VERSION', value: "${VERSION}"]],
            wait: false
  }
}
```

Figure 15.10: The trigger for connecting our pipelines in Jenkins

There are a few parameters that are passed between the jobs:

- `APP_NAME`: Passed to the job so if the tests are successful, the promote stage knows what app to deploy.
- `CHART_VERSION & VERSION`: Any update to the chart or app needs to be patched in Git so this information is passed by the job that triggers the system tests.

We can run the system tests job manually by supplying this information to the job, but each service with a Jenkinsfile should be able to pass these to the system tests. This job can also be triggered from Tekton too if we were to mix the approach to the pipelines. With the two pipelines wired together, we can trigger one if the webhook is set up by running the following command:

```
$ git commit --allow-empty -m "🍌 kickoff jenkins 🦤" && git push
```

If we now check in the Jenkins Blue Ocean Web UI, we should see the following:

Figure 15.11: The system tests pipeline

On Jenkins, we should see the system tests pipeline running and promoting if successful. The Cucumber reports are also included for the job.

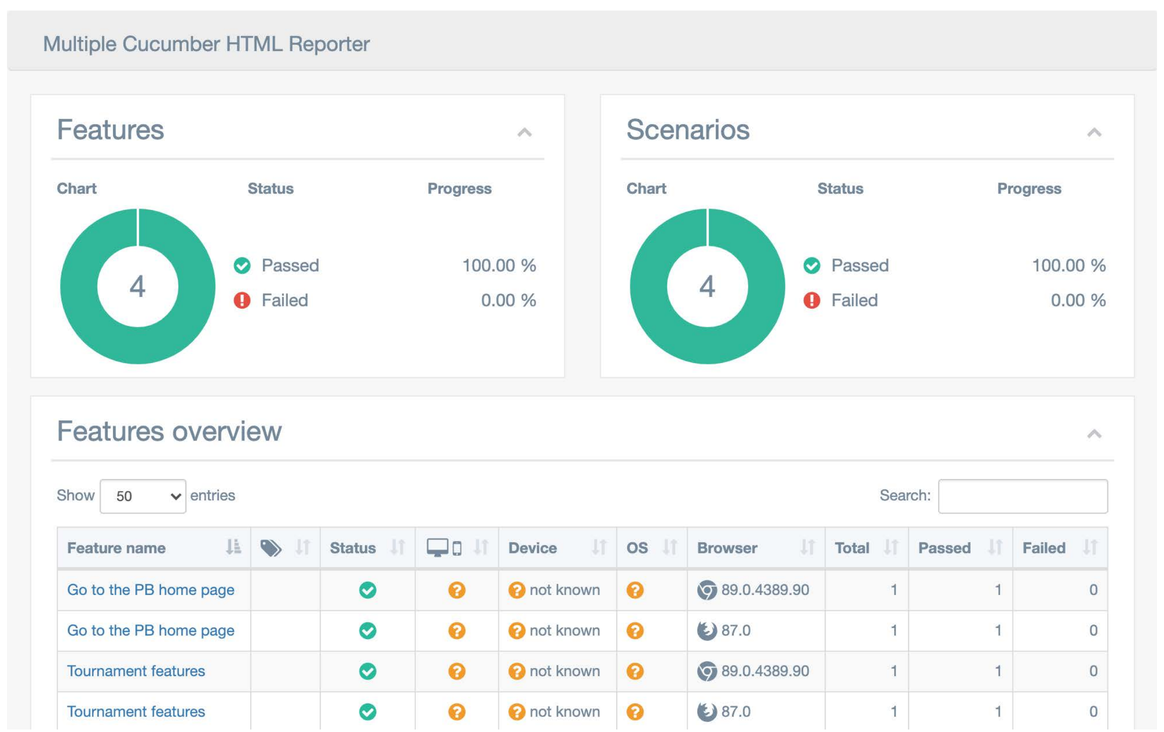


Figure 15.12: The Cucumber report in Jenkins

These provide insight into which cases were executed for what browser and report any failures that may have occurred. Let's switch gear a little now and take a look at non-functional testing.

Pipelines and Quality Gates (Non-functionals)

Quality is often just focused on whether tests pass or not. However there's also the concept of code quality. The code may perform as expected but the manner in which it's been written could be so poor that it could be a future source of problems when changes are added. So now it's time to check the quality of our code.

SonarQube

As part of the Ubiquitous Journey, we have automated the Helm chart deployment of SonarQube, which we are using to test and measure code quality. In `values-tooling.yaml`, the SonarQube stanza references the Helm chart and any extra plugins that are required. Many of the common language profile plugins are already deployed with the base version of SonarQube, for example, Java, JavaScript, and Typescript. We add in extra plugin entries for Checkstyle, our Java formatting check tool, and a dependency checker for detecting publicly disclosed vulnerabilities contained within project dependencies:

```
  # Sonarqube
  - name: sonarqube
    enabled: true
    source: https://github.com/redhat-cop/helm-charts.git
    source_path: "charts/sonarqube"
    source_ref: "sonarqube-0.0.14"
    sync_policy: *sync_policy_true
    destination: *ci_cd_ns
    values:
      initContainers: true
      plugins:
        install:
          - https://github.com/checkstyle/sonar-checkstyle/releases/
download/8.35/checkstyle-sonar-plugin-8.38.jar
          - https://github.com/dependency-check/dependency-check-sonar-
plugin/releases/download/2.0.7/sonar-dependency-check-plugin-2.0.7.jar
```

With the basic SonarQube pod deployed, there is one more piece of configuration we need to automate – the creation of a code quality gate. The quality gate is the hurdle our code must pass before it is deemed ready to release. This boils down to a set of conditions defined in code that specify particular measurements, for example:

- Do we have new blocking issues with the code that was just added?
- Is the code test coverage higher than a given percentage?
- Are there any identifiable code vulnerabilities?

SonarQube lets us define these quality gates[23] using its REST API. For PetBattle, we use a Kubernetes job to define our quality gate `AppDefault` and package it as a Helm chart for deployment. The chart is deployed using Ubiquitous Journey and ArgoCD.

Conditions

Conditions on New Code

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	A
Security Rating	is worse than	A

Figure 15.13: A SonarQube quality gate definition

The SonarQube server can be queried via a REST API, whether a recent report against a particular project has passed or failed this quality gate. We have configured a Tekton step and task in our pipelines to automatically check this each time we run a build.

Our PetBattle Java applications are configured using Maven to talk to our SonarQube server pod and generate the SonarQube formatted reports during each build, bake, and deploy. In the reusable `maven-pipeline.yaml`, we call the following target to generate these reports:

```
# code analysis step maven pipeline

    - name: code-analysis
      taskRef:
        name: maven
      params:
        - name: MAVEN_MIRROR_URL
          value: "$(params.MAVEN_MIRROR_URL)"
        - name: MAVEN_OPTS
          value: "$(params.MAVEN_OPTS)"
        - name: WORK_DIRECTORY
          value: "$(params.APPLICATION_NAME)/$(params.GIT_BRANCH)"
        - name: GOALS
          value:
            - install
            - org.owasp:dependency-check-maven:check
```

23 https://docs.sonarqube.org/latest/user-guide/quality-gates/

```
            - sonar:sonar
        - name: MAVEN_BUILD_OPTS
          value:
            - '-Dsonar.host.url=http://sonarqube-sonarqube:9000'
            - '-Dsonar.userHome=/tmp/sonar'

# code analysis step nodejs pipeline

    - name: code-analysis
      taskRef:
        name: nodejs
      params:
        - name: NPM_MIRROR_URL
          value: "$(params.NPM_MIRROR_URL)"
        - name: GOALS
          value:
            - "run"
            - "sonar"
```

Similarly, for the PetBattle UI using `nodejs`, we can configure the client to call SonarQube as part of its Tekton pipeline. Once these steps have successfully run, we can explore the SonarQube Web UI and drill down into any areas to find out more information.

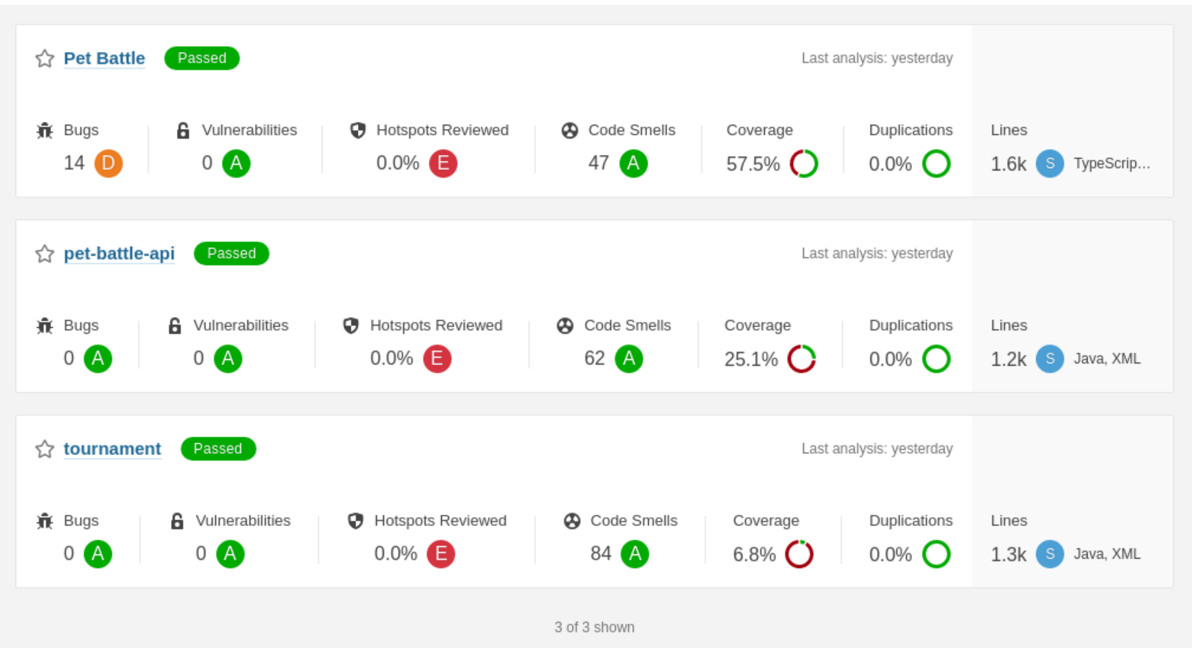


Figure 15.14: SonarQube project view

In a bit of recent development for A/B testing support in the PetBattle UI, some code bugs seemed to have crept in! Developers can drill down and see exactly what the issues are and remediate them in the code base. SonarQube ranks issues based on severity defined in the Language Quality Profile, which can be altered to suit your development code quality needs.

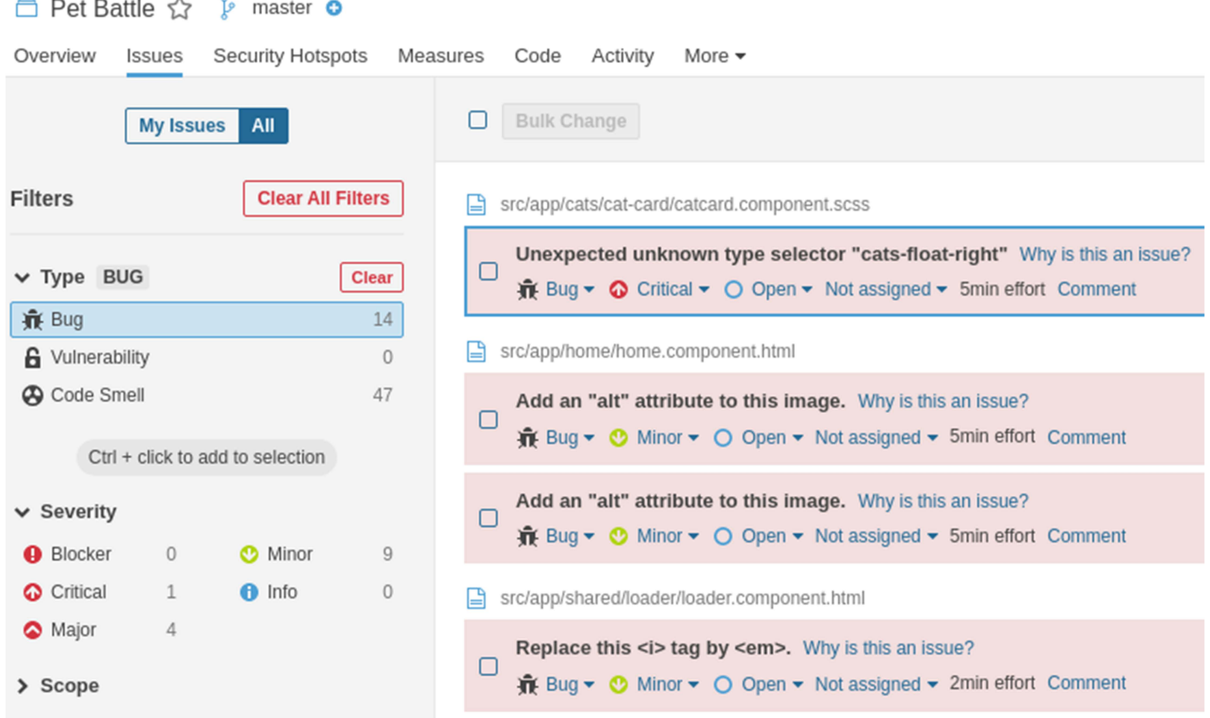


Figure 15.15: SonarQube drilling into some bug details

SonarQube also reports on the last run's code testing coverage. On the code base side, you generate coverage reports using the LCOV[24] format, so in Java, this is done by JaCoCo[25] and in JavaScript, the coverage reports are produced by the mocha/jasmine modules. These reports are uploaded into SonarQube and give the team visibility into which parts of their code base need more testing. A nice way to view this information is using the heatmap, which visualizes the bits of code that have near 100% coverage (green), down to areas that are not covered at all 0% (red). The statistics are also reported – the percentage coverage overall, the number of lines covered, and so on.

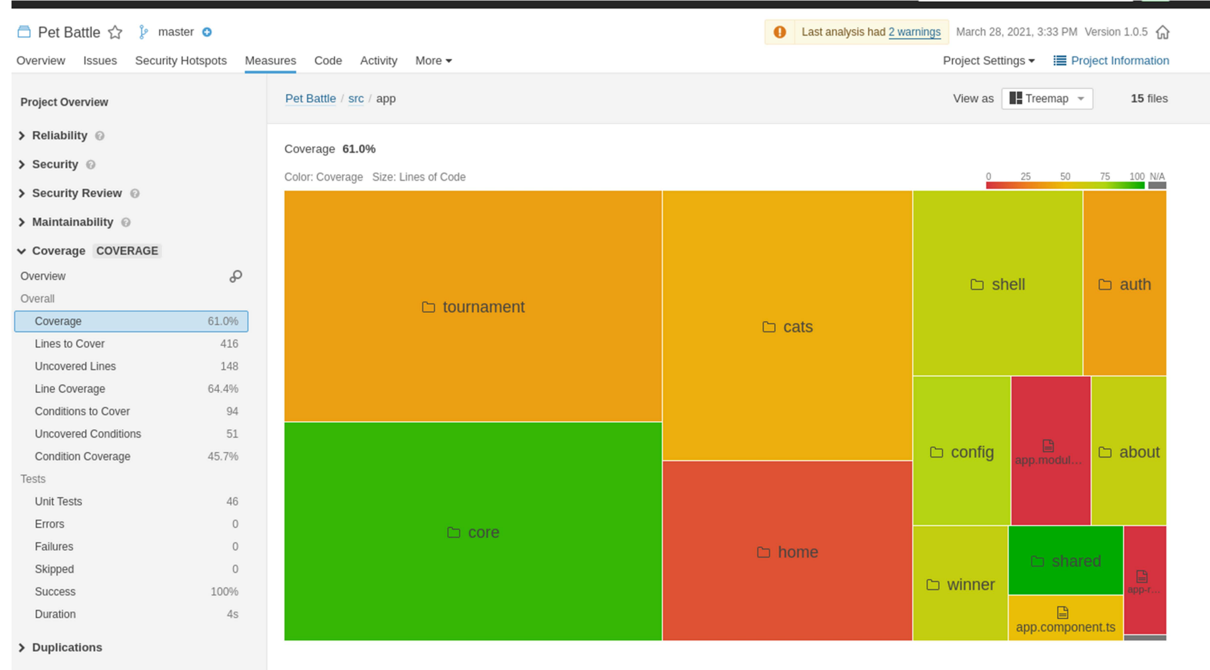


Figure 15.16: SonarQube test coverage heatmap for PetBattle

24 https://github.com/linux-test-project/lcov

25 https://www.eclemma.org/jacoco/

The last plugin we use for our Java applications is the OWASP Dependency-Check plugin.[26] We move security checking "left" in our pipeline. In other words, we want to discover early in the development process when security vulnerabilities or CVEs are creeping into our applications' dependencies. By identifying which dependencies are vulnerable to a CVE early as part of the build cycle, developers are in a much better position to update them, rather than finding there are issues once our applications are deployed.

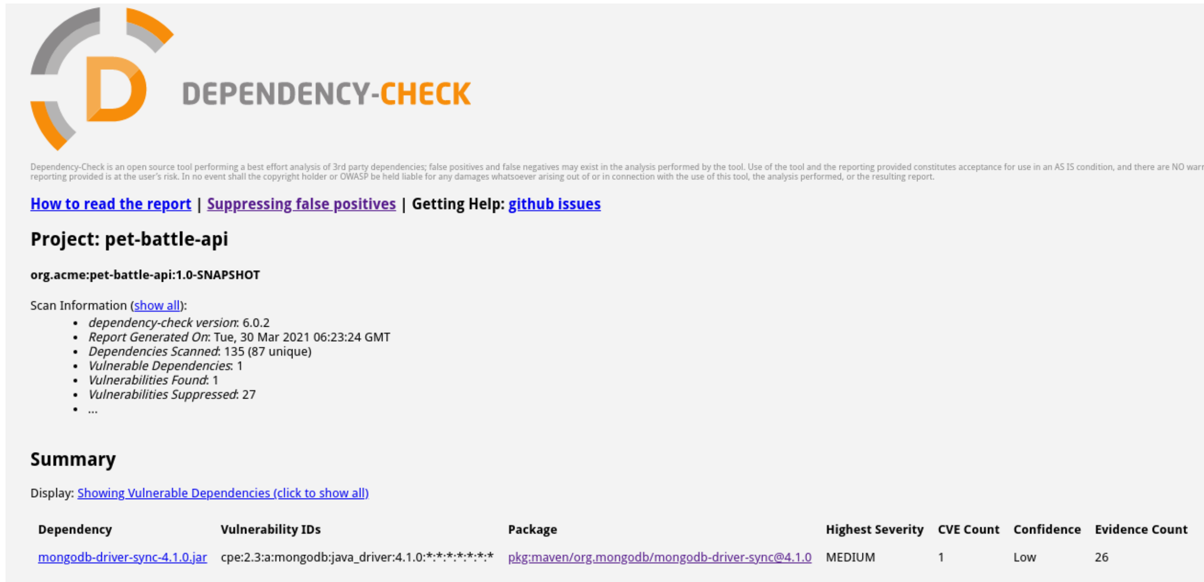


Figure 15.17: Dependency-Check plugin report

The plugin sources data from multiple open source resources including the US National Vulnerability Database[27] and Sonatype OSS Index.[28] In conjunction with security team members, developers can verify known vulnerabilities and suppress any false positives using a configuration file. The report is very detailed and includes links to these sites to assist CVE identification and reporting.

26 https://github.com/dependency-check/dependency-check-sonar-plugin

27 https://nvd.nist.gov/

28 https://ossindex.sonatype.org/

Perf Testing (Non-Functional)

One of our favorite command-line tools for getting fast feedback on the performance of REST APIs is a tool called **hey**.[29] There are a lot of similar tools available. Apache Bench[30] is probably the most venerable.

```
@4fc082b42236:~
virt:~$ hey -t 30 -c 10 -n 100 -H "Content-Type: application/json" -m GET
 https://pet-battle-api-labs-test.apps.hivec.sandbox1405.opentlc.com/cats

Summary:
  Total:        4.2048 secs
  Slowest:      0.6216 secs
  Fastest:      0.3155 secs
  Average:      0.4115 secs
  Requests/sec: 23.7823


Response time histogram:
  0.316 [1]     |■
  0.346 [2]     |■
  0.377 [12]    |■■■■■■■■
  0.407 [58]    |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.438 [17]    |■■■■■■■■■■■■
  0.469 [0]     |
  0.499 [0]     |
  0.530 [2]     |■
  0.560 [2]     |■
  0.591 [2]     |■
  0.622 [4]     |■■■


Latency distribution:
  10% in 0.3729 secs
  25% in 0.3888 secs
  50% in 0.3992 secs
  75% in 0.4122 secs
  90% in 0.5086 secs
  95% in 0.5713 secs
  99% in 0.6216 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0081 secs, 0.3155 secs, 0.6216 secs
  DNS-lookup:   0.0003 secs, 0.0000 secs, 0.0037 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0002 secs
  resp wait:    0.0631 secs, 0.0365 secs, 0.0906 secs
  resp read:    0.3402 secs, 0.2477 secs, 0.4536 secs

Status code distribution:
  [200] 100 responses
```

Figure 15.18: A simple hey run

29 https://github.com/rakyll/hey

30 https://httpd.apache.org/docs/2.4/programs/ab.html

We like **hey** because it is small, fast, written in Golang, and reports statistics in a format we can easily understand. In the preceding screenshot, we can see a very simple invocation using hey on the command line to call the PetBattle API and list all of the pets. We pass in some parameters that represent:

- `-c`: Number of workers to run concurrently
- `-n`: Number of requests to run
- `-t`: Timeout for each request in seconds

We can see the summary statistics reported, and this is the bit we love – a histogram of latency distribution, HTTP status code distribution, as well as DNS timing details. This is super rich information. Histograms are graphs that display the distribution of the continuous response latency data. A histogram reveals properties about the response times that the summary statistics cannot. In statistics, summary data is used to describe the complete dataset – minimum, maximum, mean, and average, for example. **Hey** gives us these summary statistics at the top of the output.

The graph brings the data to life as we can start to understand the distribution of the latency response over the time the test ran. Over the 4.2 seconds it took to send the 100 requests, we can see that most of the data is clustered around the 0.4-second mark, which is nearly 50% of all traffic. Often, in service performance design, we are interested in what the 95% or 99% percentile number is. That is, for all of the sample data, what the response latency is for 95% (or 99%) of the traffic. In this test run, it is measured at 0.57 seconds – in other words, 95% of the data was at or below this mark.

The shape of the histogram is also important. Where are the response latencies grouped? We can easily see if the response times are distributed evenly around the mean (Gaussian) or if they have a longer or shorter tail. This can help us characterize the performance of the service under various loads. There are many types of load profiles you could use, for example, burst loads where we throw a lot of instantaneous traffic at our API, compared to more long-lived soak tests under a lower load. You might even have known loads from similar applications in production already. A great open source tool for designing these types of test loads, which can model threading and ramping really well, is Apache JMeter[31] and we highly recommend it as a tool to have in your toolbox. To keep things simple, we won't cover that tool here.

31 https://jmeter.apache.org/

The two diagrams shown in *Figure 15.19* display simple load tests. The one on the left is a burst type of test – 300 consecutive users calling 900 times to our PetBattle API. We can see the 95% is 15.6 seconds – this is quite a long time for users to wait for their cats! The one on the right is a soak test – 50 consecutive users calling 10,000 times to our PetBattle API. A very different set of statistics: a test duration of 461 seconds, and the 95% is 2.8 sec—much better from an end user's perspective.

At this point, it is important to think about what the test is actually doing and how it relates to the PetBattle application suite in general. If we think about it, the test may not be totally indicative of the current user interface behavior. For example, we do not perform a call to return all of the images in our MongoDB at once but rather page the results. And there are of course other API endpoints to test, for example, the `topcats` API, which returns the top three most popular pets and is called every time you visit the home page. We are returning the test dataset we have loaded into PetBattle, that is, around 15 pet images, so it is not a massive amount of data. It's important to always step back and understand this wider context when we run performance tests so we don't end up testing the wrong thing!

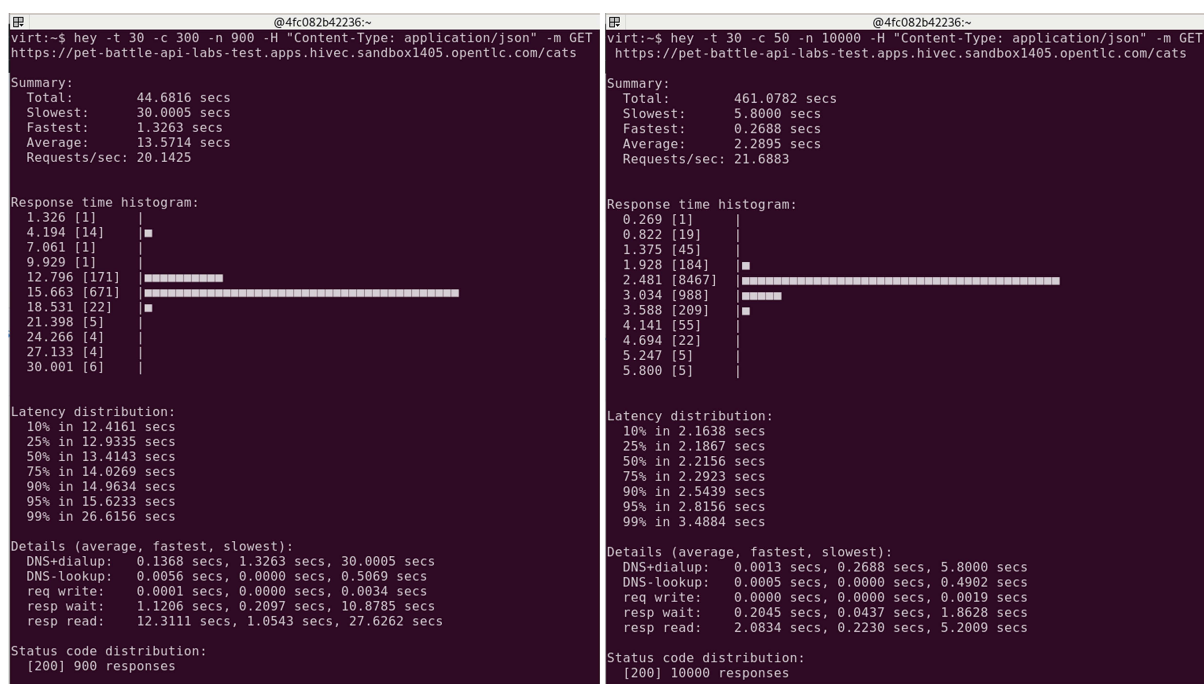


Figure 15.19: Burst and soak tests against the PetBattle API

Nonetheless, this is good data to ponder. A good result is that both the soak and burst tests only returned HTTP 200 response statuses – there were no error responses from the API. That gives us confidence that we have not broken anything or reached any internal system limits yet. We can also examine the details to make sure DNS resolution is not causing issues from the client-calling perspective.

Now we are familiar with the client or calling side of performance testing, let's switch to the PetBattle API application running on the server side. If we browse to the **Developer** view and select the `pet-battle-api` pod in the `labs-test` namespace, we can see some important server-side information:

- The PetBattle API is autoscaled to two pods.
- Monitoring metrics for the pods (check the appendix if you haven't enabled this for CRC).

As a developer, we have configured the PetBattle API application to use the **Horizontal Pod Autoscaler** (**HPA**). This specifies how the OpenShift Container Platform can automatically increase or decrease the scale of a replication controller or deployment configuration, the number of running pods, based on the metrics collected from the pods that belong to our application.

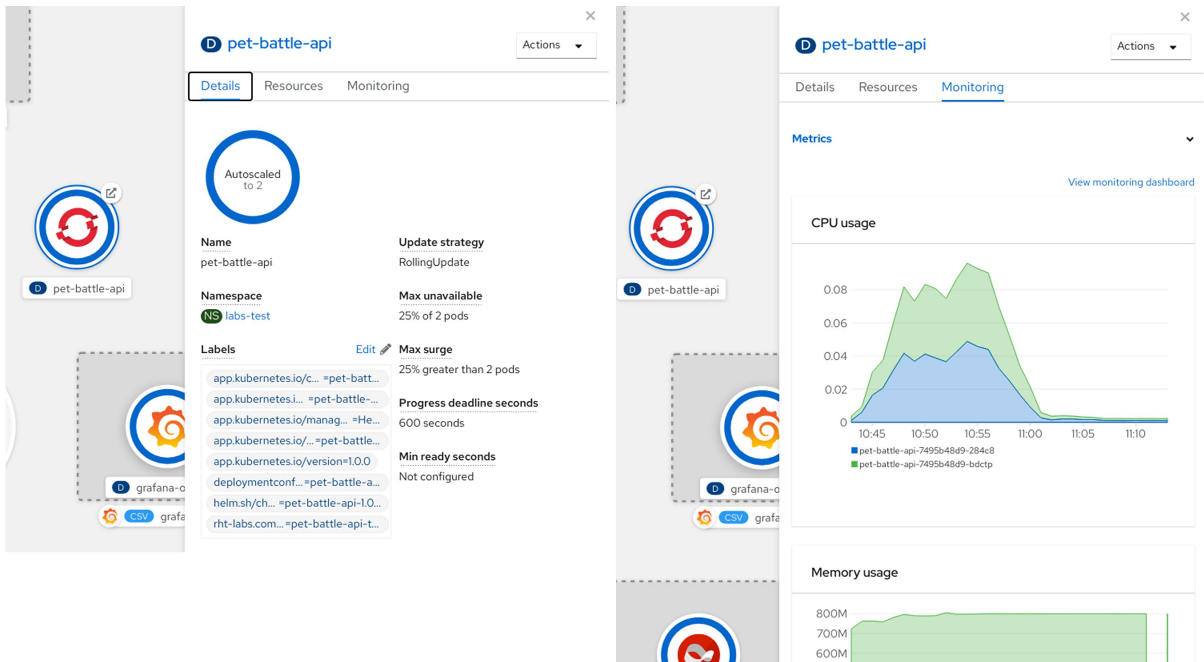


Figure 15.20: PetBattle API pod in the labs-test namespace

In our PetBattle API Helm chart, we specified the HPA with configurable values for minimum pods, maximum pods, as well as the average CPU and memory targets. Using hey, we can now test out various scenarios to help us tune the PetBattle API application under load:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: {{ include "pet-battle-api.fullname" . }}
  labels:
    {{- include "pet-battle-api.labels" . | nindent 4 }}
spec:
  scaleTargetRef:
    {{- if .Values.deploymentConfig }}
    apiVersion: v1
    kind: DeploymentConfig
    {{- else }}
    apiVersion: apps/v1
    kind: Deployment
    {{- end }}
    name: {{ include "pet-battle-api.fullname" . }}
  minReplicas: {{ .Values.replicas.min }}
  maxReplicas: {{ .Values.replicas.max }}
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: AverageValue
          averageValue: {{ .Values.hpa.cpuTarget }}
    - type: Resource
      resource:
        name: memory
        target:
          type: AverageValue
          averageValue: {{ .Values.hpa.memTarget }}
```

We initially took a rough guess at these settings in our HPA, for example, `min replicas = 2`, `max replicas =6`, `CPU = 200m`, `mem = 300Mi`, and set the resource limits and requests in our Deployment appropriately. We always have a minimum of two pods, for high availability reasons. The HPA is configured to scale based on the average memory and CPU loads. We don't yet understand whether the application is memory- or CPU-intensive, so choose to scale based on both these measurements.

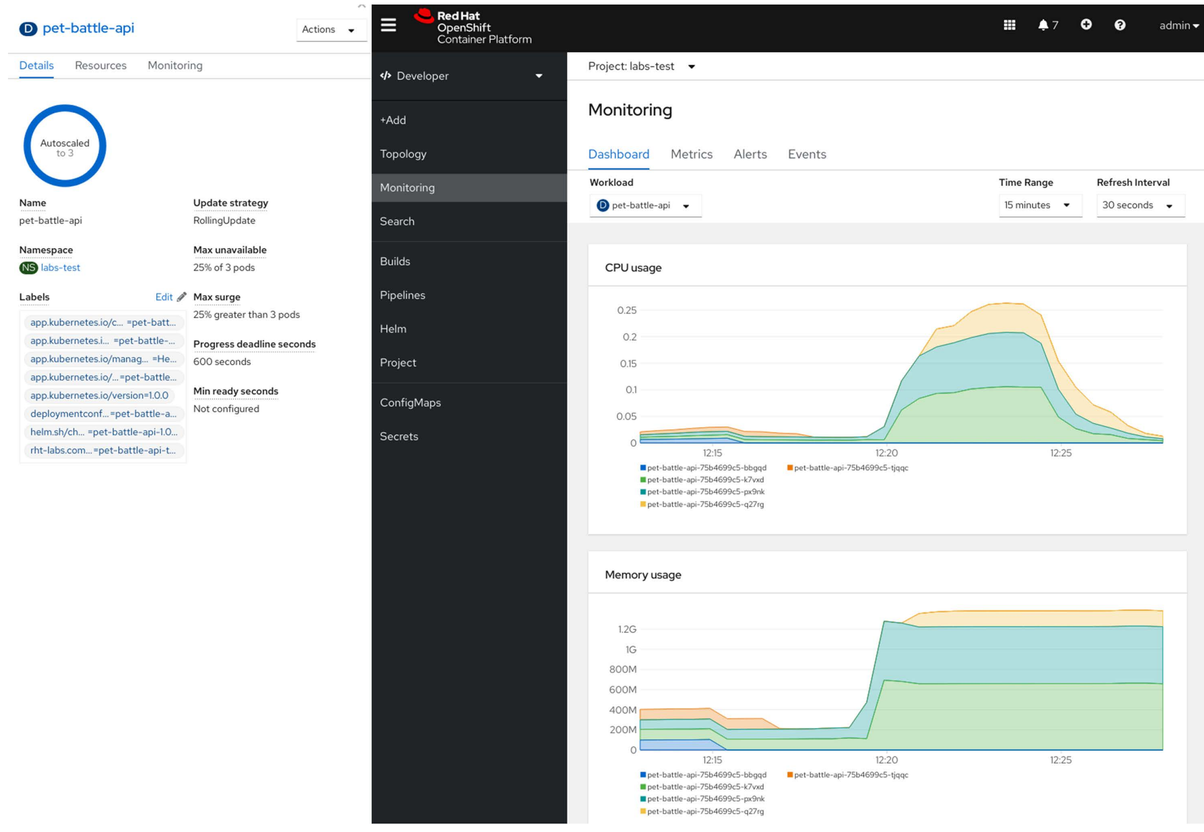


Figure 15.21: PetBattle API HPA in action, scaling pods under load

We use hey to start a burst workload, 400 concurrent requests, and watch the behavior of the HPA as it starts more pods to keep to the specified memory and CPU averages. Once the test concludes, the HPA scales our workload back down to the minimum as the application recovers resources, in this case through Java garbage collection. OpenShift supports custom metrics for the HPA as well as other types of pod scalers, for example, the Vertical Pod Autoscaler.[32]

To conclude this section, we want to point out one more Kubernetes object that the developer needs in their toolbelt – the **Pod Disruption Budget** (**PDB**). Again, using a Helm chart template for the PDB, we can limit the number of concurrent disruptions that the PetBattle API application experiences. By setting up a PDB, we can allow for higher availability while permitting the cluster administrator to manage the life cycle of the cluster nodes.

32 https://docs.openshift.com/container-platform/4.7/nodes/pods/nodes-pods-using.html

If the cluster is being updated and nodes are being restarted, we want a minimum of one `pet-battle-api` pod available at all times:

```
$ oc get pdb

NAME            MIN AVAILABLE  MAX UNAVAILABLE   ALLOWED DISRUPTIONS   AGE
pet-battle-api    1              N/A                  1                 46h
```

This ensures a high level of business service for our PetBattle API. We can see `ALLOWED_DISRUPTIONS` is set to 1 – this is because, at the time, the HPA had scaled the number of available replicas to 3 and this will change as the number of available pods changes.

One of the great things about performance testing applications on OpenShift is that all of the tools are at a developer's fingertips to be able to configure, test, measure, and tune their applications to achieve high availability and performance when under load. Each application service is independently scalable, tunable, and deployable, which makes for a faster and targeted feedback loop when dealing with scale and performance issues.

In the next section, we are going to take a look at what makes a good OpenShift Kubernetes citizen, automating Kubernetes resource validation as part of our pipeline.

Resource Validation

One aspect of testing that doesn't yet get much thought is the quality of the Kubernetes resources being deployed on the cluster. For applications to be considered *good citizens* on Kubernetes, there are a number of deployment best practices to be followed—including health checks, resource limits, labels, and so on—and we will go through a number of these in in *Chapter 16*, *Own It*. However, we need to validate the resource definitions being applied to the cluster to ensure a high level of compliance to not only industry recommendations but also any other resource recommendations that we see fit to add. This is where **Open Policy Agent** (**OPA**)[33] and associated tools can come into play. This enables us to validate resource definitions during a CI pipeline and also when applying resources to a cluster. OPA by itself is a policy validator and the policies are written using a language called Rego. Additional OPA tools such as Conftest[34] and Gatekeeper[35] add a lot of value and governance from a usability and deployment perspective. OPA is also embeddable into other third-party tools such as KubeLinter.[36]

33 https://www.openpolicyagent.org/

34 https://github.com/open-policy-agent/conftest

35 https://github.com/open-policy-agent/gatekeeper

36 https://github.com/stackrox/kube-linter

We haven't used OPA's server-side validation component, Gatekeeper,[37] as part of PetBattle but there are example Rego policies in the Red Hat Community of Practice GitHub repo[38] that are definitely worth exploring. If this is something of interest to you, definitely check out the blog on OpenShift.com that details setting up all of these components.[39]

However, to show how easy it is to use client-side resource validation and why you should include at least some resource validation in a pipeline, a simple Rego example has been created. Rego policies are easy enough to write and the Rego playground[40] is a great place to write and verify policies, so check it out.

Let's get into an example. In our Non-Functional Requirements Map, we said we wanted to be consistent with our labeling. It makes sense that we should adopt the Kubernetes best practice that suggests the `app.kubernetes.io/instance` label should be on all resources, so let's see how we can write a test to this effect and add it to our pipeline in Jenkins.

The makeup of a policy that denies the creation of a resource is simple enough. A message is formed and passed back to the interpreter if all of the statements are true in the rule. For example, we have written a policy that checks that all resources conform to Kubernetes best practice for naming conventions. The policy here is checking whether `app.kubernetes.io/instance` exists on the resource supplied to it (`input`). If each statement is true, then a message is returned as the error, guiding someone to fix the issue:

```
deny[msg] {
  label := "app.kubernetes.io/instance"
  not input.metadata.labels[label]
  msg := sprintf("\n%s: does not contain all the expected k8s labels
                 in 'metadata.labels'.\n Missing '%s'. \nSee: https://
kubernetes.io/docs/concepts/overview/working-with-objects/common-labels",
[input.kind, label])
}
```

37 https://github.com/open-policy-agent/gatekeeper

38 https://github.com/redhat-cop/rego-policies

39 https://www.openshift.com/blog/automate-your-security-practices-and-policies-on-openshift-with-open-policy-agent

40 https://play.openpolicyagent.org/

We can combine this rule with Conftest and a Helm template to create a way to statically validate our resources. In the PetBattle frontend code, there is a policy folder that has a few more policies to check whether all the standard Kubernetes labels[41] are set on our generated resources after we run the `helm template` command. By running a few commands, we can verify these are in place. First, we template our chart to produce the Kubernetes resources we will apply in deploying our software, and secondly, we tell Conftest to check each file generated against the rule:

```
# from the pet battle front end repository (https://github.com/petbattle/
pet-battle.git)
$ for file in $(ls policy/helm-output/pet-battle/templates/); do conftest
test policy/helm-output/pet-battle/templates/$file; done

6 tests, 6 passed, 0 warnings, 0 failures, 0 exceptions
FAIL - policy/helm-output/pet-battle/templates/deploymentconfig.yaml -
DeploymentConfig: does not contain all the expected k8s labels in 'metadata.
labels'.
 Missing 'app.kubernetes.io/name'.
See: https://kubernetes.io/docs/concepts/overview/working-with-objects/
common-labels

6 tests, 5 passed, 0 warnings, 1 failure,  0 exceptions
6 tests, 6 passed, 0 warnings, 0 failures, 0 exceptions
6 tests, 6 passed, 0 warnings, 0 failures, 0 exceptions
```

When executing the rules from the command line, we get a good insight into what's missing from our chart. Of course, we could just assume that we'd always make our charts adhere to the best practices, but the `jenkins-agent-helm` has also got the Conftest binary so we can execute the preceding statements in our Jenkins pipeline too. This example might seem simple but, hopefully, it gives you some idea of the things that can be automated and tested that might seem less obvious.

41 https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/#labels

Image Scanning

Red Hat provides the Quay Container Security Operator in OpenShift to bring Quay and Clair image scanning and vulnerability information into our OpenShift cluster. Any container image that is hosted on Quay.io is scanned by Clair.

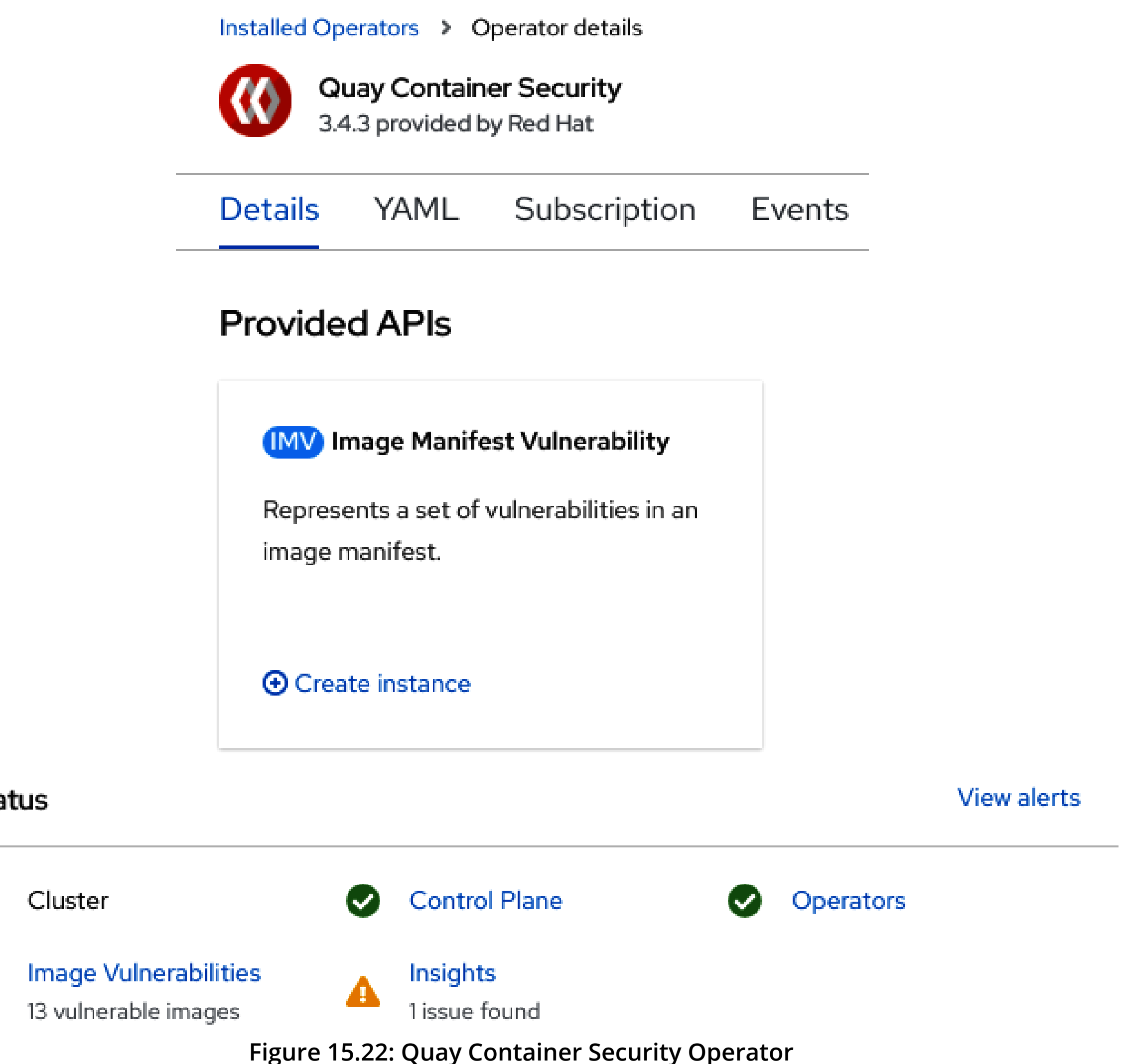


Figure 15.22: Quay Container Security Operator

Any image vulnerability data is exposed back in the OpenShift Web UI so that users and administrators can easily view which images are considered vulnerable and which namespace they are deployed to.

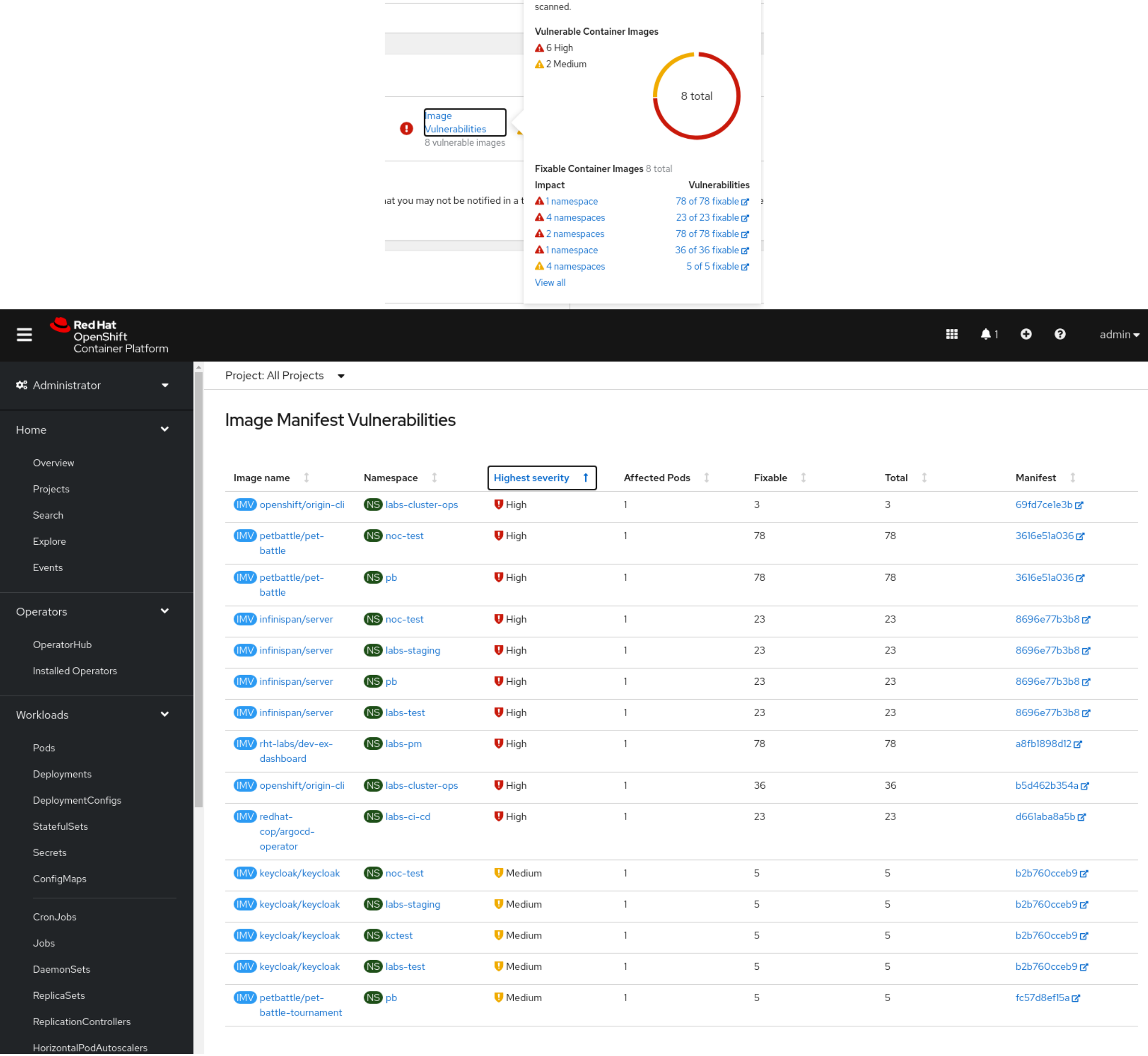


Figure 15.23: Vulnerable container images

With this operator deployed, the OpenShift overview status displays image vulnerability data, which an operator can drill into to find out the status of container images running on the platform. For PetBattle, we don't have any enforcement for image vulnerabilities discovered in our cluster. If we wanted to move the security scanner "left" in our deployment pipeline, there are some great open source scanning tools available on the OpenSCAP website.[42]

Other Non-functional Testing

There are lots of other types of testing we can do to validate our application. In this section are some of the things we feel are important to include in a pipeline, but the reality is there is much more than just this list and books could be written on this topic in and of itself!

Linting

A linter is a static code analysis tool that can check a code base for common pitfalls in design or stylistic errors. This does not check the compiled application, but the structure of the application. This is super important for languages that are not compiled, such as JavaScript. Browsers can interpret JavaScript in different ways so consistency is super critical.

If you think about a large enterprise application, there could be hundreds of developers working on the one code base. These developers could even be globally distributed with different teams looking after different parts of the application's life cycle. Having consistency in the approach to writing the software can dramatically improve maintenance costs. JavaScript is very flexible in how you can write it, whether this is from a functional programming standpoint or object-oriented, so it is important to get this consistency right.

The PetBattle frontend uses TSLint/ESLint[43] to check the style of the code adheres to a standard set of rules. These rules can be manipulated by the team, but the rules are checked into Git so if someone was to disable them or manipulate them, it would be noticed. Our Jenkins pipeline is configured to automatically check the code base using the `npm lint` command and our build will fail if a developer does not adhere to the standard.

42 https://www.open-scap.org

43 https://eslint.org/

```
pet-battle — donal@dspring-mac — ..le/pet-battle — -zsh — Solarized Dark ansi — 105×23
#( 04/01/21@11:14pm )( donal@dspring-mac ):~/Documents/Clients/petbattle/pet-battle@master✔
[   npm run lint                                                                                        ]

> pet-battle@1.3.1 lint /Users/donal/Documents/clients/petbattle/pet-battle
> ng lint && stylelint "src/**/*.scss" --syntax scss && htmlhint "src" --config .htmlhintrc

TSLint's support is discontinued and we're deprecating its support in Angular CLI.
To opt-in using the community driven ESLint builder, see: https://github.com/angular-eslint/angular-eslin
t#migrating-an-angular-cli-project-from-codelyzer-and-tslint.
Linting "pet-battle"...
All files pass linting.
TSLint's support is discontinued and we're deprecating its support in Angular CLI.
To opt-in using the community driven ESLint builder, see: https://github.com/angular-eslint/angular-eslin
t#migrating-an-angular-cli-project-from-codelyzer-and-tslint.
Linting "pet-battle-e2e"...
All files pass linting.
Browserslist: caniuse-lite is outdated. Please run next command `npm update`

   Config loaded: .htmlhintrc

Scanned 12 files, no errors found (39 ms).
#( 04/01/21@11:20pm )( donal@dspring-mac ):~/Documents/Clients/petbattle/pet-battle@master✔
```

Figure 15.24: Linting PetBattle's frontend locally scans both the JavaScript and HTML

For Java Quarkus apps, Checkstyle[44] is used to analyze the code base.

For Kubernetes resources, the aforementioned Open Policy Agent can assist, and Helm also has the `helm lint`[45] command to validate your charts.

Code Coverage

So, you've written a load of tests and you think things are going great – but how do you know your tests are any good and covering all parts of the code base? Allow me to introduce code coverage metrics! A code coverage reporter is a piece of software that runs alongside your unit test suites to see what lines of code are executed by the tests and how many times. Coverage reports can also highlight when if/else control flows within an application are not being tested. This insight can provide valuable feedback as to areas of a system that remain untested and ultimately reduce the number of bugs.

44 https://checkstyle.sourceforge.io/

45 https://helm.sh/docs/helm/helm_lint/

Our PetBattle frontend is configured to run a coverage report when our Jest tests execute. Jest makes generating the report very simple as it has a flag that can be passed to the test runner to collect the coverage for us. The coverage report is run on every execution of the build and so should be reported through Jenkins.

```
pet-battle — donal@dspring-mac — ..le/pet-battle — -zsh — Solarized Dark ansi — 105×33
  home.component.ts                |   21.05 |        0 |    9.09 |   14.29 | 21-80
 app/shared                        |     100 |      100 |     100 |     100 |
  index.ts                         |     100 |      100 |     100 |     100 |
  shared.module.ts                 |     100 |      100 |     100 |     100 |
 app/shared/loader                 |     100 |      100 |     100 |     100 |
  loader.component.html            |     100 |      100 |     100 |     100 |
  loader.component.ts              |     100 |      100 |     100 |     100 |
 app/shell                         |     100 |      100 |     100 |     100 |
  shell.component.html             |     100 |      100 |     100 |     100 |
  shell.component.ts               |     100 |      100 |     100 |     100 |
  shell.service.ts                 |     100 |      100 |     100 |     100 |
 app/shell/header                  |   76.47 |       25 |      60 |   71.43 |
  header.component.html            |     100 |      100 |     100 |     100 |
  header.component.ts              |      75 |       25 |      60 |   69.23 | 25-32
 app/tournament                    |   45.45 |        0 |   16.67 |   41.25 |
  cat404.component.ts              |     100 |      100 |     100 |     100 |
  tournament.component.html        |     100 |      100 |     100 |     100 |
  tournament.component.ts          |   37.74 |        0 |   13.04 |      36 | 34,38-39,43-85,93-111
  tournament.service.ts            |   48.28 |      100 |   16.67 |   42.31 | 21-27,41-43,54-101
 app/winner                        |     100 |      100 |     100 |     100 |
  winner.component.html            |     100 |      100 |     100 |     100 |
  winner.component.ts              |     100 |      100 |     100 |     100 |
 environments                      |     100 |      100 |     100 |     100 |
  .env.ts                          |     100 |      100 |     100 |     100 |
  environment.ts                   |     100 |      100 |     100 |     100 |
-----------------------------------|---------|----------|---------|---------|-----------------------

Test Suites: 19 passed, 19 total
Tests:       46 passed, 46 total
Snapshots:   0 total
Time:        31.624 s
#( 04/01/21@11:14pm )( donal@dspring-mac ):~/Documents/Clients/petbattle/pet-battle@master✔
```

Figure 15.25: Code coverage report from the frontend unit tests locally

When executing our tests in the Jenkins pipeline, we have configured Jest to produce an HTML report that can be reported by Jenkins on the **jobs** page. For any build execution, the report is added to the **jobs** home page. The report will allow us to discover what lines are being missed by our tests. Being able to drill into a report like this can give a good insight into where our testing is lacking.

Back to master index Zip

All files / app/cats/cat-card catcard.component.ts

58.82% Statements 20/34 **5.26%** Branches 1/19 **50%** Functions 3/6 **56.25%** Lines 18/32

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1  3x import { Component, OnInit, Input } from '@angular/core';
2  3x import { CatsService } from '../cats.service';
3  3x import { ConfigurationLoader } from '@app/config/configuration-loader.service';
4  3x import { MatomoTracker } from 'ngx-matomo-v9';
5
6     @Component({
7       selector: 'app-catcard',
8       templateUrl: './catcard.component.html',
9       styleUrls: ['./catcard.component.scss']
10    })
11 3x export class CatcardComponent implements OnInit {
12      public image: string;
13 1x   public currentVoteCount = 0;
14 1x   public spinner = true;
15 1x   public compOver = false;
16      private catId: string;private analytics = false;
17 1x
18      constructor(
19        private catsService: CatsService,
20 1x     private configSvc: ConfigurationLoader,
21 1x     private matomoTracker: MatomoTracker
22 1x   ) {}
23
24      getNewCat(vote: boolean) {
25        if (this.analytics) {
26 1x       // instrument the buttons
27 1x       const voting = vote ? 'UP_VOTE' : 'DOWN_VOTE';
28 1x       this.matomoTracker.trackEvent('A/B Tests', 'Voting', voting);
29 1x     }
30 1x     this.spinner = true;
31 1x     const body = { image: this.image, id: this.catId, count: this.currentVoteCount, vote };
32       vote ? this.currentVoteCount++ : this.currentVoteCount--;
```

Figure 15.26: Code coverage report in Jenkins gives us detailed insight

So, what should I do with these results? Historically, we have worked where coverage is low. It can serve as a great talking point to bring up in a Retrospective. Printing out the reports and discussing them as a team is a great way to assess why the team is struggling to write enough tests. Sometimes teams are drowning by being overwhelmed with pressure to churn out features and so testing can slip to the wayside. Having a coverage reporter in your build can help keep a team honest. You could even set thresholds so that if testing coverage falls below a certain percentage (some teams aim for 80% and above), the build will fail, thus blocking the pipeline until the quality is increased.

Untested Software Watermark

I worked on a project a long time ago that was poorly structured. I was a member of the DevOps team, which I know now is an antipattern in most implementations! This project had many issues, the team had planned three Sprints in advance but hadn't allocated enough time for testing. It was always the thing that got squeezed.

Through Retrospectives with the teams, we discovered that there was simply not enough time for tests. This may sound hard to hear, but the root cause for this was not laziness by the team or a lack of skills; it really was time. The project was running in 8-week blocks that were pre-planned from the beginning with a fixed output at the end. The team thought they were doing Scrum, but in actual fact, they had milestones of functionality to accomplish each sprint and there was no feedback loop. Of course, none of the Scrum team members were involved in the sizing or planning ceremonies either. This meant the teams were constantly under pressure to deliver.

Through a Retrospective, we decided to try to radiate some of this pressure the teams were under as we were not happy that quality was being sacrificed for some arbitrary deadlines. Knowing that failing the pipeline simply would not work for these customers, we had to get creative in showing the software quality. We decided to inject a watermark into any application that had low test coverage. This watermark resembled a DRAFT logo you would find on any document, but ours was a little different.

A large banner reading UNTESTED SOFTWARE was placed across the applications that failed the tests. This watermark did not affect the user behavior of the app; it was just an overlay but it was an amazing way to get people talking. Seeing a giant banner saying UNTESTED is a surefire way to have people question why things have gotten this way.

Let's look at some other ways we can visualize risks during continuous delivery.

The OWASP Zed Attack Proxy (ZAP)

Security scanning is always a hot topic. From image scanning, which we discussed earlier, to dependency checking for our application that happens in our pipelines, there are limitless numbers of things to automate from a security perspective. Let's take another example of something that can be useful to include in a pipeline – the OWASP Zed Attack Proxy.[46]

From their website: *The OWASP Zed Attack Proxy (ZAP) is one of the world's most popular free security tools which lets you automatically find security vulnerabilities in your applications. This allows the developers to automate penetration testing and security regression testing of the application in the CI/CD pipeline.*

Adding the ZAP security scanning tool to our pipelines is simple. Just add the following `stage` and add the URL you want to test. The source code for this image is available, like our other Jenkins images from the Red Hat CoP.[47] The ZAP scan in Jenkins will produce a report showing some potential vulnerabilities in our application.

```
stage('🐯 OWASP Scan') {
  agent { label "jenkins-agent-zap" }
  steps {
    sh '''
      /zap/zap-baseline.py -r index.html -t http://<some website url> ||
return_code=$?
      echo "exit value was  - " $return_code
    '''
  }
  post {
    always {
      // publish html
      publishHTML target: [
          allowMissing: false,
          alwaysLinkToLastBuild: false,
          keepAll: true,
          reportDir: '/zap/wrk',
          reportFiles: 'index.html',
          reportName: 'OWASP Zed Attack Proxy'
        ]
    }
  }
}
```

46 https://www.zaproxy.org/

47 https://github.com/redhat-cop/containers-quickstarts/tree/master/jenkins-agents

In doing so, the web report that's created can be viewed in Jenkins, which gives great details on the cause of the security vulnerability as well as any action that should be taken to remedy it.

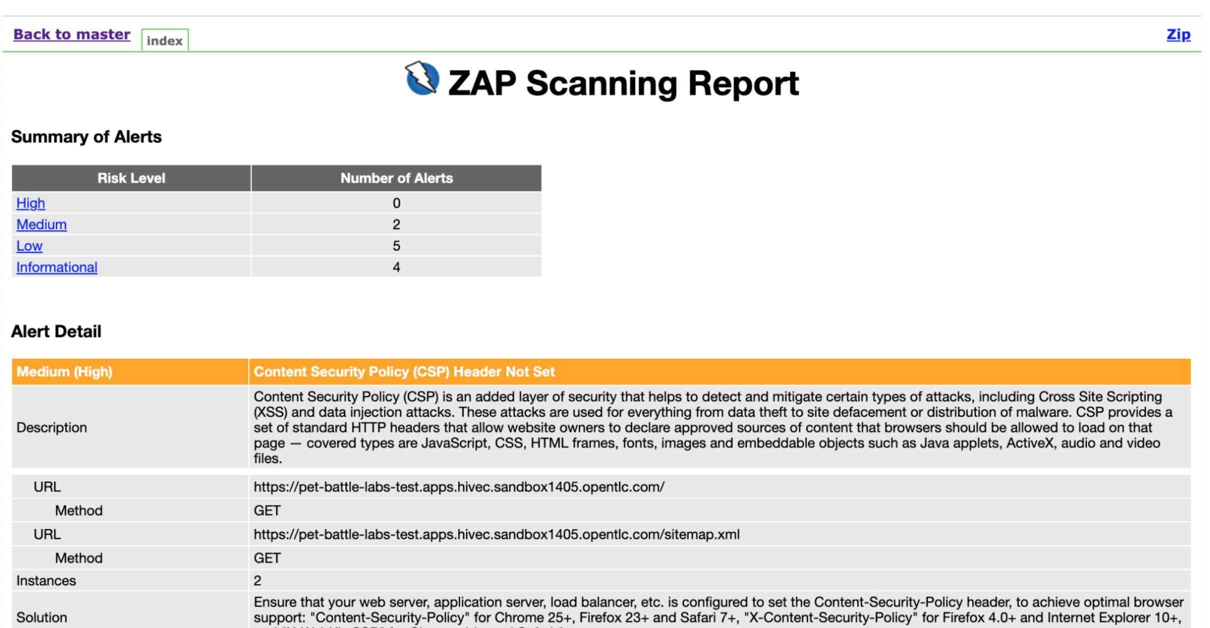


Figure 15.27: Example Zap report for PetBattle

In the final non-functional testing section, let's have a look at deliberately breaking our code using a technique called chaos engineering.

Chaos Engineering

Chaos engineering is the process of deliberately breaking, hobbling, or impacting a system to see how it performs and whether it recovers in the ensuing "chaos." While most testing is seen as an endeavor to understand how a system performs in a known, stable state, chaos engineering is the computing equivalent of setting a bull free in a fine-china shop—you know it's going to end badly but you just don't know exactly the magnitude of how bad it's going to be.

The purpose of chaos engineering is to build confidence in the resiliency of the system. It also allows you to better understand where breakage points occur and the blast radius of any failures. There are many resilience features built into the Kubernetes API specification. Pod replicas are probably the simplest mechanism, having more than one of your applications running at any given time. It is also desirable to use application-specific mechanisms such as circuit breakers, which prevent failures from spreading throughout your system. Chaos engineering takes these ideas one step further and tests a system when one or more components fully or partially fail, such as when CPU or memory resources are low.

The basic premise is that the system under test is observed in a stable working state, then a fault is injected. The system is then observed to see if it recovers successfully from the fault or not. Outcomes from such testing are a potential list of areas to tune/fix as well as an understanding of the **Mean Time to Recovery** (**MTTR**) of a system. It's important to note that chaos engineering is focused on the system as a whole—both application and infrastructure performance need to be considered and tested.

One of the key mantras behind chaos engineering is contained in its defining principles[48] – *The need to identify weaknesses before they manifest in system-wide, aberrant behaviors.*

This is one of the most important aspects to be considered when adopting this approach. You don't want to be learning about weaknesses during a production-impacting incident. It's similar to the rationale behind regularly testing disaster recovery plans. To paraphrase, a colleague of ours here at Red Hat said, "*When the excrement hits the fan, the first thing to do is turn off the fan!*" Not much time for learning there.

There are a number of tools and frameworks that can help with setting up a chaos engineering practice. Here's some to get started with (though there are others):

- Litmus Chaos[49]
- Kraken[50]
- Chaos Mesh[51]

48 https://principlesofchaos.org/

49 https://litmuschaos.io/

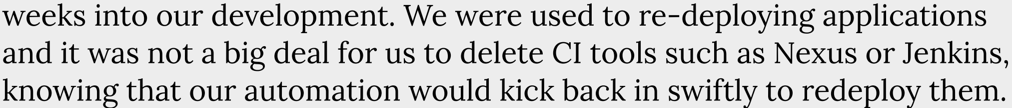
50 https://github.com/cloud-bulldozer/kraken

51 https://chaos-mesh.org/

In a world where practices such as everything-as-code and GitOps are our only way to build software and the systems that support them, a great way to validate the ability to respond to missing items is to redeploy everything, including your infrastructure, from scratch every week or every night! This might seem extreme, but it's a great way to validate that there is no hidden magic that someone has forgotten to write down or codify.

Accidental Chaos Testing

This is a story that I used to be reluctant to share, but over the years (and having done it twice), I realized it was actually a good thing to have done.

While working for an airline, I accidentally deleted the `labs-ci-cd` project along with a few other namespaces where our apps were deployed, including the authentication provider for our cluster. At the time, we were several  weeks into our development. We were used to re-deploying applications and it was not a big deal for us to delete CI tools such as Nexus or Jenkins, knowing that our automation would kick back in swiftly to redeploy them.

However, on this engagement, we were also using GitLab and, unfortunately for me, GitLab was in the same project as these other tools!

I checked with the team first and asked whether it was OK to rebuild everything in our tooling namespace. I got a resounding "yes" from my teammates, so proceeded to delete some of the things I thought needed to be cleared out and accidentally removed a few extra projects. About 30 seconds later, someone on the team perked up and asked, *Is Git down for anyone else*? This was promptly followed by another person saying, *Is anyone else not able to log in to the cluster*? My face lit up red as I immediately realized what I'd just done. Git, as we keep saying in the book, is our single source of truth. *If it's not in Git, it's not real* is our mantra! We even had it written on the walls! But I had just deleted it.

So, what happens when some silly person accidentally deletes it? After the initial shock and panic, the team pulled the Andon Cord. We quickly stormed together to see what exactly had happened in order to plan how we could recover not just Git but all the things we'd added to the cluster. Luckily for us, everything we had done was stored in Git so we were able to redeploy our tools and push our local, distributed copies of the software and infrastructure back into the shared Git repository.

The team was cross-functional and had all the tools and access we needed to be able to respond to this. Within 1 hour, we had fully restored all our applications and tools with all of our automation running smoothly again.

I think the real power in this example is how, given the right equipment and the right ownership, an empowered team can have it all. We acted as one unit fixing things at lightning speed. We were not stuck waiting in a queue or having to raise a ticket on another team to restore our infrastructure. We could do it for ourselves within minutes – not days or weeks later.

Another thing I learned was not to keep Git in the same project as the other tools in case another person like me comes along. I also learned to be mindful of the permissions we have within a cluster. As an administrator, I was able to remove things that perhaps I should not have been playing with.

So we've written the code, tested, quality-checked it and even scanned it for vulnerabilities. Now it's time to deploy it onto the cluster. Let's explore one of the key areas of benefit of using Kubernetes - the different ways you can deploy applications depending on your needs and perform user-driven experiments to determine what features your users prefer.

Advanced Deployments

The time between software being written and tested till it is deployed in production should be as short as possible. That way your organization is able to realize value from the software changes as quickly as possible. The modern approach to this problem is, of course, through automation. There are simply too many details and configuration items that need to be changed when deploying to production that even for a small application suite like PetBattle, manual deployment becomes error-prone and tedious. This drive to reduce manual toil is at the heart of many of the DevOps practices we have been discovering in this book.

We can minimize the downtime (ideally to zero!) during software deployment changes by adopting the right application architecture and combining that with the many platform capabilities that OpenShift offers. Let's look at some common deployment strategies that OpenShift supports:

- Rolling deployment:
    - Spin up a pod of the new version and then spin down a pod of the existing old version automatically. Very useful for a zero-downtime approach.
- Canary deployment:
    - Spin up a single pod of the new version, perform testing to ensure that everything is working correctly, and then replace all the old pods with new ones.
- Blue/Green deployment:
    - Create a parallel deployment and verify that everything is working correctly before switching traffic over.
    - Service Mesh traffic mirroring functionality can be useful with this approach to validate that the new version is working as expected.
- Recreate deployment:
    - Basically, scale the existing pods down to zero and then spin up the new version.
    - Use where an application must be restarted, for example, to migrate database schema or tables.
    - Think of this as a Ripley deployment: "take off and nuke the entire site from orbit. It's the only way to be sure."[52]

We can roll back to previous deployment versions using the Helm chart life cycle or the out-of-the-box `oc rollback` support. Images and configuration are versioned and cached in OpenShift to easily support rolling back to previous versions.

52 https://en.wikiquote.org/wiki/Aliens_(film)

A/B Testing

A/B testing an application is an amazing way to test or validate a new feature in production. The process is pretty simple: you deploy two (or more) different versions of your application to production, measure some aspect, and see which version performs *better*. Given that A/B testing is primarily a mechanism of gauging user experience, *better* depends on what aspect/feature you're experimenting with. For example, you could make a subtle change to a web page layout and measure how long it takes for the user to navigate to some button or how long the user continues to interact with specific items on the page.

It's a brilliant way to de-risk a new release or validate some new business or UI features with a smaller audience before releasing to a wider group. User behavior can be captured and experiments can be run to make informed decisions about what direction a product should take.

The Experiment

Let's cast our minds back to the earlier chapters where we spoke about generating options. There we spoke about the importance of experiments and our Value Slicing board included an item for which we could do an A/B test. One experiment that came up was to assess how users would vote for cats in the competition. Should they just be able to upvote (with a 👍) or should they be able to downvote (👎) too? We can build and deploy two versions of our application: one with the ability to both upvote and downvote, and one with just the ability to upvote. Our experiment is simple: to track how often people actually use the downvote button, so we can decide whether it's a feature we need or whether we should focus on building different functionality.

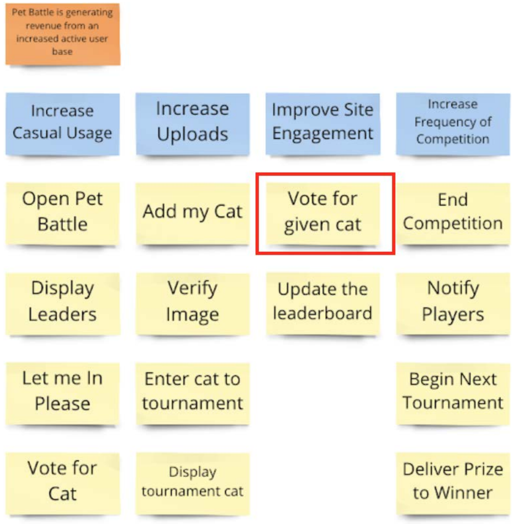


Figure 15.28: Experiment defined on a Value Slicing board

Let's now look at how we could set up a simple experiment to deploy both variants of the application and route traffic between each deployed instance to generate some data to help inform our decision-making.

Matomo – Open Source Analytics

OpenShift provides us with a mechanism to push traffic to different versions of an application. This in itself is useful, but it provides no information that we can base a decision on. For this, we need to measure how the users interact with the platform. To do this, we're going to introduce user analytics, which records metrics on the users' interactions with the website. We're going to use the open source Matomo[53] platform. There are others we could have used but, at the time of writing, this was our choice as it was open source and quite feature-complete. Let's add Matomo to our Big Picture for consistency.

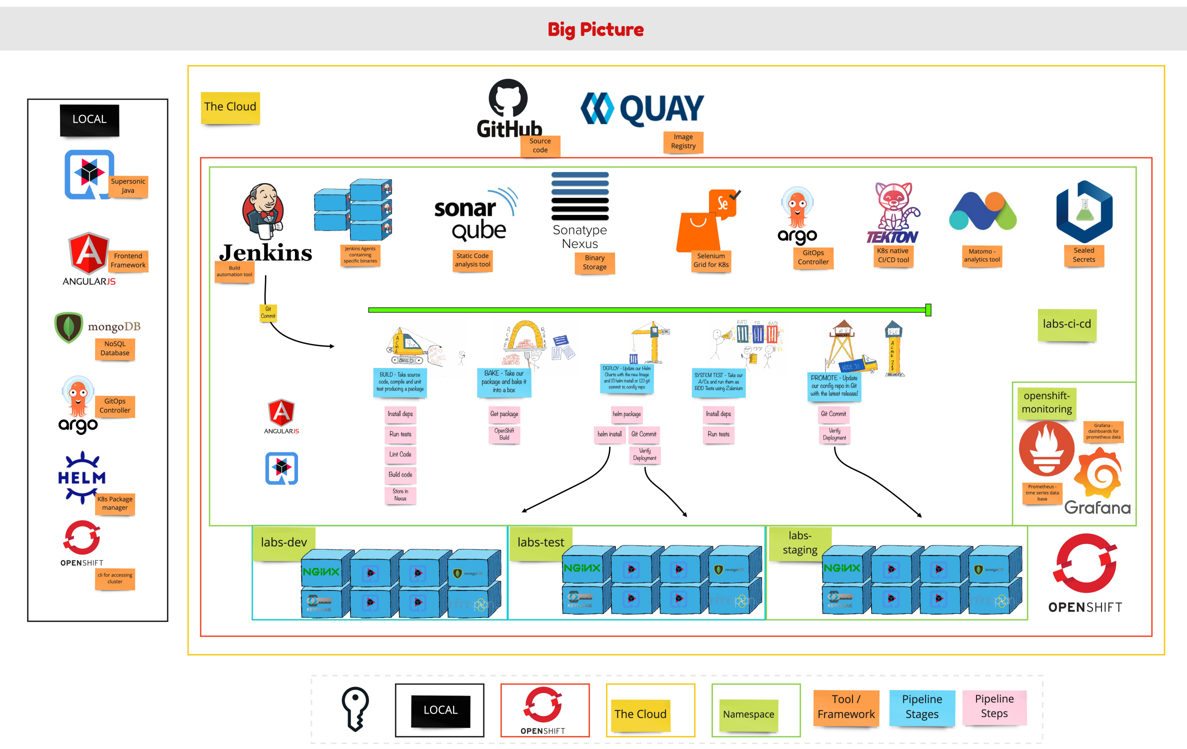


Figure 15.29: Big Picture with added tools including Matomo

53 https://matomo.org/

So how do we install the Matomo platform? Here comes Helm to the rescue again. We automated this installation as part of the PetBattle platform by just enabling it in our Ubiquitous Journey project. It's deployed by default into our `labs-ci-cd` namespace from this configuration in `ubiquitous-journey/values-tooling.yaml`:

```
# Matamo
  - name: matomo
    enabled: true
    source: https://github.com/petbattle/pet-battle-analytics.git
    source_path: charts/matomo
    sync_policy: *sync_policy_true
    destination: labs-ci-cd
    source_ref: main
    ignore_differences:
    - group: apps
      kind: Deployment
      jsonPointers:
        - /spec/replicas
        - /spec/template/spec/containers/0/image
```

However, if you want to just install the tool without involving ArgoCD, you can just clone the repository and install it manually. This chart has been forked from an existing chart[54] to tweak it for easier installation on OpenShift. Specifically, the security contexts in the MariaDB and Redis dependencies have been disabled so that the deployment will automatically use the target namespace default service account and associated **Security Context Constraint** (**SCC**) in OpenShift. For COTS software where repackaging or running as a random UID is not always possible, there are other more permissive, less secure SCCs such as `anyuid`. Also, an OpenShift route has been added to the chart to allow ingress traffic to the application:

```
$ oc login ...
$ git clone https://github.com/petbattle/pet-battle-analytics.git \
&& cd pet-battle-analytics
$ helm install pba charts/matomo
```

With the Matomo analytics deployed, we just need to configure the frontend to connect to it. To do this just update the config map's `matomoUrl` in the `chart/values.yaml` in the frontend to have the tracking code automatically track the site. This will provide basic site tracking such as the time spent on a page or the number of pages visited.

54 https://gitlab.com/ideaplexus/helm/matomo

```
23  # custom end point injected by config map. This is likely to changed
24  config_map: '{
25    "catsUrl": "https://pet-battle-api-labs-test.apps.hivec.sandbox1405.opentlc.com",
26    "tournamentsUrl": "https://pet-battle-tournament-labs-test.apps.hivec.sandbox1405.opentlc.com",
27    "matomoUrl": "https://matomo-labs-ci-cd.apps.hivec.sandbox1405.opentlc.com/",
28    "keycloak": {
29      "url": "https://keycloak-labs-test.apps.hivec.sandbox1405.opentlc.com/auth/",
30      "realm": "pbrealm",
31      "clientId": "pbclient",
32      "redirectUri": "http://localhost:4200/tournament",
33      "enableLogging": true
34    }       You, 3 months ago • ✷ Config load correction and sso stuff oh and co…
35  }'
36
```

Figure 15.30: Configuring the config_map for matomoUrl

For a more meaningful test, we might want to capture specific user behavior. The application has been instrumented to report certain events back to the Matomo server, such as mouse clicks. Whenever a user clicks the button to vote for a cat, it will capture it and report it in Matomo for us. It's very simple to do this – we just add a one-liner to the event we want to track:

```
this.matomoTracker.trackEvent('A/B Tests', 'Voting', voting)
```

Deploying the A/B Test

In PetBattle land, let's see how we could configure the deployments of the frontend to run this simple A/B test. Luckily for us, OpenShift makes this super easy by having a way to expose a `route` and connect it to more than one service, using the `alternateBackends` array to configure additional services to send traffic to. We can then apply `weights` to each service defined here in order to set the percentage of the traffic to either service that's deployed, A or B. The weights can be set between 0 and 256, and if a service is reduced to 0 then it carries on serving existing connections but no new ones. In fact, OpenShift allows us to do more than just an A or B test – also C and D, as `alternateBackends` supports up to three services!

Let's deploy our A/B experiment for `pet-battle`. We could integrate these steps with ArgoCD but to keep things nice and easy for illustrative purposes, let's just stick with using our trusty friend Helm to deploy things. We've prebuilt an image that has no ability to downvote on the home page `quay.io/petbattle/pet-battle:no-down-vote`. Let's deploy this image to our cluster by running a simple Helm command (make sure to set the config map to the correct endpoints for your cluster):

```
$ git clone git@github.com:petbattle/pet-battle.git && cd pet-battle

$ helm install nodownvote --set image_version=no-down-vote \
--set route=false chart --namespace petbattle
```

With this command, we're deploying a new instance of the `pet-battle` frontend by setting the image to the prebuilt one and disabling the OpenShift route for this as it's not needed. We'll configure our route to production by updating our `prod` app.

Running `oc get pods` should show the app started and if you check for routes, you should see none exposed:

```
$ oc get pods

NAME                            READY   STATUS      RESTARTS   AGE
nodownvote-pet-battle-1-deploy  0/1     Completed   0          2m47s
nodownvote-pet-battle-1-jxzhf   1/1     Running     0          2m43s
```

Let's deploy our `prod` version of the `pet-battle` application and add the `no-down-vote` app as one of the services we'll connect to. Our Helm chart is configured to accept the name of the service and the weight we want to apply to the experiment feature via `a_b_deploy.svc_name` and `a_b_deploy.weight`. It's defaulted to be a 50/50 round-robin split. Let's deploy it with this setup:

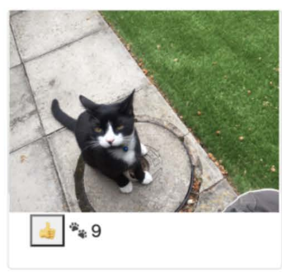
```
# install prod version
$ helm install prod --set image_version=latest chart \
--set a_b_deploy.svc_name=no-down-vote-pet-battle --namespace petbattle

# list pods
$ oc get pods

NAME                            READY   STATUS              RESTARTS   AGE
nodownvote-pet-battle-1-deploy  0/1     Completed           0          4m53s
nodownvote-pet-battle-1-jxzhf   1/1     Running             0          4m49s
prod-pet-battle-1-6bbv8         0/1     ContainerCreating   0          12s
prod-pet-battle-1-deploy        1/1     Running             0          16s
```

Navigate to the `pet-battle` UI and you should see on refreshing that there is a 50/50 chance that you will get the upvote-only version. If you open up incognito mode or a different browser and try to hit the frontend, you should get the alternative one. A different browser session is required, as the OpenShift router will by default return you to the same pod, so you'll always land on the same site version.

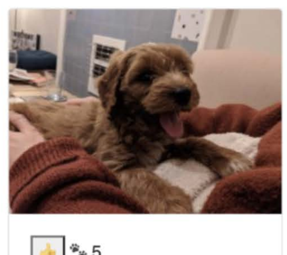
Refresh the Leaderboard

Upvote or Downvote your favourite cat to reveal another one ...

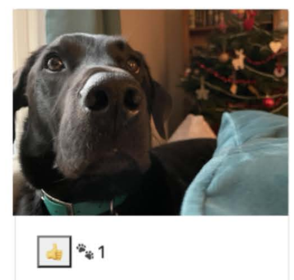
5



9



5



1

Figure 15.31: The no-downvote PetBattle frontend

Running `oc get routes` should show one route and more than one service connected to it with a 50/50 split `prod-pet-battle(50%),no-down-vote-pet-battle(50%)`. You can view the weights set as 100 each by running `oc get route prod-pet-battle -o yaml`:

```
# display the routes
$ oc get routes

NAME         HOST/PORT       PATH    SERVICES   PORT   TERMINATION    WILDCARD
prod-pet-battle       prod-pet-battle-labs-dev.apps…
      prod-pet-battle(50%),no-down-vote-pet-battle(50%)
             8080-tcp    edge/Redirect    None
```

The weights for the traffic routed to each application can be updated quite easily using Helm:

```
# update route weights
$ helm upgrade prod --set image_version=latest chart \
    --set a_b_deploy.svc_name=no-down-vote-pet-battle \
    --set a_b_deploy.weight=10 --namespace petbattle
```

Understanding the results

If we play around with the two versions that are deployed, we can see how the results of clicking the buttons are captured. If you open the Matomo app and log in, you will see some statistics there. The default password for Matomo, as set in the chart, is `My$uper$ecretPassword123#`. This might not be exactly secure out of the box but it can easily be changed via the Helm chart's values.

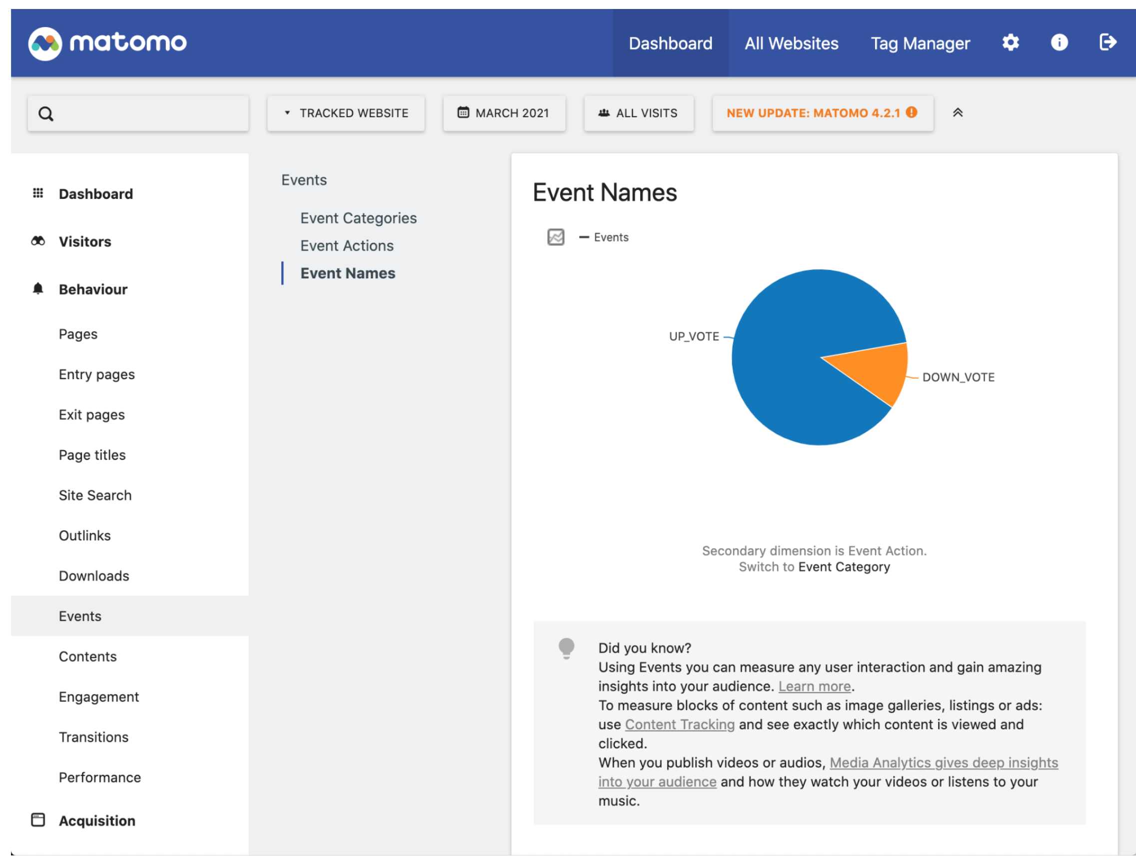


Figure 15.32: Matomo showing the number of clicks for UP_VOTE versus DOWN_VOTE

It might take a few minutes for Matomo to render the pie chart. Our simple experiment shows that more people use the **UP_VOTE** feature than the **DOWN_VOTE** feature. By connecting the A/B test to the data captured in Matomo, we can now make more informed decisions about the next actions that need to be taken for our product.

This experiment proves how easy it is to set up an A/B test. We can use the OpenShift platform to dynamically route users to multiple application versions concurrently deployed while we collect data about what is working well and what is not. There is some thinking that needs to be put into how we instrument the application to collect specific data, but the open source tooling available to us makes this easy too!

Blue/Green deployments

The Blue/Green deployment strategy is one of the fundamental deployment strategies that every team deploying applications into production should know about. Using this strategy minimizes the time it takes to perform a deployment cutover by ensuring you have two versions of the application available during deployment. It is also advantageous in that you can quickly roll back to the original version of the application without having to roll back any changes.

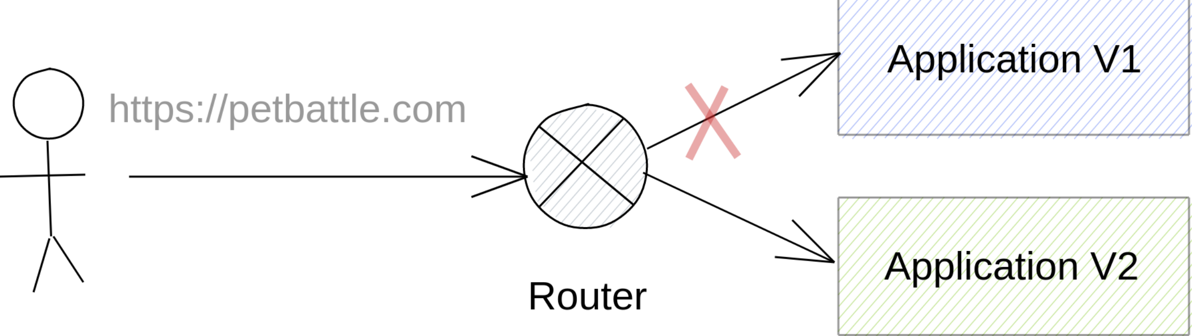


Figure 15.33: The canonical Blue/Green deployment

The trade-off here is that you need to have enough resources to be able to run two versions of the application stack you are deploying. If your application has persistent state, for example, a database or non-shared disk, then the application architecture and constraints must be able to accommodate the two concurrent versions. This is normally not an issue for smaller microservices and is one of the benefits of choosing that style of deployment.

Let's run through Blue/Green deployment using the PetBattle API as the example application stack. In this case, we are going to deploy two full stacks, that is, both the application and MongoDB. Let's deploy the blue version of our application:

```
# install the blue app stack
$ helm upgrade --install pet-battle-api-blue \
    petbattle/pet-battle-api --version=1.0.15 \
    --namespace petbattle --create-namespace
```

Now deploy the green application stack. Note that we have a different tagged image version for this:

```
# install the green app stack
$ helm upgrade --install pet-battle-api-green \
  petbattle/pet-battle-api --version=1.0.15 \
    --set image_version=green \
    --namespace petbattle
```

Next, we expose our production URL endpoint as a route that points to the blue service:

```
# create the production route
$ oc expose service pet-battle-api-blue --name=bluegreen \
--namespace petbattle
```

Finally, we can switch between the two using the `oc patch` command:

```
# switch service to green
$ oc patch route/bluegreen --namespace petbattle -p \
    '{"spec":{"to":{"name":"pet-battle-api-green"}}}'

# switch back to blue again
$ oc patch route/bluegreen --namespace petbattle -p \
    '{"spec":{"to":{"name":"pet-battle-api-blue"}}}'
```

If you browse to the `bluegreen` route endpoint, you should be able to easily determine the application stack:

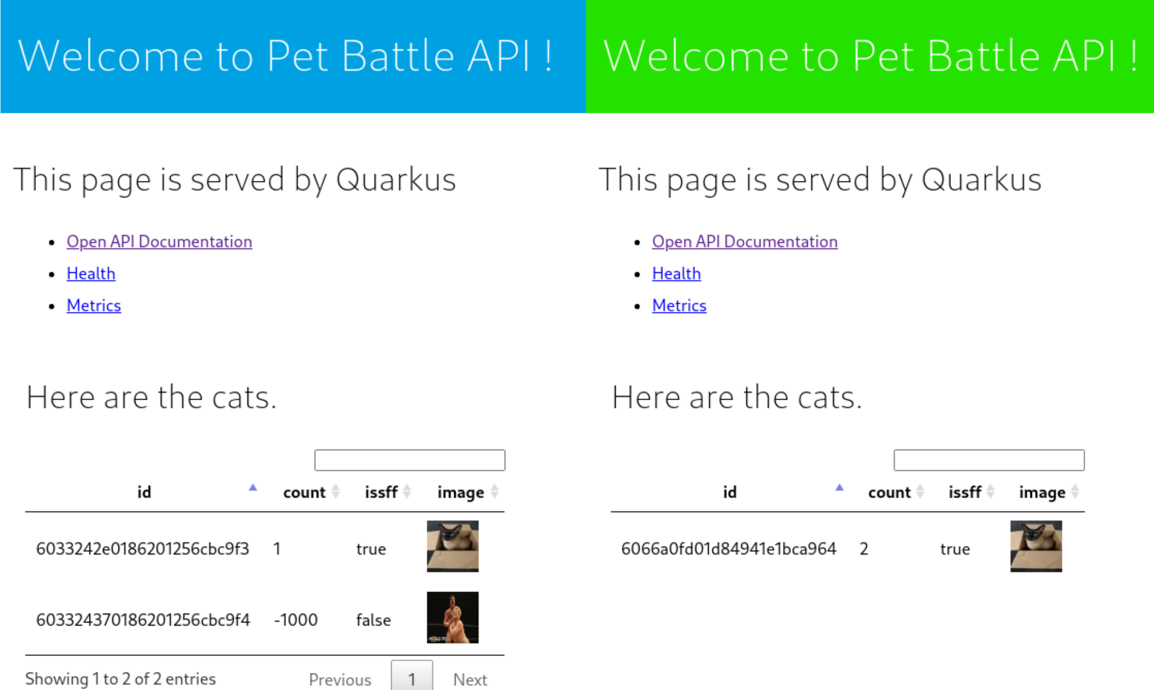


Figure 15.34: Blue/Green deployment for the PetBattle API

Even though this is somewhat of a contrived example, you can see the power of developers being allowed to manipulate the OpenShift routing tier in a self-service manner. A similar approach could be used to deploy the NSFF feature as an example – use the Helm chart parameters `--set nsff.enabled=true` to deploy an NSFF-enabled version. You can also point both applications to the same database if you want to with similar manipulation of the Helm chart values.

If you have more complex use cases where you need to worry about long-running transactions in the original blue stack, that is, you need to drain them, or you have data stores that need migrating alongside the green rollout, there are several other more advanced ways of performing Blue/Green deployments. Check out the ArgoCD rollout capability, which has a ton of advanced features,[55] the Knative Blue/Green rollout capability, or indeed Istio[56] for more ideas.

Deployment previews

We should think of OpenShift as something of a playground that we can use to deploy our applications for production all the way down to a developer preview. Gone are the days when a development team needed to raise a ticket to provision a server and manually configure it to show off their applications. Building applications in containers allows us to make shippable applications that can be repeatedly deployed in many environments. Our automation for PetBattle in Jenkins is configured to run on every commit. For Jenkins, we're using the multi-branch plugin so anytime a developer pushes a new feature to a branch, it will automatically scaffold out a new pipeline and deploy the latest changes for that feature.

When this was discussed in the previous chapter, about sandbox builds, you may have thought this was overkill and a bit of a waste. Why not just build on a pull request? It's a valid question to ask and depending on the objective you're trying to achieve, building on a pull request is probably sufficient. We have used the sandbox builds as another way to introduce feedback loops.

55 https://argoproj.github.io/argo-rollouts

56 https://github.com/hub-kubernetes/istio-blue-green-deployment

Developers do not exist in isolation; they are surrounded by other members of the team, including Product Owners and Designers. Our ability to dynamically spin up a new deployment of a feature from our pipeline means we can connect the coding efforts to the design team really easily. Developers can get very fast feedback by sharing a link to the latest changes or the implementation of a new feature with the design team. This feedback loop can quickly allow subtle changes and revisions to be made before the engineer loses the context of the piece of work. Creating deployment previews from every commit also allows a developer to very quickly share two versions of what an app might look like with a Product Owner while they make their decision about which to choose.

From our Jenkins pipeline, there is a branch called `cool-new-cat`. When this is built, it will push a new version of the app to the `dev` environment. The change in the app is subtle for illustrative purposes, but we can see the banner has been changed. With this new version of the app in the `dev` environment, we can get some feedback prior to merging it to master and generating a release candidate.

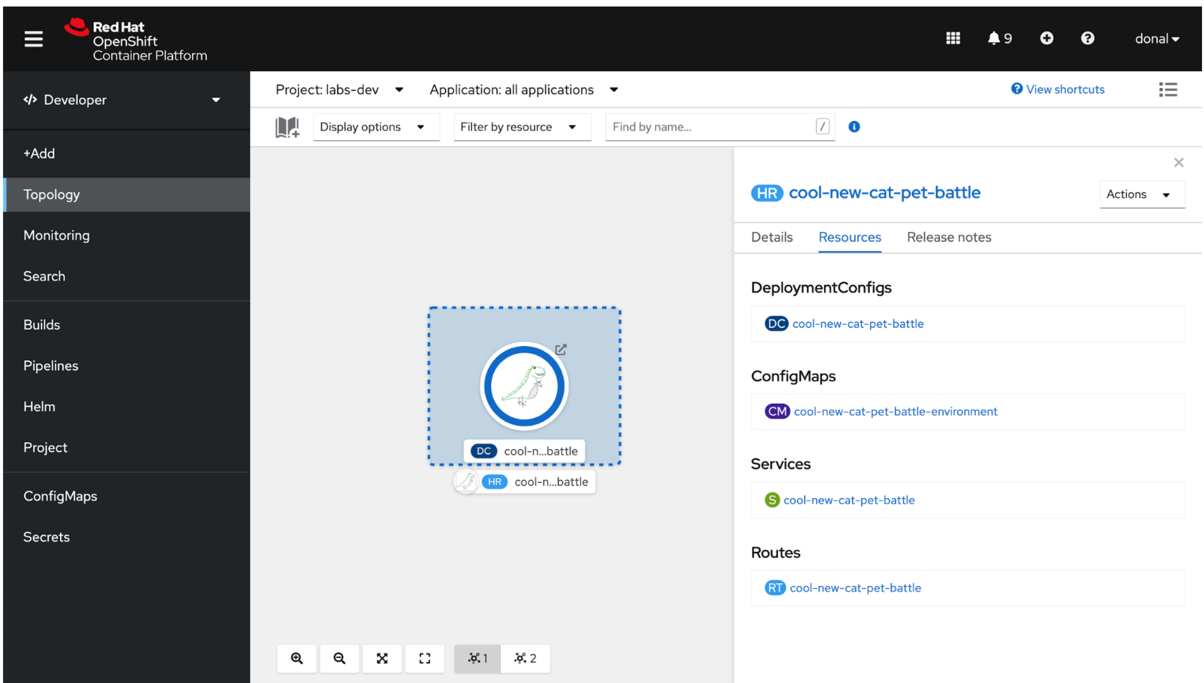


Figure 15.35: New feature deployed to the sandbox generating a deploy preview to collect feedback

Figure 15.35 shows the sandbox version of the being deployed along with it's associated route, service and configmap.

Conclusion

Congratulations! You've just finished the most technologically focused chapter of this book so far. Please don't go off and think that you have to use each and every technology and technique that has been mentioned—that's not the point. Investigate, evaluate, and choose which of these technologies applies to your own use cases and environment.

Several of the testing practices are part of our technical foundation. Unit testing, non-functional testing, and measuring code coverage are all critical practices for helping build quality into our applications and products from the start. We covered many small but invaluable techniques, such as resource validation, code linting, and formatting, that help make our code base less of a burden to maintain.

We covered a number of different approaches for deployments, including A/B, Canary, Blue/Green, and Serverless. These core techniques allow us to deliver applications more reliably into different environments. We even briefly covered artificial intelligence for reducing unwanted images uploaded into our PetBattle product. By focusing our efforts on what happens when things go wrong, we can more easily embrace and prepare for failures—big and small.

16

Own It

"*Annnddd we're live! PetBattle is finally in production, we can crack open the Champagne and toast our success*." But now what? How do we know that the site is doing what we expect it to do and, more importantly, how will we know when it isn't performing as we intended? Do we just sit around waiting for customers to complain that the site is down or that errors are happening? Not exactly a good user experience model—or a good use of our time.

In this chapter, we will discuss the tools and techniques that can be utilized to monitor the site and notify us when things start to go wrong so we can react before the entire site goes down. We will also discuss advanced techniques, such as Operators, that can help you automate a lot of the day-to-day operations.

Observability

Observability[1] is the process of instrumenting software components to assist with extracting data. This data can then be used to determine how well a system is functioning and subsequently be used to notify administrators in the event of issues.

1 https://en.wikipedia.org/wiki/Observability

When it comes to observing the state of our PetBattle applications, there are a number of aspects to consider:

- How do we know if an application instance is initialized and ready to process traffic?
- How do we know if the application has failed without crashing, such as by becoming deadlocked or blocked?
- How do we access the application logs?
- How do we access the application metrics?
- How do we know what version of the application is running?

Let's start by exploring some application health checks.

Probes

Hello, hello?... Is this thing on? In Kubernetes, the health of an application is determined by a set of software *probes* that are periodically invoked by the kubelet. A probe is basically an action invoked by the platform on each Pod that either returns a success value or a failure value.

Probes can be configured to perform one of the following types of actions:

- Connect to a specific TCP port that the container is listening on. If the port is open, the probe is considered successful.
- Invoke an HTTP endpoint, if the HTTP response code is 200 or greater but less than 400.
- Shell into a container and execute a command—this may involve checking for a specific file in the directory. This enables probes to be placed on applications that don't natively provide health checks out of the box. If the command exits with a status code of 0, then the probe is successful.

If a probe fails a configured number of times, the kubelet managing the Pod will take a pre-determined action, for example, by removing the Pod from the service or restarting the Pod.

Kubernetes currently supports three different kinds of probes:

1. **Readiness**: This decides whether the Pod is ready to process incoming requests. If the application needs some time to start up, this probe ensures that no traffic is sent to the Pod until this probe passes. Also, if the probe fails while it's running, the platform stops sending any traffic to the Pod until the probe once again succeeds. Readiness probes are key to ensuring a zero-downtime experience for the user when scaling up or upgrading Pods.

2. **Liveness**: This checks to see whether a Pod has a process deadlock or it's crashed without exiting; if so, the platform will kill the Pod.
3. **Startup**: This is used to prevent the platform from killing a Pod that is initializing but is slow in starting up. When the startup probe is configured, the readiness and liveness probes are disabled until the startup probe passes. If the startup probe never passes, the Pod is eventually killed and restarted.

Most of the time, you will probably only utilize the readiness and liveness probes, unless you have a container that's very slow in starting up.

In the PetBattle Tournament Service component, the liveness and readiness probes are configured as follows.

In `DeploymentConfig` (or the Deployment), the `/health/live` and `/health/ready` URLs are automatically created by the Quarkus framework:

```
...
livenessProbe:
  failureThreshold: 3
  httpGet:
    path: /health/live
    port: 8080
    scheme: HTTP
  initialDelaySeconds: 0
  periodSeconds: 30
  successThreshold: 1
  timeoutSeconds: 10
readinessProbe:
  failureThreshold: 3
  httpGet:
    path: /health/ready
    port: 8080
    scheme: HTTP
  initialDelaySeconds: 0
  periodSeconds: 30
  successThreshold: 1
  timeoutSeconds: 10
```

Different probes can invoke the same action, but we consider this bad practice.
The semantics of a readiness probe are different from those of a liveness probe. It's recommended that liveness and readiness probes invoke different endpoints or actions on the container.

For example, a readiness probe can invoke an action that verifies whether an application can accept requests. If during the Pod's lifetime the readiness probe fails, Kubernetes will stop sending requests to the Pod until the probe is successful again.

A liveness probe is one that verifies whether an application can process a request successfully; for example, if an application were blocked or accepting a request but waiting a long time for a database connection to become available, the probe would fail and Kubernetes would restart the Pod. Think of liveness probes as the Kubernetes equivalent of the IT Crowd[2] way of working.

Domino Effect

One question that we get asked a lot is, should a health check reflect the state of the application's downstream dependencies as well as the application itself? The absolute, definitive answer is *it depends*. Most of the time, a health check should only focus on the application, but there are always scenarios where this isn't the case.

If your health check functionality does a deep check of downstream systems, this can be expensive and result in cascading failures, where a downstream system has an issue and an upstream Pod is restarted due to this downstream issue. Some legacy downstream systems may not have health checks, and a more appropriate approach in this scenario is to add resilience and fault tolerance to your application and architecture.

Fault Tolerance

A key aspect of this is to utilize a **circuit breaker** pattern when invoking dependencies. Circuit breakers can *short circuit* the invocation of downstream systems when they detect that previous calls have failed. This can give the downstream system time to recover or restart without having to process incoming traffic.

The basic premise of a circuit breaker is that in the case of the failure of a downstream system, the upstream system should just assume that the next request will fail and not send it. It potentially also takes appropriate actions for recovery by, say, returning a default value.

After a given period of time, known as the backoff period, the upstream system should try sending a request to the downstream system, and if that succeeds, it reverts to normal processing. The rationale behind the backoff period is to avoid the situation where the upstream systems overwhelm the downstream systems with requests as soon as it starts up.

---

2 https://www.quotes.net/mquote/901983

Circuit breaker functionality can be performed at an individual level within an application's code: multiple frameworks such as Quarkus, Netflix Hystrix, and Apache Camel support circuit breakers and other fault-tolerant components. Check out the Quarkus fault tolerance plugin for more details.[3]

Platform-level circuit breaker functionality is provided by the **service mesh** component within OpenShift. This has various substantial advantages over application-level circuit breakers:

- It can be used on any container communicating via HTTP/HTTPS. A sidecar proxy is used to inject the circuit breaker functionality without having to modify the code.
- It provides dynamic configuration of the circuit breaker functionality.
- It provides metrics and visibility of the state of circuit breakers throughout the platform.
- Service mesh also provides other fault-tolerance functionalities, such as timeout and retries.

Logging

Ahh, logging! No *true* developer[4] has earned their stripes until they've spent countless hours of their existence trawling through production logs trying to figure out exactly what went wrong when a user clicked "Confirm". If you have managed to do this across multiple log files, all hosted on separate systems via multiple terminal windows, then you are truly righteous in the eyes of the IDE-bound masses.

The good news is that application logging on Kubernetes is a first-class citizen on the platform—just configure your application to write its logs to STDOUT and the platform will pick it up and you can view/trawl through them. OpenShift goes one level deeper by shipping an aggregated logging stack with EFK (Elasticsearch, Fluentd, and Kibana) out of the box. This allows developers to search and view logs across multiple containers running on multiple nodes across the cluster. If you want to give this a try, follow the documentation at https://docs.openshift.com/container-platform/4.7/logging/cluster-logging-deploying.html.

3 https://quarkus.io/guides/smallrye-fault-tolerance

4 https://en.wikipedia.org/wiki/No_true_Scotsman

Tracing

So, first things first: no, tracing is not application logging running at the trace log level. When it comes to OpenShift, tracing is the functionality added to the Kubernetes platform that enables developers to trace a request across a distributed set of application components running in different containers on different nodes of a cluster. Tracing is an exceptionally useful tool used to determine and visualize inter-service/component dependencies and performance/latency blackholes throughout a distributed system.

Tracing is provided as part of the OpenShift service mesh component. The underlying tracing functionality is provided by the Jaeger[5] distributed tracing platform. To support tracing, applications must include a client library that sends instrumented request metadata to a Jaeger collector, which in turn processes it and stores the data. This data can then be queried to help visualize the end-to-end request workflow. The Jaeger client libraries are language-specific and utilize the vendor-neutral OpenTracing specification.

If you're thinking, "*Woah! Collecting metadata for every request would be very expensive to store and process*," you'd be right. Jaeger *can* do this, but for scale purposes it's better to record and process a *sample* of requests, rather than each and every one of them.

Metrics

Probes are useful for telling when an application is ready to accept traffic, or whether it is stuck. Tracing is great at providing a measure of latency throughout a distributed system, while logging is a great tool to retrospectively understand exactly what occurred and when it occurred.

However, to comprehend the deep state (no, not *that* deep state!) of a system and potentially predict its future state after a period of time, you need to measure some of the key quantitative characteristics of the system and visualize/compare them over a period of time.

The good news is that metrics are relatively easy to obtain; you can get them from infrastructure components and software components such as JVM, and you can also add domain-specific/custom metrics to your application.

5 https://www.jaegertracing.io/

Given the multitude of metrics available, the hard bit is figuring out which metrics are valuable to your role and need to be retained; for example, for application operator connection pool counts, JVM garbage collection pause times are invaluable. For a Kubernetes platform operator, JVM garbage collection pause times are less critical, but metrics from platform components, such as etcd-related metrics, are crucial.

The good news is that OpenShift provides metrics for both the cluster and the applications running on it. In this section, we're going to focus on the application-level perspective. In the Kubernetes community, the *de facto* approach is to use Prometheus[6] for gathering and Grafana[7] for the visualization of metrics. This doesn't mean that you can't use other metrics solutions, and there are some very good ones out there with additional features.

OpenShift ships with both Prometheus and Grafana as the default metrics stack. Additionally, it also ships with the Prometheus Alertmanager. The Alertmanager facilitates the sending of notifications to operators when metric values indicate that something is going or has gone wrong and *la merde* has or is about to hit the fan. Examples of this include a high number of threads or large JVM garbage collection pause times.

Great, so how do we enable this for PetBattle? It is relatively straightforward:

1. Use a metrics framework in your application that records metrics and exposes the metrics to Prometheus.
2. Configure Prometheus to retrieve the metrics from the application.
3. Visualize the metrics in OpenShift.

Once the metrics are being retrieved, the final step is to configure an alert using the Prometheus Alertmanager.

Gather Metrics in the Application

Taking the PetBattle Tournament service component as an example, this is developed using the Quarkus Java framework. Out of the box, Quarkus supports/recommends the use of the open-source Micrometer metrics framework.[8]

6 https://prometheus.io/

7 https://grafana.com/oss/

8 https://micrometer.io/

To add this to the Tournament service, we simply need to add the dependency to the Maven POM along with the Prometheus dependency. For example:

```
<dependency>
   <groupId>io.quarkus</groupId>
   <artifactId>quarkus-micrometer</artifactId>
</dependency>
<dependency>
   <groupId>io.micrometer</groupId>
   <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Then, we configure a Prometheus registry, which is used to store the metrics locally in the application before being retrieved by the Prometheus collector. This is done in the `src/main/resources/application.properties` file.

```
# Metrics
quarkus.micrometer.enabled=true
quarkus.micrometer.registry-enabled-default=true
quarkus.micrometer.binder-enabled-default=true
quarkus.micrometer.binder.jvm=true
quarkus.micrometer.binder.system=true
quarkus.micrometer.export.prometheus.path=/metrics
```

With this configuration, the Prometheus endpoint is exposed by the application Pod. Let's go ahead and test it:

```
# grab the pod name for the running tournament service
$ oc get pod -n petbattle | grep tournament
$ oc exec YOUR_TOURNAMENT_PODNAME -- curl localhost:8080/metrics
...
# HELP mongodb_driver_pool_size the current size of the connection pool,
including idle and and in-use members
# TYPE mongodb_driver_pool_size gauge
mongodb_driver_pool_size{cluster_id="5fce8815a685d63c216022d5",server_
address="my-mongodb:27017",} 0.0
# TYPE http_server_requests_seconds summary
http_server_requests_seconds_
count{method="GET",outcome="SUCCESS",status="200",uri="/openapi",} 1.0
http_server_requests_seconds_
sum{method="GET",outcome="SUCCESS",status="200",uri="/openapi",} 0.176731581
http_server_requests_seconds_count{method="GET",outcome="CLIENT_
ERROR",status="404",uri="NOT_FOUND",} 3.0
http_server_requests_seconds_sum{method="GET",outcome="CLIENT_
ERROR",status="404",uri="NOT_FOUND",} 0.089066563
```

```
http_server_requests_seconds_
count{method="GET",outcome="SUCCESS",status="200",uri="/metrics",} 100.0
# HELP http_server_requests_seconds_max
# TYPE http_server_requests_seconds_max gauge
http_server_requests_seconds_
max{method="GET",outcome="SUCCESS",status="200",uri="/openapi",} 0.176731581
http_server_requests_seconds_max{method="GET",outcome="CLIENT_
ERROR",status="404",uri="NOT_FOUND",} 0.0
...
```

If successful, you should get an output similar to the above. Notice that you're not just getting the application-level metrics—the MongoDB connection pool metrics are also there. These are automatically added by the Quarkus framework once configured in the `application.properties` file.

Configuring Prometheus To Retrieve Metrics From the Application

Prometheus is somewhat unusual in its mode of operation. Rather than having some sort of agent pushing metrics to a central collector, it uses a pull model where the collector retrieves/scrapes metrics from a known HTTP/HTTPS endpoint exposed by the applications. In our case, as seen above, we're exposing metrics using the `/metrics` endpoint.

So how does the Prometheus collector know when, where, and how to gather these metrics? OpenShift uses a Prometheus operator[9] that simplifies configuring Prometheus to gather metrics. We just need to deploy a `ServiceMonitor` object to instruct Prometheus on how to gather our application metrics.

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
 labels:
   app.kubernetes.io/component: pet-battle-tournament
   k8s-app: pet-battle-tournament
 name: pet-battle-tournament-monitor
spec:
 endpoints:
 - interval: 30s
   port: tcp-8080
   scheme: http
 selector:
   matchLabels:
     app.kubernetes.io/component: pet-battle-tournament
```

9 https://github.com/prometheus-operator/prometheus-operator

There are a few things to note that might save you some time when trying to understand this configuration: basically, this configuration will scrape associated components every 30 seconds using the default HTTP path `/metrics`. Now, `port: tcp-8080` is mapped to the port name in the service—see below, highlighted in bold. If the service had a port name of `web`, then the configuration would be `port: web`.

```
$ oc describe svc my-pet-battle-tournament
Name:                my-pet-battle-tournament
Namespace:           pet-battle-tournament
Labels:              app.kubernetes.io/component=pet-battle-tournament
                     app.kubernetes.io/instance=my
                     app.kubernetes.io/managed-by=Helm
                     app.kubernetes.io/name=pet-battle-tournament
                     app.kubernetes.io/version=1.0.0
                     deploymentconfig=my-pet-battle-tournament
                     helm.sh/chart=pet-battle-tournament-1.0.0
Annotations:         Selector:  app.kubernetes.io/component=pet-battle-
tournament,app.kubernetes.io/instance=my,app.kubernetes.io/name=pet-battle-
tournament,deploymentconfig=my-pet-battle-tournament
Type:                ClusterIP
IP:                  172.30.228.67
Port:                tcp-8080  8080/TCP
TargetPort:          8080/TCP
Endpoints:           10.131.0.28:8080
Port:                tcp-8443  8443/TCP
TargetPort:          8443/TCP
Endpoints:           10.131.0.28:8443
Session Affinity:   None
Events:              <none>
```

User workload monitoring needs to be enabled at the cluster level before ServiceMonitoring will work.[10] This is also a classic demonstration of two of the major, powerful, misunderstood, and unused features of Kubernetes, *labels* and *label selectors*.

The following line means that Prometheus will attempt to retrieve metrics from all components that have the label `app.kubernetes.io/component: pet-battle-tournament`. We don't need to list each component independently; we just need to ensure that the component has the correct *label*, and that the *selector* is used to match that label. If we add a new component to the architecture, then all we have to do is ensure that it has the correct label. Of course, all of this assumes that the method of

10 https://docs.openshift.com/container-platform/4.7/monitoring/enabling-monitoring-for-user-defined-projects.html

scraping the metrics is consistent across all of the selected components; that they are all using the `tcp-8080 port`, for example.

```
selector:
  matchLabels:
    app.kubernetes.io/component: pet-battle-tournament
```

We're big fans of labels and their associated selectors. They're very powerful as a method of grouping components: Pods, Services, and so on. It's one of those hidden gems that you wish you knew of earlier.

Visualizing the Metrics in OpenShift

Once we have our metrics retrieved, we need to interpret the information that they're conveying about the system.

Querying using Prometheus

To visualize the metrics, go into the developer console and click on **Monitoring (1)**, as shown in *Figure 16.1*. Then click on **Custom Query (2)** in the dropdown, and enter a query using the **Prometheus query language (PromQL) (3)**. In the following example, we've used the *http_server_requests_seconds_count* metric, but there are others as well.

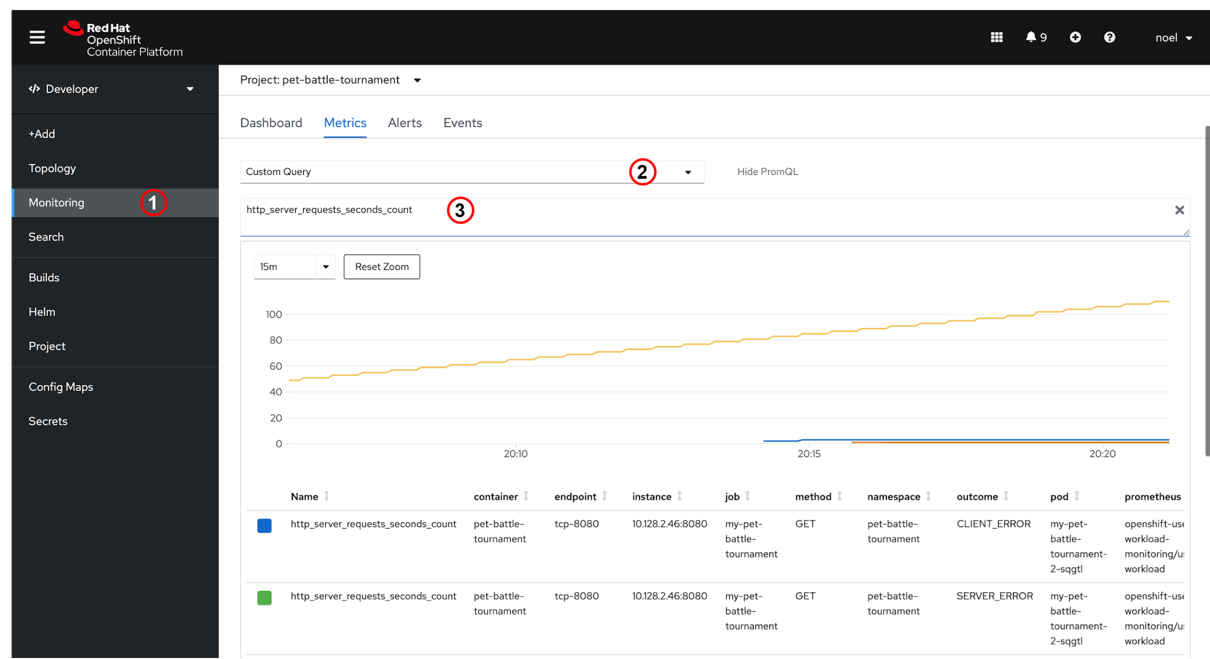


Figure 16.1: PetBattle tournament metrics

Let's explore some of the built-in dashboards OpenShift provides for monitoring.

Visualizing Metrics Using Grafana

OpenShift comes with dedicated Grafana dashboards for cluster monitoring. It is not possible to modify these dashboards and add custom application metrics, but it is possible to deploy an application-specific Grafana instance and customize that as we see fit. To do this, we first need to ensure that the Grafana Operator is installed in the namespace that we're using.

We will then deploy a custom Grafana setup by deploying the following custom resources:

- A *Grafana resource* used to create a custom *grafana* instance in the namespace
- A *GrafanaDataSource resource* to pull metrics from the cluster-wide Prometheus instance
- A *GrafanaDashboard* resource for creating the dashboard

The good news is that all of this is done via Helm charts, so you just have to do the following:

```
$ oc get routes
...
grafana-route     grafana-route-pb-noc.apps.someinstance.com
```

Open `grafana-route` in a browser, log in, *et voila*! It should look something like that shown in *Figure 16.2*. If there is an error with no data, check the BEARER_TOKEN is in place. This can be fixed manually by running the commands at https://github.com/petbattle/pet-battle-infra/blob/main/templates/insert-bearer-token-hook.yaml#L80

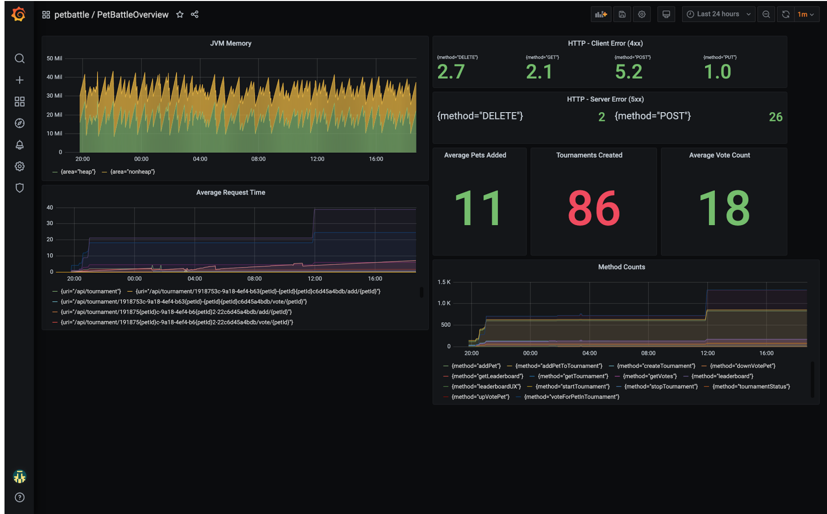


Figure 16.2: PetBattle metrics in Grafana

We will now take a look at some of the tools that can help us further with observability.

Metadata and Traceability

With the adoption of independently deployable services-based architectures, the complexity of managing these components and their inter-relationships is becoming problematic. In the following sections, we will outline a number of techniques that can assist you with this.

Labels

As mentioned earlier, labels and label selectors are among the more powerful *metadata management* features of Kubernetes. At its core, labels are a collection of text-based key/value pairs that can be attached to one or more objects: Pods, services, Deployments, and so on. Labels are intended to add information/semantics to objects that are relevant to the user and not the core Kubernetes system. A label selector is a method by which a user can group items together that have the same labels.

One of the most common uses of labels and label selectors in Kubernetes is the way that services use label selectors to group related Pods as endpoints for the service.

It's probably better shown by way of an example.

So, let's start with our three Infinispan Pods. Given that the Infinispan operator deploys its Pods via StatefulSets, the Pod names are pretty straightforward: `infinispan-0`, `infinispan-1`, `infinispan-2`. Take note of the labels attached to the Pods (highlighted in bold).

```
$ oc get pods --show-labels=true

NAME                                        READY   STATUS      RESTARTS   AGE
LABELS
infinispan-0                               1/1     Running     0          2m25s
app=infinispan-pod,clusterName=infinispan,controller-revision-hash=infinispan-
66785c8f,infinispan_cr=infinispan,statefulset.kubernetes.io/pod-
name=infinispan-0

infinispan-1                               1/1     Running     0          5m51s
app=infinispan-pod,clusterName=infinispan,controller-revision-hash=infinispan-
66785c8f,infinispan_cr=infinispan,statefulset.kubernetes.io/pod-
name=infinispan-1

infinispan-2                               1/1     Running     0          4m12s
app=infinispan-pod,clusterName=infinispan,controller-revision-hash=infinispan-
66785c8f,infinispan_cr=infinispan,statefulset.kubernetes.io/pod-
name=infinispan-2
```

When the Tournament service wants to connect to one of these Infinispan pods, it uses the Infinispan service that is also created and managed by the operator.

```
 $ oc get svc

NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)     AGE
infinispan   ClusterIP   172.30.154.122   <none>    11222/TCP   5d20h
```

If we go into the definition of the service, we'll see the selector (highlighted in bold):

```
$ oc describe service infinispan
Name: infinispan
Namespace: pet-battle-tournament
Labels: app=infinispan-service
        clusterName=infinispan
        infinispan_cr=infinispan
Annotations: service.alpha.openshift.io/serving-cert-signed-by:
               openshift-service-serving-signer@1607294893
             service.beta.openshift.io/serving-cert-secret-name:
               infinispan-cert-secret
             service.beta.openshift.io/serving-cert-signed-by:
               openshift-service-serving-signer@1607294893
Selector:          app=infinispan-pod,clusterName=infinispan

Type:              ClusterIP
IP:                172.30.154.122
Port:              infinispan  11222/TCP
TargetPort:        11222/TCP
Endpoints: 10.128.2.158:11222,10.129.3.145:11222,10.131.0.25:11222
Session Affinity:  None
Events:            <none>
```

This adds the Pods with the labels `app=infinispan-pod,clusterName=infinispan` into the service as endpoints. Two things to note here: the selector didn't use all the labels assigned to the Pod; and if we scaled up the number of Infinispan Pods, the selector would be continuously assessed and the new Pods automatically added to the service. The preceding example is a pretty basic example of a selector; in fact, selectors are far more powerful, with equality- and set-based operations also available. Check out the examples in the Kubernetes documentation for more information.[11]

11 https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

Great, so now what? What information could you use to label a resource? It depends on what your needs are. As demonstrated previously in the monitoring section, labels and selectors can be useful in configuring Prometheus. Labels can also be useful in grouping components together, as in the components that comprise a distributed application.

Kubernetes has a set of recommended labels[12] that we've used when building and deploying the PetBattle application:

Label	**Description**	**Value**
app.kubernetes.io/name	The name of the application	`pet-battle-tournament-service`
app.kubernetes.io/instance	A unique name identifying the instance of an application	`petbattle`
app.kubernetes.io/version	The current version of the application	`1.0.0`
app.kubernetes.io/component	The component within the architecture	`pet-battle-tournament-service`
app.kubernetes.io/part-of	The name of a higher-level application this one is part of	`petbattleworld`
app.kubernetes.io/managed-by	The tool being used to manage the operation of an application	`Helm`

Table 16.1: Kubernetes-recommended labels

12 https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/

With these labels in place, it is possible to retrieve and view the components of the application using selectors, such as to show the component parts of the PetBattle application without the supporting application infrastructure, that is, Infinispan or Keycloak. The following command demonstrates this:

```
$ oc get all -l app.kubernetes.io/part-of=petbattleworld \
    --server-print=false

NAME                                                         AGE
replicationcontroller/dabook-mongodb-1                       2d18h
replicationcontroller/dabook-pet-battle-tournament-1         26m

NAME                                                         AGE
service/dabook-mongodb                                       2d18h
service/dabook-pet-battle-tournament                         2d18h

NAME                                                         AGE
deploymentconfig.apps.openshift.io/dabook-mongodb            2d18h
deploymentconfig.apps.openshift.io/dabook-pet-battle-tournament  26m

NAME                                                         AGE
imagestream.image.openshift.io/dabook-pet-battle-tournament     26m

NAME                                                         AGE
route.route.openshift.io/dabook-pet-battle-tournament        2d18h
```

Let's look at other mechanisms we can use to enhance traceability.

Software Traceability

One of the issues that we've observed from customers over the years is the reliance that people have on the name of the software artifact that they're putting into production, such as `super-important-app-1.2.99.0.bin or critical-service-1.2.jar`. While this works 99.9% of the time, occasionally we've noticed issues where an incorrect version has been deployed with interesting outcomes.

In the land of containers, your deployment is a versioned artifact that contains a version of your software, and this in turn may be deployed using a versioned Helm chart via a GitOps approach. A good build and deployment pipeline will ensure that these levels of artifact versioning will always be consistent and provide traceability. As a backup, we also add additional traceability to the deployed artifacts as annotations on the resources and build info logging in the application binary.

Annotations

Annotations are similar to Kubernetes labels—that is, string-based key/value pairs—except that they're not used to group or identify objects via selectors. Annotations can be used to store different types of information; in our case, we're going to use annotations to store Git information to help with software traceability.

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    app.openshift.io/vcs-url:
      https://github.com/petbattle/tournamentservice.git
    app.quarkus.io/commit-id:a01a310aadd46911bc4c66b3a063ddb090a3feba
    app.quarkus.io/vcs-url:
      https://github.com/petbattle/tournamentservice.git
    app.quarkus.io/build-timestamp: 2020-12-23 - 16:43:07 +0000
    prometheus.io/scrape: "true"
    prometheus.io/path: /metrics
    prometheus.io/port: "8080"
```

The annotations are automatically added as part of the Maven build process using the Quarkus Maven plugin. Also notice the annotations are used to provide scrape information for Prometheus, as can be seen highlighted in the preceding code.

Build Information

An approach that has nothing to do with Kubernetes per se, but we strongly recommend to be used in general, is to output source control and build information as part of the application startup. An example of this is embedded into the Tournament service.

```
$ java -jar tournament-1.0.0-SNAPSHOT-runner.jar

GITINFO -> git.tags:
GITINFO -> git.build.version:1.0.0-SNAPSHOT
GITINFO -> git.commit.id.full:b5d6bfabeea6251b9c17ea52f0e87e2c8e967efd
GITINFO -> git.commit.id.abbrev:b5d6bfa
GITINFO -> git.branch:noc-git-info
GITINFO -> git.build.time:2020-12-24T13:26:25+0000
GITINFO -> git.commit.message.full:Moved gitinfo output to Main class
GITINFO -> git.remote.origin.url:git@github.com:petbattle/tournamentservice.
git
```

We use the Maven plugin `git-commit-id-plugin` to generate a file containing the Git information and package that file as part of the **Java archive** (**jar**). On startup, we simply read this file and output its contents to the console. Very simple stuff, but very effective and a lifesaver when needed. When running on OpenShift, this information will be picked up by the OpenShift logging components.

Alerting

So we have all the metrics to provide us with some insight into how the system is performing. We've got spectacular graphs and gauges in Grafana but we're hardly going to sit watching them all day to see if something happens. It's time to add alerting to the solution.

What Is an Alert?

An alert is an event that is generated when some measurement threshold (observed or calculated) is about to be or has been breached. The following are some examples of alerts:

- The average system response time in the last five minutes goes above 100 milliseconds.
- The number of currently active users on the site falls below a certain threshold.
- Application memory usage is approaching its maximum limit.

Alerts usually result in notifications being sent to human operators, whether that is through an email or instant message, say. Notifications can also be sent to trigger automation scripts/processes to deal with the alert. Service owners can analyze their existing alerts to help improve the reliability of their services and systems and reduce the manual work associated with remediating problems.

Why Alert?

Alerts call for human action when a situation has arisen within the system that cannot be automatically handled. This may include scenarios where automatic resolution of the problem is deemed too risky and human intervention is required to help triage, mitigate, and resolve the issue. Alerting can also be an issue by causing concern for site reliability engineers who manage and operate the system, particularly when alerts are numerous, misleading, or don't really help in problem cause analysis. They may generate benign alerts that don't prompt any action.

There are certain qualities that make up a *good alert*. Alerts should be actionable by the human beings who respond to them. To be actionable, the alert must also have arrived in time for something to be done about it and it should be delivered to the correct team or location for triaging. Alerts can also include helpful metadata such as documentation links to assist in making triage faster.

Alert Types

We can think of alerts as falling into three broad categories.[13] The first are **proactive** alerts, meaning that your business service or system is not in danger yet but may be in trouble after some period of time. A good example of this is where your system response time is degrading but it is not at a stage where external users would be aware of the issue yet. Another example may be where your disk quota is filling up but is not 100% full yet, but it may do in a few days' time.

A **reactive** alert means your business service or system is in immediate danger. You are about to breach a service level and immediate action is needed to prevent the breach.

An **investigative** alert is one where your business service or system is in an unknown state. For example, it may be suffering a form of partial failure or there may be unusual errors being generated. Another example may be where an application is restarting too many times, which is indicative of an unusual crash situation.

Each of these alerts may also be directed to different teams, depending on their severity. Not all alerts need to be managed with the same level of urgency. For example, some alerts must be handled by an on-call human resource immediately, while for others it may be fine to handle them during business hours by an application business support team the following day. Let's explore how we can easily configure and add alerting to our applications using the OpenShift platform features to help us out.

13 https://www.oreilly.com/content/reduce-toil-through-better-alerting/

Managing Alerts

OpenShift has platform monitoring and alerting that supports both built-in platform components and user workloads. The product documentation is the best place to start when looking to configure these.[14] As we outlined earlier, monitoring and alerting make use of the Prometheus monitoring stack. This is combined with an open-source tool called Thanos[15] that aggregates and provides access to multiple instances of Prometheus in our cluster.

A basic configuration for the PetBattle application suite consists of creating two ConfigMaps for user workload monitoring and alerting. We use ArgoCD and a simple kustomize YAML configuration to apply these ConfigMaps using GitOps. If we open up the ubiquitous journey `values-day2ops.yaml` file, we can create an entry for user workload monitoring.

```
# User Workload Monitoring
- name: user-workload-monitoring
  enabled: true
  destination: openshift-monitoring
  source: https://github.com/rht-labs/refactored-adventure.git
  source_path: user-workload-monitoring/base
  source_ref: master
  sync_policy: *sync_policy_true
  no_helm: true
```

The next step is to make use of application metrics and a ServiceMonitor and configure specific Prometheus alerts for our PetBattle suite.

User-Defined Alerts

In the *Metrics* section, we created ServiceMonitors for our API and Tournament applications that allow us to collect the micrometer metrics from our Quarkus applications. We want to use these metrics to configure our alerts. The simplest approach is to browse to the Thanos query endpoint that aggregates all of our Prometheus metrics. You can find this in the `openshift-monitoring` project.

```
$ oc get route thanos-querier -n openshift-monitoring
```

---

14 https://docs.openshift.com/container-platform/4.7/monitoring/configuring-the-monitoring-stack.html#configuring-the-monitoring-stack

15 https://github.com/thanos-io/thanos

We want to create a simple reactive alert based on whether the PetBattle API, Tournament, and UI Pods are running in a certain project. We can make use of Kubernetes Pod labels and the Prometheus query language to test whether our Pods are running.

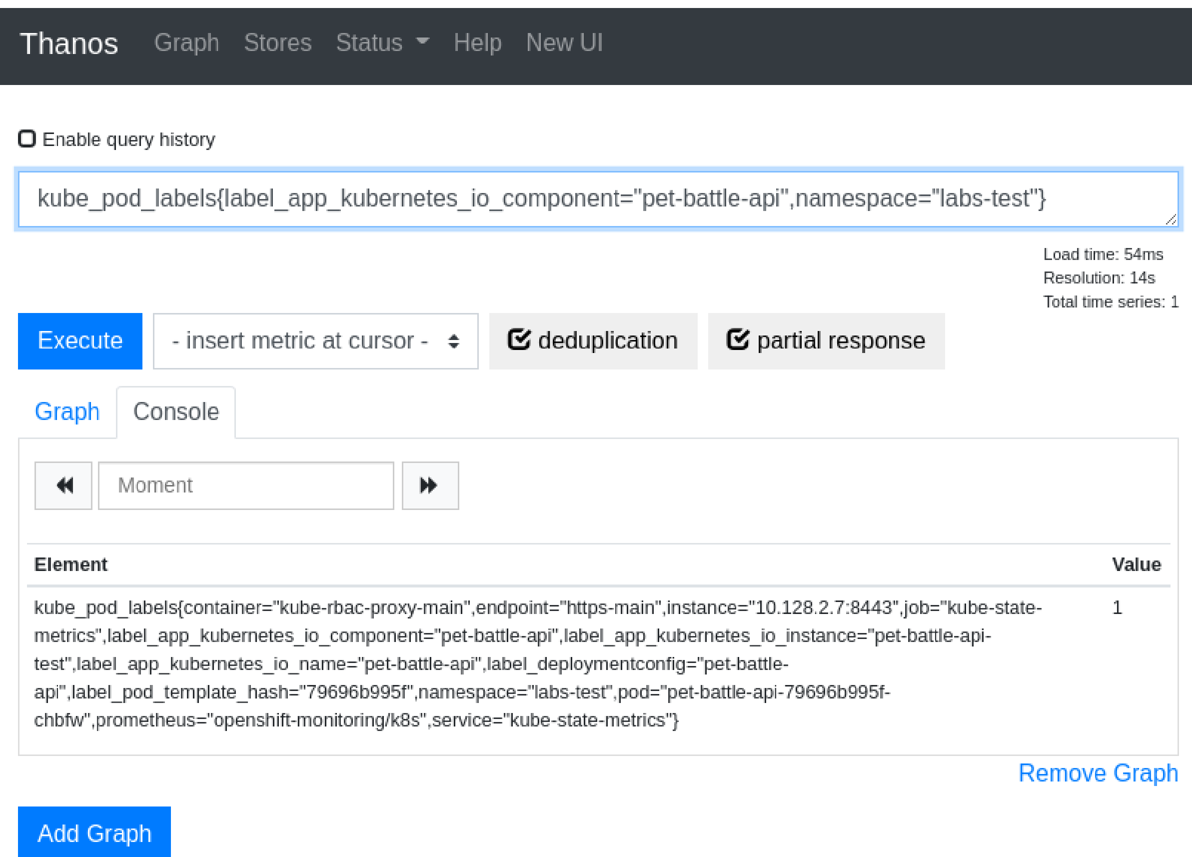


Figure 16.3: Thanos query interface

For this use case, we combine the `kube_pod_status_ready` and `kube_pod_labels` query for each Pod and namespace combination and create a PrometheusRule to alert us when a condition is not met. We wrapped the generation of the alerts in a Helm chart so we can easily template the project and alert severity values[16] and connect the deployment with our GitOps automation.

16 https://github.com/petbattle/ubiquitous-journey/blob/main/applications/alerting/chart/templates/application-alerts.yaml

```
spec:
  groups:
  - name: petbattle.rules
    rules:
    - alert: PetBattleApiNotAvailable
      annotations:
        message: 'Pet Battle Api in namespace {{ .Release.Namespace }} is
not available for the last 1 minutes.'
      expr: (1 - absent(kube_pod_status_ready{condition="true" ... for: 1m
      labels:
        severity: {{ .Values.petbattle.rules.severity }}
```

The firing alerts can be seen in the OpenShift web console as seen in *Figure 16.4*. In this example, we have configured the `labs-dev` alerts to only have a severity of *info* because they are not deemed as crucial deployments in that environment. The severity may be set as *info*, *warning*, or *critical*, and we use *warning* for our `labs-test` *and* `labs-staging` environments, for example. These are arbitrary but standard severity levels, and we can use them for routing alerts, which we will cover in a moment.

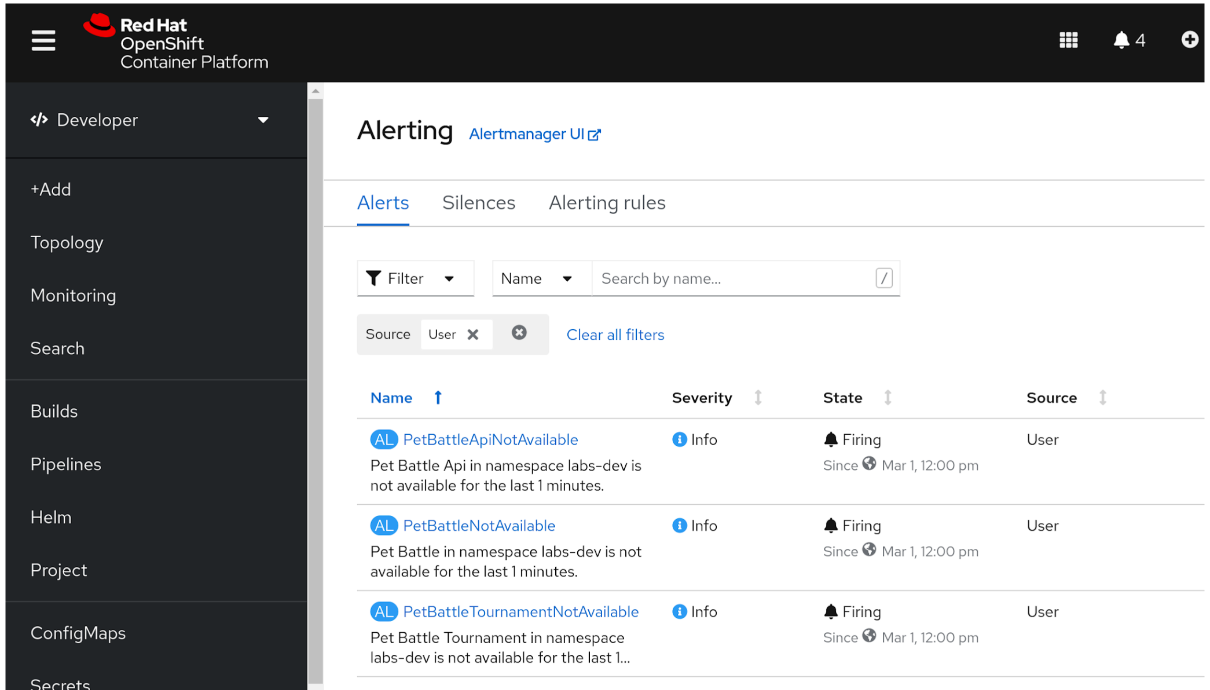


Figure 16.4: PetBattle alerts firing in OpenShift

We can use the same method to create an investigative or proactive alert. This time we wish to measure the HTTP request time for our API application. During testing, we found that if API calls took longer than ~1.5 sec, the user experience in the PetBattle frontend was deemed too slow by end users and there was a chance they would disengage from using the web application altogether.

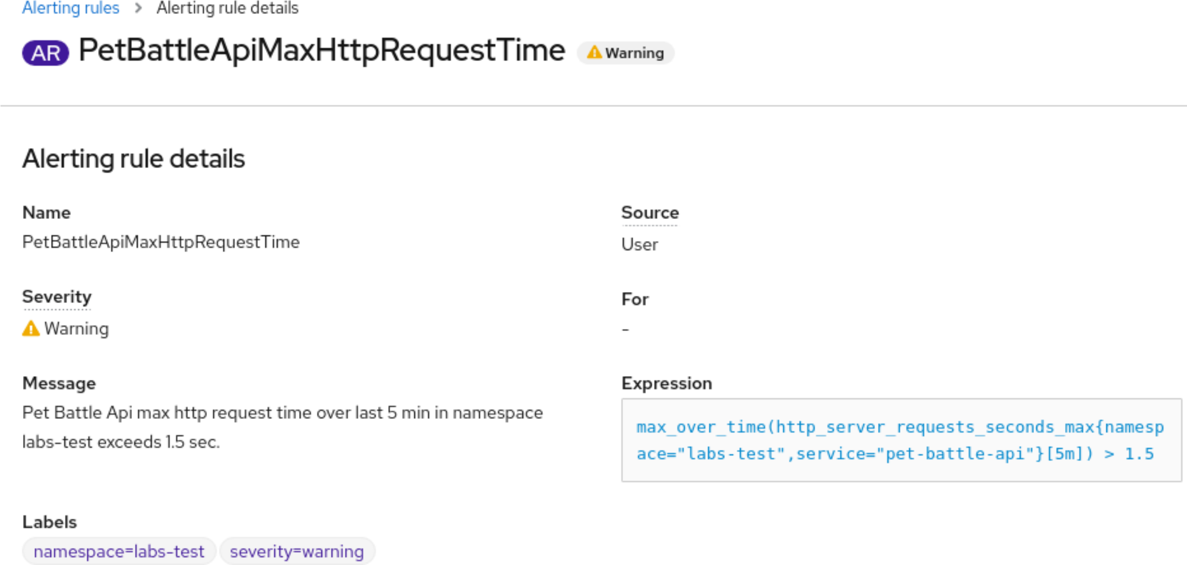


Figure 16.5: Maximum request time alert rule

In this alert, we use the Prometheus query language and the `http_server_requests_seconds_max` metric for the PetBattle API application to test whether the maximum request time over the last five-minute period exceeded our 1.5 sec threshold. If this alert starts to fire, possible remediation actions might include manually scaling up the number of API Pods or perhaps increasing the database resources if that is seen to be slow for some reason. In future iterations, we may even try to automate the application scale-up by using a Horizontal Pod Autoscaler, a Kubernetes construct that can scale our applications automatically based on metrics.

In this way, we can continue to build on our set of alerting rules for our PetBattle application suite, modifying them as we run the applications in different environments, and learn what conditions to look out for while automating as much of the remediation as we can.

OpenShift Alertmanager

As we have seen, OpenShift supports three severity levels of alerting: *info*, *warning*, and *critical*. We can group and route alerts based on their severity as well as on custom labels—that is, project or application labels. In the OpenShift administrator console,[17] you can configure the Alertmanager under **Cluster Settings**.

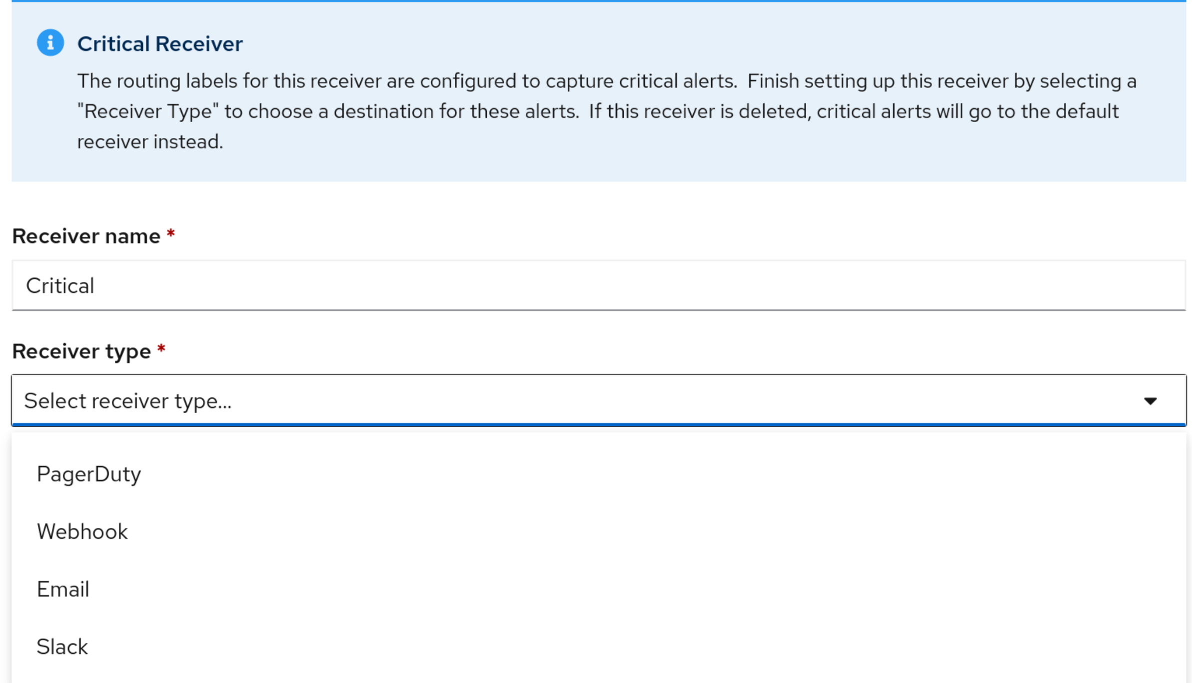


Figure 16.6: Alertmanager routing configuration

Alerts may be grouped and filtered using labels and then routed to specific receivers, such as PagerDuty, Webhook, Email, or Slack. We can fine-tune the routing rules so that the correct teams receive the alerts in the correct channel, based on their urgency. For example, all *info* and *warning* severity alerts for the PetBattle UI application may be routed to the *frontend developers* Slack channel, whereas all *critical* alerts are routed to the on-call PagerDuty endpoint as well as the Slack channel.

17 https://docs.openshift.com/container-platform/4.7/monitoring/managing-alerts.html

Alerting is a critical component to successfully manage the operational aspects of a system but you should be careful and ensure that the operations team isn't overwhelmed with alerts. Too many alerts or many minor or false-positive alerts can lead to *alert fatigue*, where it becomes an established practice within a team to ignore alerts, thus robbing them of their importance to the successful management of the system.

Service Mesh

Service mesh functionality has been one of the largest additions/extensions to Kubernetes in its short history. There's a lot of debate around the additional complexity of using a service mesh and whether all the features are even required.

For the purposes of this book, we're going to focus on the service mesh provided out of the box within OpenShift, which is based on the open-source Istio project. There are other implementations, such as Linkerd, SuperGloo, and Traefik, out there that are excellent and offer similar functionality to Istio.

The OpenShift service mesh provides the following features out of the box:

- **Security**: Authentication and authorization, mutual TLS (encryption), policies
- **Traffic management**: Resiliency features, virtual services, policies, fault injection
- **Observability**: Service metrics, call tracing, access logs

Why Service Mesh?

We previously talked about resiliency and how patterns like circuit breakers can help systems recover from downstream failures. A circuit breaker can be added in the scope of application code through frameworks such as **SmallRye Fault Tolerance** or **Spring Cloud Circuit Breaker** for Java projects; similar frameworks such as **Polly**[18] exist for .NET, **PyBreaker**[19] for Python, and **Opossum**[20] for Node.js. A key requirement for all of these frameworks is that they have to be added to the existing source code of the application, and the application needs to be rebuilt. When using a service mesh, a circuit breaker is external to the application code and no changes are required at the application level to take advantage of this feature.

18 https://github.com/App-vNext/Polly

19 https://pypi.org/project/pybreaker/

20 https://nodeshift.dev/opossum/

The same is true with **Mutual TLS** (**mTLS**), which is used for encrypting traffic between services. Operators such as CertManager or CertUtil can assist with managing and distributing certificates, but modification of the application code is still required to use the feature. Service meshes simplify this as the inter-component traffic is sent via a *sidecar proxy* and functionality such as mTLS is *automagically* added to this—once again, without having to change the application code.

The Istio component of a service mesh also manages TLS certificate generation and distribution so that it helps reduce the management overhead when using mTLS.

So how does a service mesh perform all of this magical functionality? Basically, the service mesh operator adds a service proxy container (based on the Envoy project) to the application Pod and configures the application traffic to be routed through this proxy. The proxy registers with the Istio control plane and configuration settings, certificates, and routing rules are retrieved and the proxy configured. The Istio documentation goes into much more detail.[21]

Aside – Sidecar Containers

A common object that people visualize when they hear the word *sidecar* is that of a motorbike with a single-wheel passenger car—a *pod*—attached to it. Being attached to the bike, the pod goes wherever the bike goes—except in comedy sketches where the bike and sidecar separate and rejoin, but that's another subject entirely.

In Kubernetes, a sidecar is a container that runs in the same Kubernetes Pod as the main application container. The containers share the same network and ICP namespace and can also share storage. In OpenShift, when using the service mesh functionality, a Pod annotated with the correct annotation `sidecar.istio.io/inject: "true"` will have an Istio proxy automatically injected as a sidecar alongside the application container. All subsequent communications between the application and external resources will flow through this sidecar proxy and hence enable the usage of features such as circuit breakers, tracing, and TLS, as and when they are needed. As the great Freddie Mercury once said, "*It's a kind of magic*."

```
# Let's patch the deployment for our pet battle apps
# running in petbattle ns and istioify it
$ helm upgrade \
--install pet-battle-tournament \
--version=1.0.39 \
--set pet-battle-infra.install_cert_util=true \
--set istio.enabled=true \
--timeout=10m \
```

21 https://istio.io/latest/docs/

```
--namespace petbattle
petbattle/pet-battle-tournament
$ oc get deployment pet-battle-tournament -o yaml \
--namespace petbattle
...
template:
    metadata:
      annotations:
...
        sidecar.istio.io/inject: "true"
      labels:
        app.kubernetes.io/component: pet-battle-tournament
        app.kubernetes.io/instance: pet-battle-tournament
        app.kubernetes.io/name: pet-battle-tournament
```

It is possible to have more than one sidecar container if required. Each container can bring different features to the application Pod: for example, one for Istio, another for log forwarding, another for the retrieval of security credentials, and so on. It's easy to know when a Pod is running more than a single container; for example, the `READY` column indicates how many containers are available per Pod and how many are ready—that is, its readiness probe has passed.

```
$ oc get pods

NAME                              READY   STATUS     RESTARTS   AGE
infinispan-0                     1/1     Running    0          4h13m
Pet-battle-3-68fm5                2/2     Running    0          167m
pet-battle-api-574f77ddc5-l5qx8 2/2     Running    0          163m
pet-battle-api-mongodb-1-7pgfd  1/1     Running    0          3h50m
pet-battle-tournament-3-wjr6r   2/2     Running    0          167m
pet-battle-tou.-mongodb-1-x7t9h 1/1     Running    0          4h9m
```

Be aware, though, that there is a temptation to try and utilize all the service mesh features at once, known as the *ooh… shiny* problem.

Here Be Dragons!

The adoption of a service mesh isn't a trivial exercise when it comes to complex solutions with multiple components and development teams. One thing to understand about a service mesh is that it crosses a lot of team boundaries and responsibilities. It includes features that are focused on the developer, operations, and security teams; all of these teams/personnel need to work together to understand and get the best out of using the features provided by the mesh. If you're just starting out, our advice is to start small and figure out what features are necessary in production and iterate from there.

In the case of PetBattle, we decided that we were going to primarily focus on using some of the features in the areas of traffic management and observability. The rationale behind this was that Keycloak already addressed many of the security requirements, and we also wanted to finish the book before the end of the decade.

Service Mesh Components

The functionality of the service mesh is made up of a number of independent components:

- Jaeger and Elasticsearch provide the call tracing functionality and logging functionality.
- Kiali provides the mesh visualization functionality.
- OpenShift Service Mesh provides the core Istio functionality.

The good news is that all of these components are installed and managed by Operators, so installation is reasonably straightforward. These components are installed via Helm, and if you want to know more about how they are installed, then the Red Hat OpenShift documentation will have the relevant details.

One key thing to note is that at the time of writing this book, OpenShift Service Mesh ships with a downstream version of Istio called Maistra. This is primarily due to the out-of-the-box multi-tenancy nature of OpenShift, as well as limiting the scope of Istio cluster-scoped resources. OpenShift Service Mesh also ships with an **Istio OpenShift Routing** (**IOR**) component that maps the Istio gateway definitions onto OpenShift routes. Note that Istio is still the upstream project and bugs/feature requests are fixed/implemented, as necessary.

For traffic management, Istio has the following core set of resources:

- **Gateways**: Controls how traffic gets into the mesh from the outside, akin to OpenShift routes.
- **Virtual service**: Controls how traffic is routed within the service mesh to a destination service. This is where functionality such as timeouts, context-based routing, retries, mirroring, and so on, are configured.
- **Destination rule**: Service location where traffic is routed to once traffic rules have been applied. Destination rules can be configured to control traffic aspects such as load balancing strategies, connection pools, TLS setting, and outlier detection (circuit breakers).

There are other resources such as service entry, filters, and workloads, but we're not going to cover them here.

PetBattle Service Mesh Resources

We'll briefly introduce some of the resources that we use in PetBattle and explain how we use them.

Gateways

The gateway resource, as stated earlier, is used to create an ingress route for traffic coming into the service mesh.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: petbattle-gateway-tls
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE
      credentialName: "pb-ingressgateway-certs"
    hosts:
    - "*"
```

A few things to note with this definition is that it will create an OpenShift route in the *istio-system* namespace and not the local namespace. Secondly, the route itself will use SSL, but it won't be able to utilize the OpenShift router certificates by default. Service mesh routes have to provide their own certificates. As part of writing this book, we took the pragmatic approach and copied the OpenShift router certificates into the *istio-system* namespace and provided them to the gateway via the *pb-ingressgateway-certs* secret. Note that this is for demonstration purposes only—*do not try this in production*. The correct approach for production is to generate and manage the PKI using as-a-service certificates.

Virtual services

PetBattle contains a number of VirtualServices, such as *pet-battle-cats-tls*, *pet-battle-main-tls*, and *pet-battle-tournament-tls*.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: pet-battle-cats-tls
spec:
  hosts:
  - "*"
  gateways:
  - petbattle-gateway-tls
  http:
  - match:
    - uri:
        prefix: /cats
      ignoreUriCase: true
    route:
    - destination:
        host: pet-battle-api
        port:
          number: 8080
    retries:
      attempts: 3
      perTryTimeout: 2s
      retryOn: gateway-error,connect-failure,refused-stream
```

The VirtualServices are all similar in function in that they are all configured to:

1. Match a specific URI; in the example above, `/cats`.
2. Once matched, route the traffic to a specific destination.
3. Handle specific errors by performing a fixed number of request retries.

Destination Rule

Finally, the traffic is sent to a destination or even distributed to a set of destinations depending on the configuration. This is where DestinationRules come into play.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: pet-battle-api-port
spec:
  host: pet-battle-api.prod.svc.cluster.local
  trafficPolicy: # Apply to all ports
    portLevelSettings:
    - port:
        number: 8080
      loadBalancer:
        simple: LEAST_CONN
```

In our example, the traffic sent to a specific port is load balanced based on a simple strategy that selects the Pod with the least number of active requests. There are many load balancing strategies that can be used here, depending on the needs of the application—everything from simple round robin to advanced consistent hashing load balancing strategies, which can be used for session affinity. As ever, the documentation goes into far greater detail.[22]

We can visualize the flow of traffic from the above example, as seen in *Figure 16.7*:

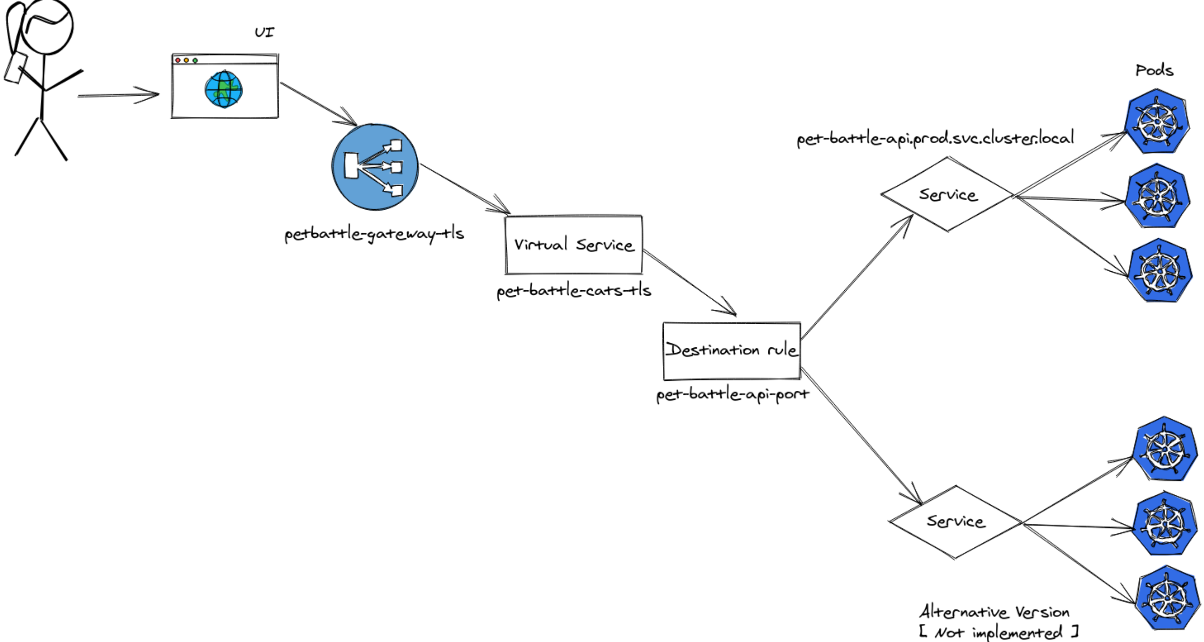


Figure 16.7: PetBattle traffic flow

22 https://istio.io/latest/docs/

Note that *Figure 16.7* shows an example of how destination rules can be used to send traffic to an alternative version of the service. This can be useful for advanced deployment strategies such as Canary, Blue/Green, and so on. We haven't discussed how to do this with OpenShift Service Mesh in this book, but the reader is encouraged to explore this area in more detail. A good place to start is the aforementioned Istio documentation.

Managing all of these resources is reasonably simple when it's just a few services, and PetBattle utilizes service mesh functionality in a very basic manner. However, when there are many services and features, such as multiple destinations used in advanced deployment models, the amount of settings and YAML to interpret can be overwhelming. This is where mesh visualization functionality can be useful to visualize how all of this works together. For this, we use the Kiali functionality, which is part of OpenShift Service Mesh. *Figure 16.8* shows how PetBattle is visualized using Kiali.

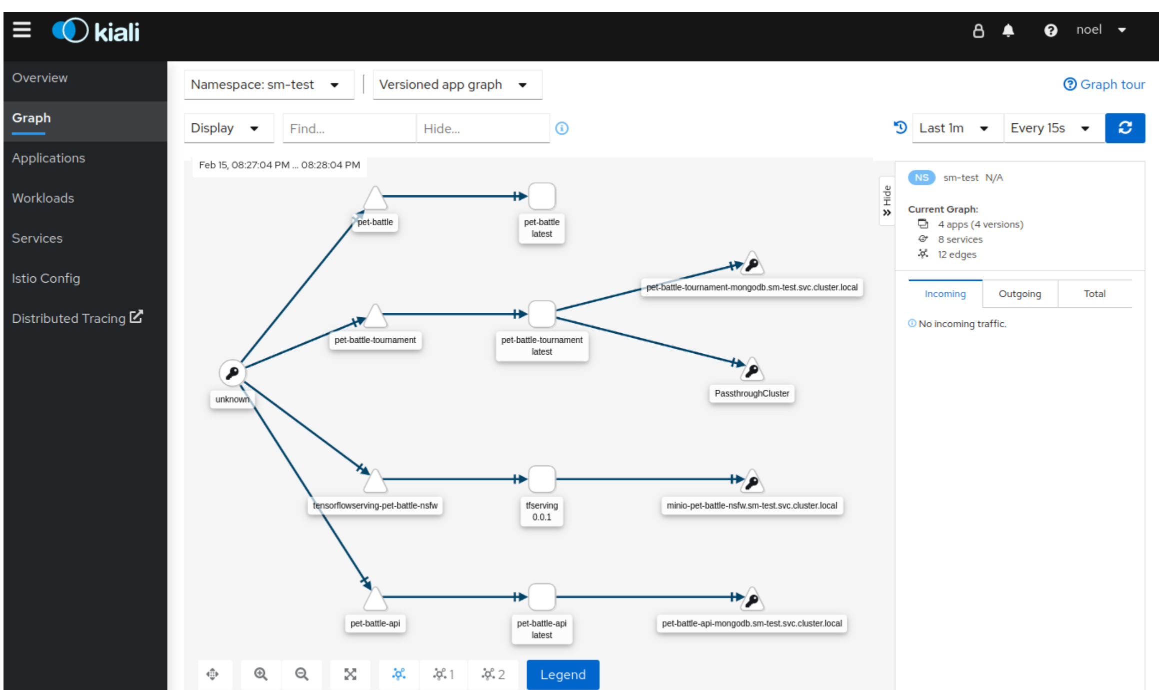


Figure 16.8: Kiali service graph for PetBattle

Kiali can be very useful for diagnosing the current state of the mesh, as it can dynamically show where traffic is being sent as well as the state of any circuit breakers being used. It also integrates with Jaeger for tracing requests across multiple systems. Kiali can also help prevent configuration issues by semantically validating the deployed service mesh resources.

Next we're going to explore one of the most powerful features of OpenShift 4 - Operators.

Operators Everywhere

Fundamental to the OpenShift 4 platform is the concept of *Operators*. So far, we have used them without talking about why we need them and what they actually represent on a Kubernetes platform such as OpenShift. Let's cover this briefly without totally rewriting the book on the subject.[23]

At its heart, the Operator is a software pattern that codifies knowledge about the running and operation of a particular software application. That application could be a distributed key value store, such as etcd. It might be a web application such as the OpenShift web console. Fundamentally, the operator can represent *any* application domain that could be codified. A good analogy for an operator is the *expert system*, a rules-based bit of software that represents knowledge about a certain thing that is put to work in a meaningful way. If we take a database as an example, the Operator might codify what a real human database administrator does on a day-to-day basis, such as the deployment, running, scaling, backup, patching, and upgrading of that database.

The physical runtime for an operator is nothing more than a Kubernetes Pod, that is, a collection of containers that run on a Kubernetes platform such as OpenShift. Operators work by extending or adding new APIs to the existing Kubernetes and OpenShift platform APIs. This new endpoint is called a **Custom Resource** (**CR**). CRs are one of the many extension mechanisms in Kubernetes.

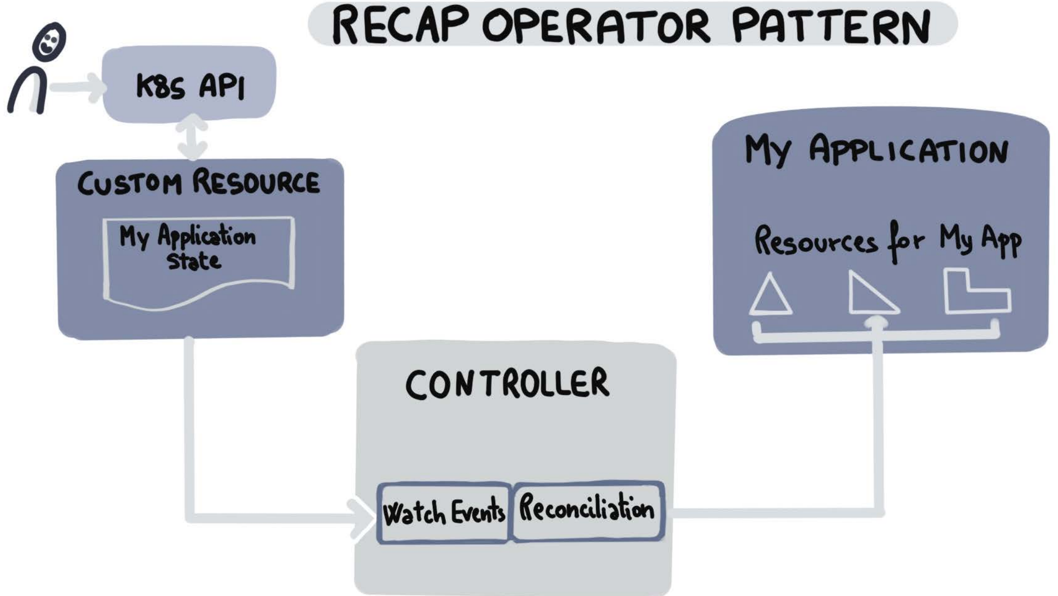


Figure 16.9: The Operator pattern

23 https://www.redhat.com/en/resources/oreilly-kubernetes-operators-automation-ebook

A **Custom Resource Definition** (**CRD**) defines what the CR is. Think of it as the definition or schema for the CR. The Operator Pod *watches* for events on the platform that are related to their custom resources and takes *reconciliation* actions to achieve the desired state of the system. When an Operator Pod stops or is deleted from the cluster, the application(s) that it manages should continue to function. Removing a CRD from your cluster does affect the application(s) that it manages. In fact, deleting a CRD will in turn delete its CR instances. This is the Operator pattern.

With Operators, all of the operational experience required to run/manage a piece of software can be packaged up and delivered as a set of containers and associated resources. In fact, the whole of the OpenShift 4 platform exists as a collection of operators! So, as the platform owner, you are receiving the most advanced administrator knowledge bundled up through Operators. Even better, Operators can become more advanced over time as new features and capabilities are added to them. A good understanding of how to configure Operators is required for OpenShift platform administrators. This usually involves setting properties in the OpenShift cluster global configuration web console, setting CR property values, using ConfigMaps, or similar approaches. The product documentation[24] is usually the best place to find out what these settings are for each Operator.

Cluster Settings

Details ClusterOperators Global configuration

Edit the following resources to manage the configuration of your cluster.

Filter by name or description... /

Configuration resource

APIServer

Alertmanager

Authentication

Figure 16.10: OpenShift Cluster Settings—configuring platform Operators

24 https://docs.openshift.com/container-platform/4.7

In OpenShift, the lifecycle management (upgrading, patching, managing) of Operators themselves is automated through the **Operator Lifecycle Manager** (**OLM**). These components make upgrading the OpenShift platform itself a lot more reliable and easier to manage from a user's perspective; it massively reduces the operational burden. Because Operators themselves are delivered as versioned images, we get the same benefits from immutable container images that we do for our own applications, i.e., the same image version can be run consistently in multiple cloud environments increasing quality and eliminating snowflakes (unique applications for unique environments).

And it is not just the OpenShift platform itself that can take advantage of Operators. The sharing and distribution of software using Operators via the Operator Hub[25] is open to software developers and vendors from all over the world. We use OLM and Operator *subscriptions* to deploy these into our cluster. The tooling required to build and develop Operators (the SDK) is open source and available to all.[26]

So, should *all* applications be delivered as Operators? The short answer is no. The effort required to code, package, test, and maintain an Operator may be seen as overkill for many applications. For example, if your applications are not being distributed and shared with others, and you only need to build, package, deploy, and configure your application in a few clusters, there are many simpler ways this can be achieved, such as using container images, Kubernetes, and OpenShift native constructs (BuildConfigs, Deployments, ReplicaSets, ConfigMaps, Secrets, and more) with tools such as Helm to achieve your goals.

Operators Under the Hood

To fully understand how operators work, you need to understand how the Kubernetes control loop works.

Control Loops

In very basic terms, core Kubernetes is just a **Key-Value** (**KV**) store—an etcd datastore with an API. Processes use this API to perform **Create**, **Read**, **Update**, and **Delete** (**CRUD**) actions on keys within the KV store. Processes can also register with the KV store to be notified when there are value changes to keys or sets of keys that they're interested in.

25 https://operatorhub.io

26 https://github.com/operator-framework/operator-sdk

When these processes get a change notification, they react to that notification by performing some activity, such as configuring iptables rules, provisioning storage, and so on. These processes understand the current state of the system and the desired state and work toward achieving that desired state. In other words, these processes are performing the role of a *control loop*, meaning they attempt to bring the state of the system to a desired state from where it currently resides.

In this example, the *process* is a controller that observes the state of a resource or set of resources and then makes changes to move the resource state closer to the desired state. As consumers of Kubernetes, we constantly use controllers. For example, when we instruct Kubernetes to deploy a Pod, the Pod controller works to make that a reality. Control loops are key to the operation on Kubernetes and it's a declarative and, eventually, consistent approach. For much more information, take a look at the Kubernetes Controller docs[27] and the OpenShift blog site for recommendations on how to build your own Operator.[28]

Operator Scopes

Operators can either be cluster-scoped or namespace-scoped. A cluster-scoped operator is installed once in a namespace and can create and manage resources in other namespaces; that is, cluster-wide. The OpenShift service mesh operator and its related operators such as Kiali and Jaeger are cluster-scoped. They are installed by default into the `openshift-operators` or `openshift-operators-redhat` namespace and create and manage resources when a related CRD is deployed in another namespace, such as PetBattle.

A namespace-scoped operator is one that is deployed in a namespace and only manages resources in that namespace. We use a number of these in PetBattle, such as Cert-Utils and Keycloak.

All Operators are installed via a CRD called a **Subscription**. Without going into too much detail (see the official documentation for more), a Subscription describes how to retrieve and install an instance of an operator. The following is an example of a Subscription that we use to install the Grafana operator.

27 https://kubernetes.io/docs/concepts/architecture/controller/

28 https://www.openshift.com/blog/kubernetes-operators-best-practices

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
 name: grafana-operator
spec:
 channel: alpha
 installPlanApproval: Automatic
 name: grafana-operator
 source: community-operators
 sourceNamespace: openshift-marketplace
 startingCSV: grafana-operator.v3.7.0
```

To see some of the namespace-scoped operators that PetBattle needs, run the following command.

```
$ oc get subscriptions

NAME                PACKAGE             SOURCE                CHANNEL
cert-utils-operator cert-utils-operator community-operators   alpha
grafana-operator    grafana-operator    community-operators   alpha
infinispan          infinispan          community-operators   2.1.x
keycloak-operator   keycloak-operator   community-operators   alpha
```

Let us now take a look at how operators can be used by our PetBattle team.

Operators in PetBattle

We use operators to create and manage resources such as Infinispan cache and Keycloak SSO instances. We simply install the Infinispan operator and deploy a relevant custom resource to tell it to create and manage a replicated cache. We don't have to know about spinning up Infinispan Pods or creating SSL certificates or provisioning storage space. The operator will do all of this for us, and if something fails or is accidentally deleted, the operator will look after the recreation of the resource. In the Infinispan example, if we delete the Infinispan K8s service, the operator will be notified about its deletion and recreate the service automatically. As developers, we don't have to worry about managing it.

It is simpler to think of Operators as *looking after stuff so you don't have to*. It is also possible to use multiple operators in combination to automate complex workflows. For example, we use Keycloak for its SSO gateway and user management functionality. The Keycloak instance is deployed and managed via a Keycloak Operator. We just need to build and send a custom resource to the API and the operator will do the rest. One of the resources managed by the Operator is a Kubernetes Secret that contains the TLS certificates and keys, which clients interacting with the Keycloak instance will need to use. Given that Keycloak is the security gateway to our application, it is prudent to ensure that all communications are encrypted. However, this causes issues for Java-based applications; to use SSL, the JVM requires that it be provided with a Java TrustStore containing the SSL/TLS certificates and keys so that the JVM can trust them.

So, how do we take the Secret with the TLS certificates and keys and convert that into a TrustStore that the Java applications can use? We could do a whole heap of scripting with Bash, the Java Keytool, and potentially other tools to extract the certs/keys, creating the TrustStore, converting, and finally injecting the certs/keys into said TrustStore. This is manual, complex, and error-prone work. We will also have to recreate these TrustStores for each environment and handle lifecycle events such as certificate expiry.

Alternatively, we could use an operator, in this case, the *Cert-Utils* operator. We first install the *Cert-Utils* operator in the PetBattle namespace. This Operator was developed by the Red Hat Consulting PAAS Community of Practice[29] to help manage certificates and JVM Keystores, along with TrustStores.

To use this Operator, we first create a ConfigMap containing a set of specific annotations. The Cert-Utils operator will detect these annotations and create a TrustStore containing the relevant certificates and keys; it will also add the TrustStore to the ConfigMap. Finally, we can mount the ConfigMap into a Deployment and instruct the JVM to use that TrustStore. The following resource definition will create the TrustStore with the relevant certificates and keys.

```
apiVersion: v1
kind: ConfigMap
metadata:
  annotations:
    service.beta.openshift.io/inject-cabundle : "true"
    cert-utils-operator.redhat-cop.io/generate-java-truststore: "true"
    cert-utils-operator.redhat-cop.io/source-ca-key: "service-ca.crt"
    cert-utils-operator.redhat-cop.io/java-keystore-password: "jkspassword"
  name: java-truststore
```

This does the following:

29 https://github.com/redhat-cop/cert-utils-operator

- The `service.beta.openshift.io/inject-cabundle` annotation will inject the service signing certificate bundle into the ConfigMap as a `service-sa.crt` field.
- The `cert-utils-operator.redhat-cop.io` annotation will create the Java TrustStore in the ConfigMap under the name `truststore.jks` with the `jkpassword` password.

In the Tournament service, the following Quarkus configuration will mount the `java-truststore` ConfigMap and configure the JVM accordingly.

```
# Mount the configmap into the application pod in the /tmp/config/ directory
quarkus.kubernetes-config.enabled=true
quarkus.openshift.config-map-volumes.javatruststore.config-map-name=java-
truststore
quarkus.openshift.mounts.javatruststore.path=/tmp/config/

# Instruct the JVM to use the Truststore
quarkus.openshift.env-vars.JAVA_OPTS.value=-Djavax.net.ssl.trustStore=/tmp/
config/truststore.jks -Djavax.net.ssl.trustStorePassword=jkspassword

# Tell Infinispan client to use the Truststore when connecting
quarkus.infinispan-client.trust-store=/tmp/config/truststore.jks
quarkus.infinispan-client.trust-store-password=jkspassword
```

We've only just scratched the surface of operators. OpenShift ships with a number of supported operators and there are many community operators available as well. We used many community-based operators in this book, such as the Infinispan operator and Keycloak operator; there are productized versions of these operators available as well. There are many more operators from multiple vendors available from OperatorHub.[30]

It is also possible to write your own operators if required. The OperatorFramework[31] is an open-source SDK with which you can write your own operators using either Go, Ansible, or Helm.

30 https://operatorhub.io/

31 https://operatorframework.io/

Service Serving Certificate Secrets

Keycloak uses an OpenShift feature called *service serving certificate secrets*.[32] This is used for traffic encryption. Using this feature, OpenShift automatically generates certificates signed by the OpenShift certificate authority and stores them in a secret. The application, in this case Keycloak, can then mount this secret and use these certificates to encrypt traffic. Any application interacting with a Keycloak instance then just has to trust these certificates. OpenShift also manages the lifecycle of these certificates and automatically generates new certificates when the existing certificates are about to expire.

To turn on this feature, simply add the following annotation to a service:

```
service.beta.openshift.io/serving-cert-secret-name=<NameOfMysecret>
```

In the case of Keycloak, the operator does this as part of its processing:

```
$ oc get svc keycloak -o yaml

apiVersion: v1
kind: Service
metadata:
  annotations:
    description: The web server's https port.
    service.alpha.openshift.io/serving-cert-secret-name: sso-x509-https-
secret
    service.alpha.openshift.io/serving-cert-signed-by: openshift-service-
serving-signer@1615684126
    service.beta.openshift.io/serving-cert-signed-by: openshift-service-
serving-signer@1615684126
```

The secret contains the actual certificate and associated key:

```
$ oc get secret sso-x509-https-secret -o yaml

apiVersion: v1
data:
  tls.crt: …...
  tls.key: ….
kind: Secret
metadata:
  annotations:
```

32 https://docs.openshift.com/container-platform/4.7/security/certificates/service-serving-certificate.html

And it contains the certificate details as well:

```
$ oc get secret sso-x509-https-secret -o json \
    | jq -r '.data."tls.crt"' | base64 --decode \
    |openssl x509  -text -noout

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1283774295358672234 (0x11d0e1eb7ea18d6a)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = openshift-service-serving-signer@1615684126
        Validity
            Not Before: Mar 15 08:34:54 2021 GMT
            Not After : Mar 15 08:34:55 2023 GMT
        Subject: CN = keycloak.labs-staging.svc
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
```

Such Operator patterns simplify the burden of running complex middleware infrastructure applications on the OpenShift platform.

Conclusion

To be able to successfully run your software at scale in production, a good understanding of the instrumentation that surrounds the software stack is required. OpenShift is a modern platform that provides all of the capabilities required to observe and, in a lot of cases, automatically heal your applications while they're running.

In this chapter, we have discussed many common technical patterns that allow application developers to make use of these common platform capabilities. For example, one of the simplest patterns is to always log to STDOUT so the platform logging mechanisms can be leveraged. With containers, it becomes an antipattern to log to specific files mounted in a temporary filesystem within your container, because they are not clearly visible.

More complex patterns are also important to keep your business service applications running, even during disruption and change. Correctly configuring liveness, readiness, and startup probes so that your application can deploy without loss of service, configuring Pod disruption budgets for when nodes are restarted. Using application features to expose Prometheus metric endpoints for alerting and monitoring on the platform is a great way to alert teams when human interaction is required.

The service mesh is an advanced extension to OpenShift, extrapolating many features that would have traditionally been packaged into your applications so they can be more efficiently managed at the platform level. This is a common theme: taking application and development cross-cutting features and leveraging them to the benefit of all platform services.

The Operator pattern eases the operational burden of running complex middleware infrastructure applications on the OpenShift platform, packaging all the years of expert knowledge as software. It is no secret that OpenShift itself uses this fantastic pattern for all of its core capabilities. The real power comes in being able to lifecycle manage this complexity in an automated manner. Human toil is massively reduced because the system can self-heal and auto-upgrade without interference. Doing more with less is still the name of the game.

As a cross-functional product team, once you have learned and mastered these capabilities, it really does become possible to *give the developers the pagers*. The quality of any business service delivery starts with business discovery, which then transitions to application software, expands through to platform capabilities, and finally on and out into the world of networking and end user devices connected via the internet. Once developers and cross-functional product teams are empowered to build, run, and own their software—in every environment that it is required to run in—only then will they fully equate and connect happy customers with the software supply chain that they code, automate, and continuously deliver.

Section 7: Improve It, Sustain It

This book has been a journey. We established the foundation in *Section 2*, *Establishing the Foundation*, to support our Application Product team. We moved that team onto the Discovery Loop portion of the Mobius Loop in *Section 3*, *Discover It*. They then progressed into the Options Pivot in *Section 4*, *Prioritize It*, before moving onto their first iteration of the Delivery Loop in *Section 5*, *Deliver It*.

We then dove deep into building applications with the PetBattle product team in *Section 6*, *Build It*, *Run It*, *Own It*.

In the last segment of the Mobius Loop journey, we return to the Options Pivot and ask ourselves what we learned from the Delivery Loop.

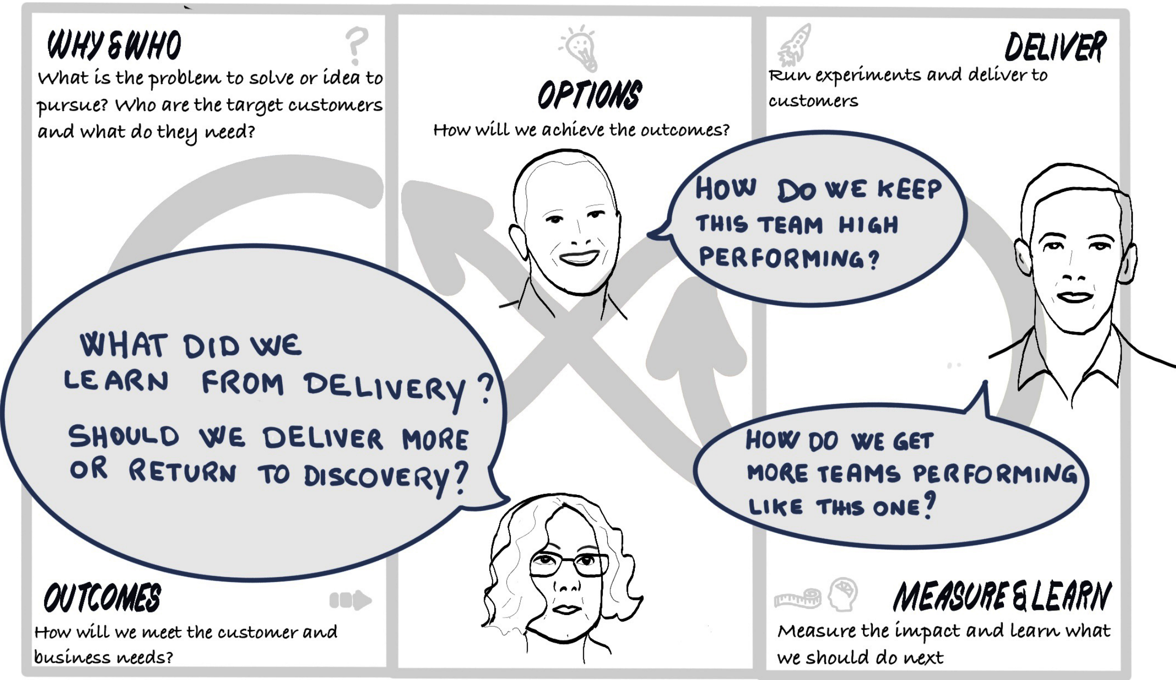


Figure 17.0.1: Improvement and sustainability – setting the scene

What should we do next? Should we go around the Delivery Loop again or should we return to the Discovery Loop? What about revisiting our options? These will be the questions we answer in *Chapter 17*, *Improve It*.

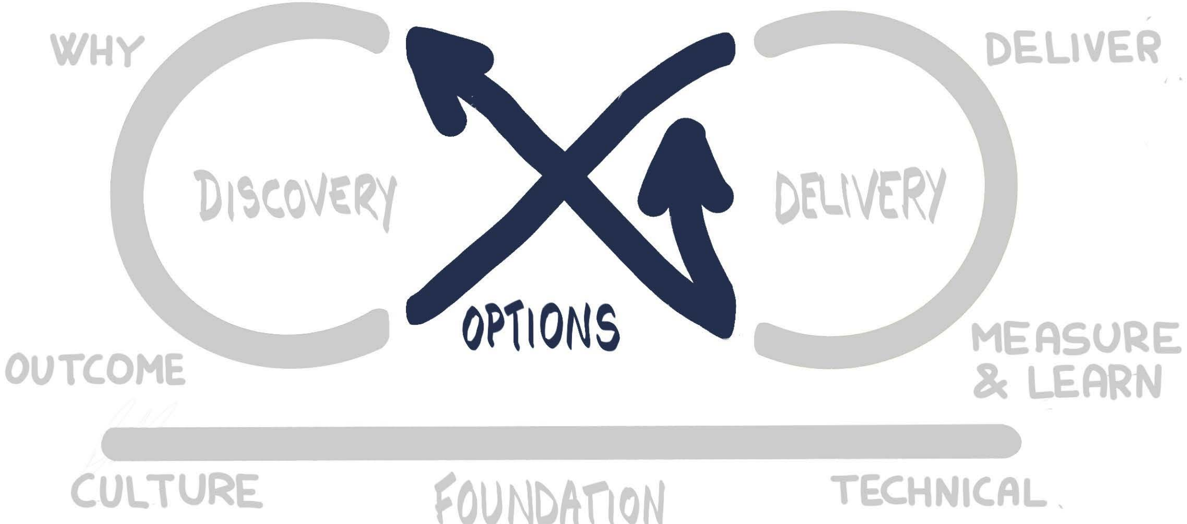


Figure 17.0.2: The Options Pivot

In *Chapter 18*, *Sustain It*, we will revisit the entire journey we've been on and explore what it takes to sustain this way of working, including how to sustain the people, processes, and technology improvements we have been making throughout this book. We can then start to scale and spread these ways of working to other teams throughout the organization.

17

Improve It

We did it! We made it all the way around the Mobius Loop.

First, we built a foundation of open culture, open leadership, and open technology. We successfully navigated our way around the Discovery Loop using practices such as the North Star and Impact Mapping to discover our Why, and practices such as Empathy Mapping and other human-centered design tools to discover our Who. We even started the Discovery of our How by commencing some Event Storming, Non-Functional Mapping, and Metrics-Based Process Mapping. We did all of this to gather just enough information and just enough collective, shared understanding to derive some measurable Target Outcomes.

We used these Target Outcomes to guide our way through the Options Pivot. We explored several prioritization techniques and practices such as User Story Mapping, Value Slicing, impact and effort prioritization, how/now/wow prioritization, design sprints, and weighted-short-job-first to produce our initial Product Backlog. We designed experiments that went into this backlog.

We then moved to the first iteration of the Delivery Loop, where features were coded, applications were written, and we carried out experiments and conducted research. We made use of established Agile practices to achieve this in a way where we could measure and learn against the original outcomes as quickly as possible.

In the previous sections, we took a deeper dive into the technology that teams use to build, run, and own their solution.

In each part of the Mobius Loop, we collected valuable information that was summarized at the end of the section on canvases. If we piece these three canvases together, we can see how everything connects:

- Discovery:
  - Who were we doing this for and why?
  - What was their problem, need, or opportunity?
  - What were the customer and organizational outcomes we set out with?
  - What was the impact on the outcomes?
- Options:
  - What were the actions we agreed to deliver?
  - What were the options that would help reach the outcomes?
  - What was the relative priority?
  - What did we learn?
- Delivery:
  - What was done?
  - What did we say we were going to research, experiment, and launch?

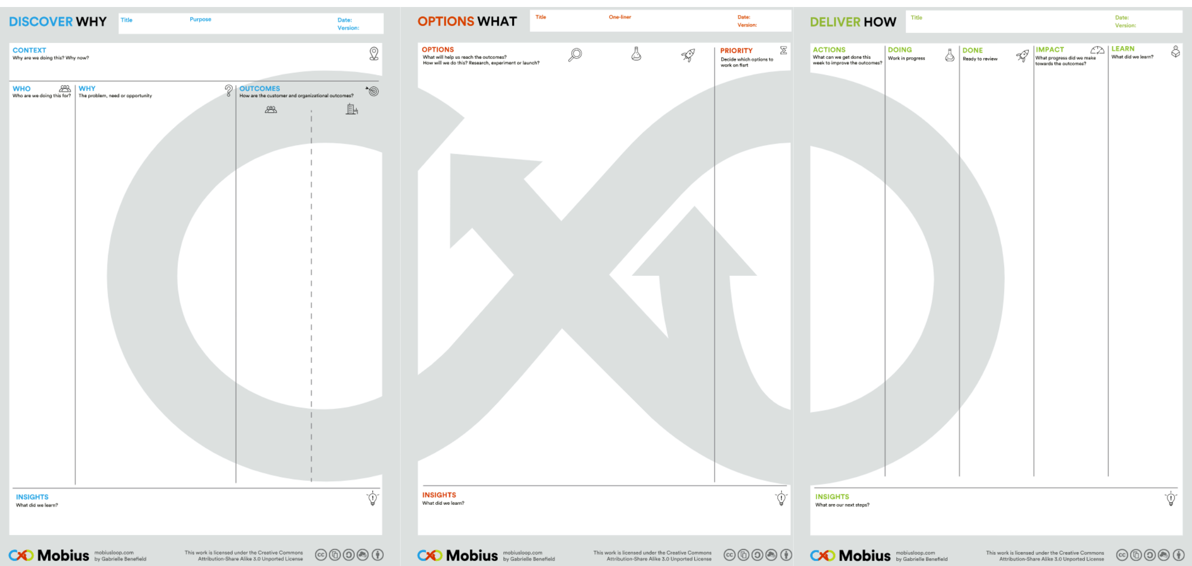


Figure 17.1: The Mobius Loop

This chapter is called *Improve It*. Maybe the whole book should be called that because, really, continuous improvement is what it's all about. Everything we do is focused on how we can continuously improve; whether this is the technology, our users' experience, our culture, or the metrics that we employ.

In this chapter, we are going to explore what we do when we reach the end of an iteration of the Delivery Loop: What did we learn? Did we learn enough? Have we moved toward our target measurable outcomes? And, most importantly, what should we do next?

What Did We Learn?

In *Chapter 13*, *Measure and Learn*, we explored the techniques that we use to measure and learn from the increments of the products we deliver in a Delivery Loop by measuring the following:

- Feedback from Showcase and Retrospective events
- Learning from user testing
- Capturing the results of experiments run
- Service delivery and operational performance
- Service level agreements, service level indicators, and service level objectives
- Security
- Performance
- Culture
- Application metrics
- Infrastructure platform and resource usage

This learning is very important and should drive conversations, inferences, and conclusions about what was learned. This is why visualizing these metrics is so powerful. Instantly, we can all see what the current measure is, what the measure was before the last iteration of the Delivery Loop, and what the target measure is to achieve the desired outcome and impact.

If the conversation suggests we are not able to learn from these metrics, we need to inspect why that is. Running a deep Retrospective to ask why we are not learning enough from our Delivery Loops can be very helpful. Techniques such as the Five Whys or the Ishikawa Diagram are excellent deep retrospective approaches in facilitating these discussions and driving improvement actions that the team can put in place to facilitate learning.

Ultimately, teams need to decide whether they are measuring what matters, whether the measures are accurate and reflective of their work, and whether the data is confidently taking them toward their Target Outcomes. The most important question teams should ask themselves is: *Did we learn enough?*

Did We Learn Enough?

We have a very important decision to make at this point of the Mobius Loop. As we come out of the Delivery Loop, we can decide to go around the Delivery Loop again or we can return to the Options Pivot to revisit and reprioritize our options based on the metrics and learning captured during the Delivery Loop iteration. Otherwise, we could proceed back around the Discovery Loop.

Questions we like to ask based on our learning include the following:

- As a result of what we have delivered, have we reached an outcome? Or do we need to pull some more items from the Options list or Product Backlog and do some more delivery? If so, turn right and go around the Delivery Loop again.
- Do the metrics captured from the most recent Delivery Loop suggest we've reached one or more measurable outcomes? If so, return to Discovery to validate this and work toward the next set of outcomes.
- Does the learning from our Delivery validate, invalidate, or improve understanding around assumptions and hypotheses made during Discovery? If so, let's go back to Discovery and update those artifacts.
- Has the learning from our Delivery given us some new information about the priorities of options? If so, let's go back to the Options Pivot and revisit some of those prioritization practices.
- Have the results from experiments given us some ideas for new or improved experiments? If so, let's go back to the Options Pivot and design those experiments.

Over time, you will do more iterations of Discovery and Delivery Loops and spend more time in the Options Pivot. The Mobius Loop provides a fantastic visualization as to how long you've spent on each loop, how fast you've traveled around each loop, and how often you've pivoted from one loop to another. It will also inform what level of balance you have between continuous discovery and continuous delivery.

Some warning signs to look out for include the following:

- **We just keep going round and round in circles of the Delivery Loop**. This suggests that we're not taking the time to revisit and re-assess outcomes and are moving toward being a feature factory and blindly building outputs.
- **We spend too much time on the Discovery Loop**. This suggests that we are in a mode of **analysis paralysis**. We overthink and overanalyze our Why and Who and never get to test our ideas or hypotheses. We may risk missing market windows or not delivering anything.

- **We jump from Discovery to Delivery**. This suggests that we're not taking the learning from Discovery and distilling, organizing, and making some important decisions about what to deliver next or how to get knowledge and learning more quickly (through, for example, research or experiments, as opposed to blindly building features).
- **We jump from Delivery to Discovery**. This suggests we're not taking the time to factor in learning back to our options and prioritization.
- **We never move back to another part of the Mobius Loop**. This suggests we're not working in an iterative or incremental way and not building learning into our system of work. This is really linear work and, as we saw in *Chapter 12*, *Doing Delivery*, when we explored **Cynefin**, it is only really a good solution when work is in a simple domain.

Let's look at a story where a major pivot and shift between loops was triggered by learning.

We Need Two Apps, Not One!

This comes from one of our first European Open Innovation Labs residencies[1]. A start-up in Switzerland was looking to disrupt healthcare with a remote check-in application to the **emergency room** (**ER**).

Together with the customer, we used Event Storming to understand the business process and identify key user flows. The team used User Story Mapping to identify the early slices

that could be used to test the market, followed by Scrum to deliver a very early product in three one-week sprints.

The focus on the first iteration of Delivery was all about the app that patients would use when they needed to go to the ER. It was entirely focused on their experience and most user research had been on these patient actors.

1 https://www.redhat.com/en/blog/red-hat-welcomes-swiss-based-medical-company-easier-ag-waterford-its-emea-open-innovation-labs

When stakeholders saw the first increment of the app, there was a sudden epiphany moment. In order to fully appreciate, understand, and learn from this solution, we would need to have prototypes for not one but two apps. The experience of doctors and nurses triaging requests would be key.

This triggered an immediate shift back into the Discovery Loop to focus on these personas and processes as the team lacked insight into their needs and problems.

This is one of many examples where the team gathered *just enough* information and feedback to trigger a pivot and change in direction to initial thinking.

"Just Enough" Leads to Continuous Everything

Throughout this book, we've used the phrase *just enough* many times. Right back in the opening paragraphs, we wondered if you had just enough information from the back cover of this book to explore more. In *Chapter 7*, *Open Technical Practices – The Midpoint*, when we were building the technical foundation, we said we were building just enough of an architecture so that product developments kept moving forward. When we moved on to the Discovery Loop in *Chapter 8*, *Discovering the Why and Who*, we used practices to get just enough information to align people and get a shared understanding and enough confidence to go into the delivery of early experiments. In *Chapter 9*, *Discovering the How*, when we started to discover our How by using practices such as Event Storming, we got just enough information to be able to design how certain components would interact. And in *Chapter 10*, *Setting Outcomes*, we explained that we only ever do just enough Discovery to progress to the Options Pivot and start an iteration of Delivery.

Now that we've been around the Delivery Loop, what was just enough information to start with has grown. We know much more now than when we first started. From the measures and learning captured out of delivery practices, user testing, experimental results, and metrics systems, we now have the opportunity to revisit and update the Discovery practices. Let's look at a few examples.

The Impact Map introduced in *Chapter 8*, *Discovering the Why and Who*, generated a lot of hypothesis statements. For example, in our PetBattle case study, we hypothesized that creating the Daily Tournament feature would increase site engagement by uploaders and this would help achieve our overall goal to generate 100K in revenue

with our existing and new customer base by the end of the year. Well, we've built that feature now. In fact, we created an A/B test in *Chapter 11*, *The Options Pivot*, and designed an experiment about how and when the next tournament should begin. What did the results tell us? Did we achieve the impact of more site engagement? Did this hypothesis prove true or false? Let's update the Impact Map.

We met Mary, one of our users. We ran the Empathy Mapping practice and other human-centered design techniques from *Chapter 8*, *Discovering the Why and Who*. We heard from her how much she likes the top three leaderboards. We first did some user prototyping and testing with Mary. We could do another Empathy Map on the latest working software of PetBattle to capture the latest information on what she thinks, sees, hears, and says about the latest increment of the software. Let's update or create new Empathy Maps with Mary and other users.

We used Event Storming to get us just enough understanding of the business flow for the one where Mary enters the daily tournament and wins a prize. There were a number of pink square sticky notes on the Event Storm, which represented assumptions, questions, and unknowns. We know much more information now that we have delivered some features, conducted some research, run some experiments, and developed the conversation. We can update the Event Storm and perhaps even start to generate a whole new part of the system or area of functionality.

Usually, we only ever have three or four Target Outcomes. Perhaps we've now met them or are close to meeting them; our learnings and measures may warrant us rethinking or rewriting Target Outcomes. Maybe we need to think about the next Target Outcomes to take our application to the next level and stay ahead of the competition.

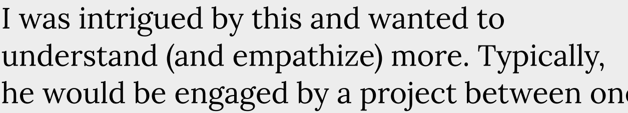
Our User Story Map, Value Slice Board, and prioritization practices were introduced in *Chapter 11*, *The Options Pivot*. We have now delivered items in the top slice or slices of value. Our learning may have triggered us to rethink existing priorities or re-slice and re-plan value delivery. Let's update these artifacts to reflect our latest viewpoints.

Our Product Backlog is always ready for more Product Backlog Refinement. With all of the updates to the Discovery and Options Pivot artifacts, it's sure to need another look and update. Let's refine the Product Backlog.

So, all the artifacts we've produced from all the practices we have used are never done—they are living breathing artifacts. They should always be visible and accessible by team members and stakeholders. The more often we update these based on measures and learning, the more valuable our top Product Backlog items will be, and the more responsive and reactive our products can be to user, market, and stakeholder needs. This is known as **business agility**.

Learning from Security Experts

There's a lot of debate in tech and Agile communities about how security fits into Agile processes. I recall having a long conversation with the security controller of an Irish telecommunications company. He said, "*Agile and security do not mix and they never will.*"



I was intrigued by this and wanted to understand (and empathize) more. Typically, he would be engaged by a project between one and four weeks before the go-live of a new system or application. He would scrutinize every framework used, the logical architecture, the physical architecture, hosting, and data storage, and would eventually run a series of security penetration tests. Often, he would have to work nights and weekends because his security sign-off was one of the last milestones before the commercial launch of the associated product.

We were determined to try and *shift left* this security analysis and the checks involved, to integrate the security controller into the process and make the penetration testing more continuous and automated.

We started by inviting the security controller to all Sprint Review showcase events. He didn't show up to the first few but eventually came to one. I don't think he expected to get much out of this, but he was one of the most vocal stakeholders in the room. He quizzed us about what versions of frameworks were being used in the code, whether cross-site scripting had been considered in the UI, and where we were using open source tools in our CI/CD. We didn't have all the answers, but we did capture items on the Product Backlog to conduct some research and set up some refinement workshops in the next sprint.

A couple of sprints later, we were demonstrating the outputs, outcomes, and learning from the research items captured from earlier feedback. We were also showing how we had extended our CI/CD pipeline with automated code scanning and vulnerability checks—and we were researching automated solutions for cross-site scripting attacks.

The security controller realized this was not going to be one of those projects putting him under severe pressure days before go-live. The team was learning from him and he was injecting his needs and knowledge into automation. He would still have a week set aside to run his penetration tests but he felt much more confident. It turned out to be a very positive experience and one of the quickest security sign-offs ever done.

This experience highlighted the importance of increasing confidence in important stakeholders like security controllers and sharing learning continuously between the team and stakeholders. And, of course, confidence can be measured through practices such as confidence voting!

This story highlights the importance of always improving metrics and automation. If someone is carrying out the same activities (such as security tests) repeatedly, it is a candidate for automation.

Always Improve Metrics and Automation

We introduced a wide range of metrics in *Chapter 13*, *Measure and Learn*. This is not a one-time, finite list to follow as a guide, but an initial set of suggestions. Where we identify the need to measure based on stakeholder feedback, we should add that in, especially if we set it as an opportunity to increase learning or confidence.

Automating the collection, analysis, and presentation (or visualization) of metrics is what closes feedback loops to almost real time. Developers can get real-time feedback on the impact of code check-ins, while technical stakeholders receive real-time feedback, assurance, and confidence from the quality checks performed on built deliverables. Product Owners and business stakeholders, meanwhile, can receive such data on the usage and adoption of application features; and organizations on their overall ability and speed to deliver new features.

This links nicely to how we can visualize and quantify, through metrics, the full impact of continuous delivery infrastructure.

Revisiting the Metrics-Based Process Map

Back in *Chapter 10*, *Setting Outcomes*, we introduced the Metrics-Based Process Mapping practice as a tool for discovering the case for continuous delivery. We have used this practice many times with many different organizations. The metrics produced after adopting new practices, technology, and culture are mind-blowing.

More recently, we've chosen not to use the practice on the first iteration of Discovery if we're working with a brand-new team. While it may appear to be a great way to capture the metrics of legacy processes, we've learned that new teams do not have the foundation of culture or psychological safety to engage in this activity. You have to ask very probing questions like *How long does it take you to do this process*? and *How often do you make mistakes or inaccuracies that are found in later tasks, meaning you need to do some rework on your bit*? Without safety in the culture, they can result in misleading answers and even damage the culture. However, we have found it an awesome practice to run after passing through some Delivery Loops.

My management only really understand numbers and spreadsheets

When working with a European automotive company, I ran daily Agile coaching sessions with the Product Owner and ScrumMaster. This was an "ask me anything" type of session, which is often a great opportunity to chat about ways of working and generate different ideas to experiment with in the future.

In the penultimate week of our engagement, the Product Owner said how happy he was with the team and the product that had evolved over the five preceding weeks. He loved the information radiators and visualization; he loved seeing the product come to life with Showcases every week and how the team was meeting real end users every week to test, experiment, and validate the software and approach. The technology was awesome and what the team was doing with CI/CD and other automation was just mind-blowing!

But he was also concerned. This residency engagement had been run in a pop-up off-site facility. He had brought some of his peers in to "walk the

walls" and see the way of working in action, but much of his leadership had not seen it. He explained how his management only really understood numbers and spreadsheets–they weren't really convinced by the use of sticky notes and colorful walls. How could he prove to them that this was truly a better way of working?

This made me think of a practice in our Open Practice Library we had not yet used: Metrics-Based Process Mapping. This was exactly the tool to use for this particular problem. So, we got some sticky notes and a huge movable board.

Figure 17.2: Using a Metrics-Based Process Map for an European automotive company

We captured all the things that used to happen between their business or users requesting a feature and that feature running in production. We captured the Lead Time, Process Time, and Completeness and Accuracy metrics all as per the practice described in *Chapter 9*, *Discovering the How*.

We captured these for the old way they used to deliver increments of their software and the new way. For the old way, they trawled through some ticketing systems to help them get metrics. For the new way, they collected metrics from Jenkins, GitHub, Ansible, and other tools used during the engagement. They also visualized the old and new structures of teams. The resulting board is shown in *Figure 17.3*:

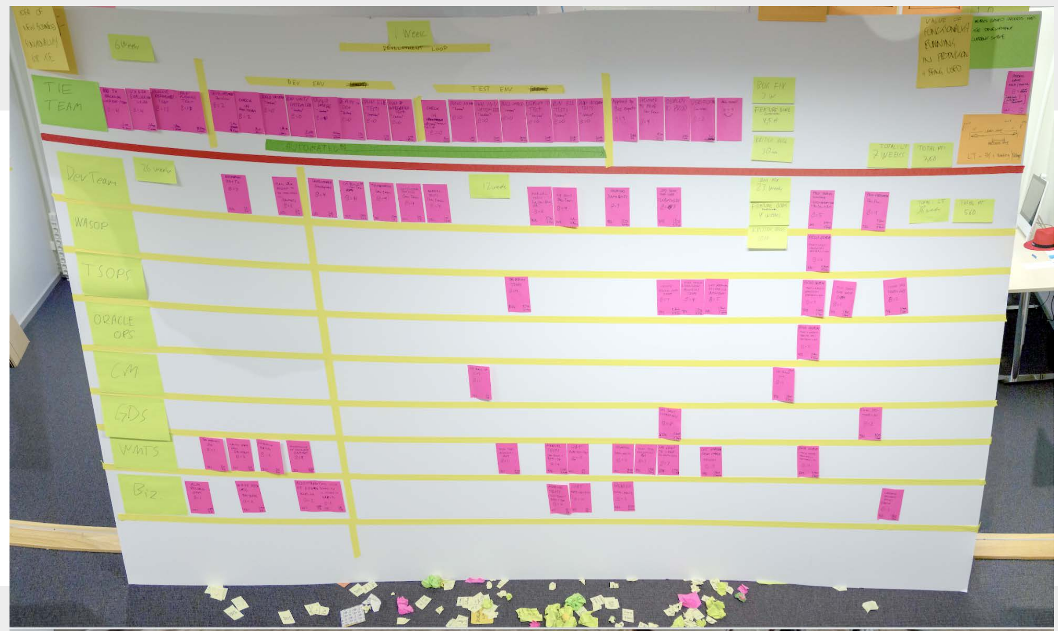


Figure 17.3: Capturing different metrics from the legacy approach and the new approach

The actual numbers and information on the sticky notes are not important for this book, but the process and resulting visualization shows some big learnings. Each pink sticky note represents a *thing* that happens between a feature request and the feature running in production.

Everything below the red line represents the old legacy approach to delivering features. The tasks were completed by people in different departments and teams—each represented by the horizontal yellow lines.

As you read the pink sticky notes from left to right, every time you have to cross a yellow line to read the next sticky note to the right represents a handoff or a handover to another team. This often involved raising a ticket, booking a resource, reserving some time to communicate, and waiting or being in a queue.

Everything above the red line represents the new approach to delivering this software product that had been kick-started by our engagement. It's all on one horizontal line because it is a new, long-lived, cross-functional product team doing all the work. Infrastructure, design, development, testing, deployment, and operations were all performed by this new product team.

The green line represents where the pink task immediately above has been automated. This is CI/CD in action! The pipeline has automated code scanning, builds, containerization, deployments, unit tests, end-to-end tests, contract tests, vulnerability, security tests, and UI tests. As we summed up the old ways and new ways and considered a few nuances for different types of development work this team would undertake, the measurable impact of continuous delivery was staggering. This was shown to a group of senior leaders, which brought a few gasps.

Total lead time from idea to first delivery to users	Prioritized bug	Functionality done confirmation	Critical bug fix
Current Way of Working	Current Way of Working	Current Way of Working	Current Way of Working
38 Weeks	23 Weeks	4 Weeks	10 Weeks
New Way of Working	New Way of Working	New Way of Working	New Way of Working
7 weeks	3 weeks	4.5 Hours	30 min
Improvement	Improvement	Improvement	Improvement
31 weeks reduction 5.4x faster	20 weeks reduction 7.6x faster	155 hours reduction 34x faster	9.5 hours reduction 20x faster

Table 17.1: Measuring the impact of the new way of working

One stakeholder asked what the main contributor to this shift in numbers was. The Product Owner explained that there were three things. Some of this was down to technology, including the OpenShift platform, being used. Some of this was down to the ways of working, including the Mobius Loop and Foundation practices that had been employed. But, most of all, it was because of this team. This long-lived product team had proven that they could not only achieve these metrics but would go on to improve them further.

The three contributing factors in the preceding story are the things we should strive to continuously improve by continuous learning: improve the technology, improve the ways of working, and improve the team.

Most of this chapter has been about the ways of working and how we use the Mobius Loop model to promote continuous improvement. Let's now consider the tech and then the team.

Improve the Technology

The challenge with writing any book about technology is that it can very quickly become obsolete—just ask two of the authors who wrote *DevOps with OpenShift* a few years before writing this. Many of the patterns, technologies, and products have evolved since the writing of *DevOps with OpenShift*, and we fully expect that to continue in the coming years.

Section 6, *Build It*, *Run It*, *Own It*, was focused on the PetBattle technology and how the team would build, run, and own it in 2021. If you're reading this book in 2025, say, we are hopeful that many of the chapters in this book will still hold true, but *Section 6* may well have moved on.

It's vital that teams are given ample time, space, and capacity to learn new technologies and experiment, research, and implement new products and frameworks to continuously improve the technology in the solutions they build.

For a long-lived product team, this highlights the importance of great Product Ownership and well-refined Product Backlogs that can correctly articulate the value of all work items. This includes non-functional work to architectures to improve and evolve them, while also refreshing them with new technologies.

Some of the approaches we use in Red Hat that we have encouraged our customers to use include the following:

- Allocating time to technology, research, and experimentation
- Strong regular investment in technology, training, and, where appropriate, certification
- Communities of practice and interest around a technology
- Establishing technical book clubs
- Organizing and hosting regular webinars, lunch and learns, and other community events
- Attending external technical conferences
- Attending internal conferences (for example, Red Hat runs a one-week internal Tech Exchange event every year for all its services and technical pre-sales staff)

Above all, listen and learn from the team members and reserve time in each iteration to promote learning. Continuous investment in team learning is one of the strongest success factors of a high-performing, long-lived team.

Long Live the Team

From the outset of this book, we have promoted the importance of long-lived, cross-functional teams. In *Chapter 1*, *Introduction – Start with Why*, we led with how we want this book to help the I-shaped individual (specialists in one particular skill) to become more T-shaped (multi-functional knowledge and single skill depth), or even M-shaped (multi-functional knowledge and multiple depth skills).

Above, we explored some ways we can improve the technology. Those suggestions weren't really directly improving the tech. They were focused on the people using the technology and improving their abilities to use technology. How can we measure and learn the impact of improving team skills? What are some of the ways we can visualize this?

Visualizing the Transition from I to T to M

A practice we've been improving over recent years has been capturing, measuring, and visualizing skills in a team. When we form a new team, for example, when we start one of our residency engagements, it's really helpful to understand individuals' skill sets and experience. But we want to do this in a psychologically safe way and not have new colleagues feel threatened or that they are under examination. So, we've stopped running tech-tests as it was affecting team morale. Instead, we use a simple visual such as a spider chart to visualize the skill levels (0-5) for the different capabilities needed in the cross-functional team:

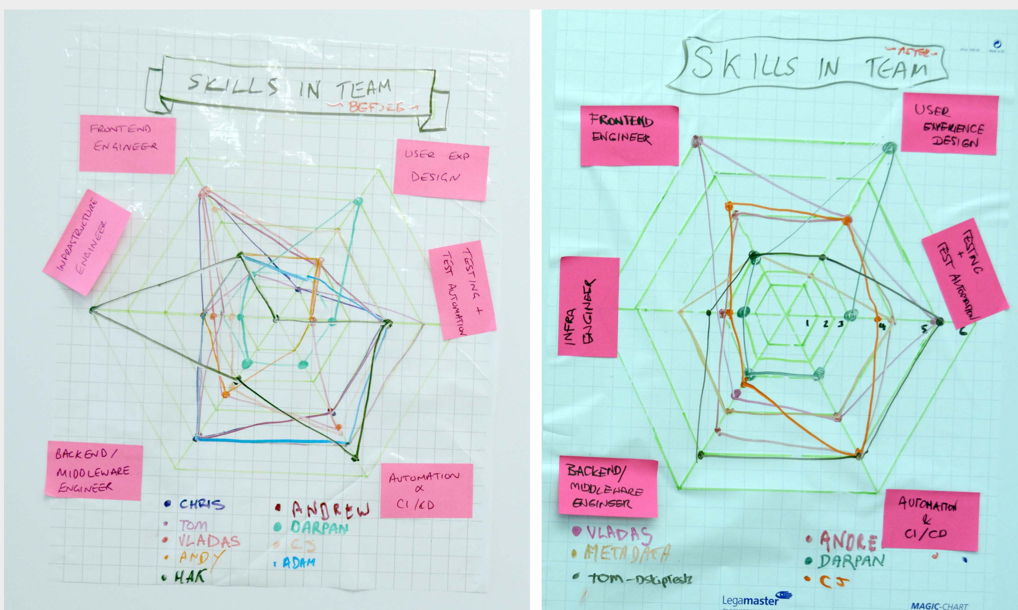


Figure 17.4: Visualizing the skills in the team using spider charts

This can, of course, be done using simple spreadsheet software:

BEFORE

	User Experience	GitOps	Back End	CI/CD	Testing and Test Automation	Automation	Infrastructure and Containers	Ways of Working (Agile, Lean)
Donal	3	4	4	5	5	4	4	5
Cansu	3	4	2	4	3	4	5	5
Fraser	3	3	5	4	4	2	3	4
Marcus	4	4	4	5	4	5	5	5
Jou	1	4	2	4	4	5	5	3
Onur	3	2	3	2	6	3	2	4
Praveen	4	3	6	3	4	4	3	5
Tim	3	2	5	2	2	2	4	3
Mike	4	3	6	4	4	4	4	5
KV	2	2	2	2	2	2	2	3
Total	30	31	39	35	39	34	37	44

AFTER

	User Experience	GitOps	Back End	CI/CD	Testing and Test Automation	Automation	Infrastructure and Containers	Ways of Working (Agile, Lean)
	3	5	4	5	5	5	5	5
	3	5	3	5	4	4	5	5
	3	4	5	4	3	4	4	4
	4	5	5	5	5	5	5	5
	2	4	2	4	4	5	5	4
	3	4	3	3	6	4	3	4
	4	4	6	5	4	4	5	4
	3	3	5	3	3	3	4	5
	5	4	6	5	5	5	5	4
	3	4	4	4	3	4	4	5
	33	43	43	43	42	44	44	47

Figure 17.5: Using spreadsheets to analyze team skills

It can also be visualized using different kinds of charts:

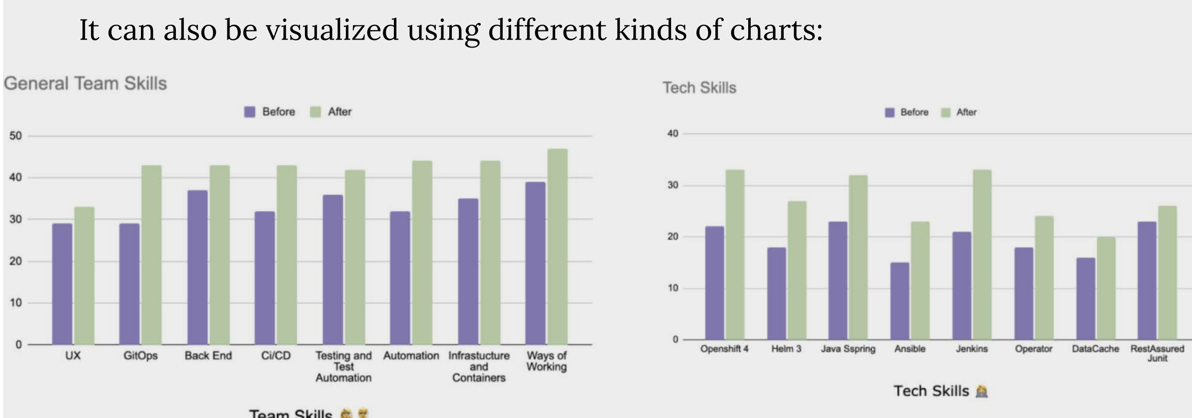


Figure 17.6: Using histograms to analyze team skills

As a long-lived team, this is a great practice to get into the habit of using. The team can regularly inspect whether there is team learning occurring and whether the team and individuals are becoming more cross-functional.

Wizards and Cowboys

Another fun learning culture practice we've used recently is the analogy of wizards and cowboys. In *Chapter 6*, *Open Technical Practices – Beginnings*, *Starting Right*, we explained why we are trying to eradicate the Unicorn Developer. We do not want to have heroes in the team that possess all the knowledge and are highly depended upon.

When working with a geo-spatial company, we noticed in a Retrospective that there was a risk that the individuals were learning heroic things themselves and knowledge was not being shared. The team agreed to buy a wizard's hat and that, when these incidents happened, the person who had achieved something amazing would wear the wizard's hat until the knowledge had been shared in a pair or, even better, in a mobbing session.

> The cowboy hat was also used to highlight anti-patterns and things that people shouldn't have done (such as applying `-DskipTests=true` to get a build deployed without running automated tests). If such behaviors were spotted, whoever was responsible would have to wear the cowboy's hat until the anti-pattern had been shared with the whole team and they had been educated.

It's a bit of fun and creates a safe space for better team culture. It's also educational and promotes continuous learning.

Conclusion

In this chapter, we explored what to do when we come out of a Delivery Loop iteration to ensure we improve as we either go round the Delivery Loop again, return to the Options Pivot, or go back to the Discovery Loop. We looked at how we take metrics and learning from Delivery, assessing what we learned and whether this is enough to decide the next steps to be taken.

We are now operating in a continuous flow of innovation from Discovery to Delivery and back again. We started with *just enough* and *just in time* information to get going. We're learning all the time.

We looked at how we can measure the improvements in our new system and ways of working by returning to the Metrics-Based Process Mapping practice to quantify improvements in the technology, ways of working, and the team. We recognized the importance of continuous learning and continuous improvement in all of these.

In the final chapter of this book, we will look at ways to sustain everything we've covered in this book. There has been a great focus on one team and one dream. This has been intentional. As we look to sustain this, we will also see how we can re-use all of the patterns, approaches, and practices used throughout this book to grow, mature, and scale a product-centric mindset to applications and platforms—even to leadership and strategy.

18

Sustain It

In the previous chapter, we transitioned from a single trip around the Mobius Loop, covered from *Chapter 8*, *Discovering the Why and Who*, through *Chapter 13*, *Measure and Learn*, into an ongoing, never-ending, continuous journey of learning and improvement informed by metric-based outcomes.

This book has focused on how a team can travel round the Mobius Loop from Discovery through Options and Delivery and back to Discovery again. The Open Practice Library has helped instantiate Mobius into a true outcome-based delivery built on a foundation of open culture and open technology.

The PetBattle case study was all about one team achieving one dream to be a high-performing, cross-functional, long-lived unit, delivering an awesome product and getting the very best out of the underlying technology, including OpenShift. Through the use of people, processes, and technology, they are a seed of inspiration and a trigger of infectious enthusiasm.

Mobius is more than just a map for a team to follow; it is the navigator for the whole system of working.

In this chapter, we are going to ask, *how can we use Mobius and open practices to drive this mindset of continuous discovery, continuous delivery, and Options Pivoting on a foundation of culture and technology throughout a much larger organization*? In other words, how can we get 50 teams working together like the PetBattle team? We will explore how we can enable more Application Product Teams to work the same way, what it means for the platform, and what it means for the leadership and organizational strategy.

Let's start by visualizing the journey undertaken so far.

The Journey So Far

Let's use another open source artifact available in the Mobius Kit[1] called the Mobius Navigator. It provides a macro perspective and underlying philosophy behind Mobius that can be applied to any complex system. The Mobius Navigator is a mental model that can be applied to almost anything, at all levels of an organization.

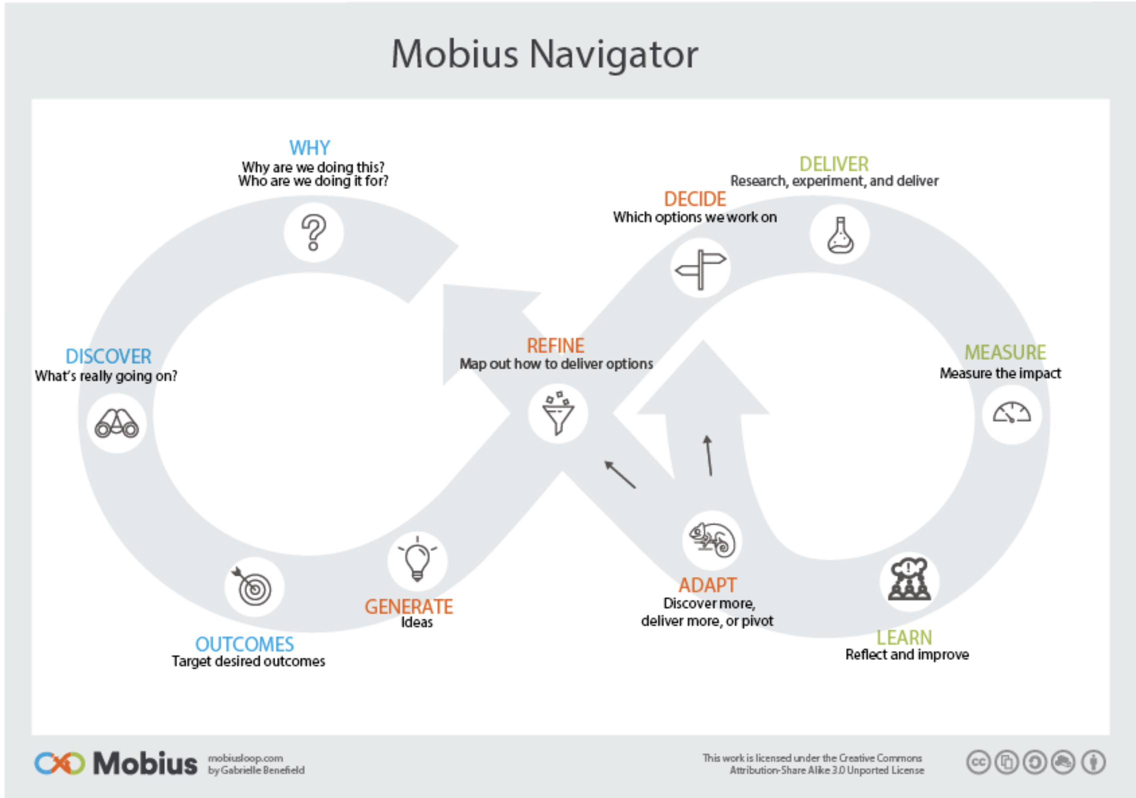


Figure 18.1: The Mobius Navigator

1 https://www.mobiusloop.com/kit/

We typically start on the left loop, although we don't have to. Sometimes legacy systems and organizations will start amidst delivery on the right loop and will seek to pivot over to the left as soon as is feasible. This happens because there are important, powerful questions that need to be asked about Discovery, and the sooner the organizations can get back to asking those questions, the better.

As we work our way around the Mobius Navigator from the left, we ask the powerful *Why* questions. Why are we doing this? Who are we doing this for, and why will it help them? We discover what's really going on to come up with outcomes. We then generate ideas, refine and map out how to deliver options, and decide which options we're going to work on next. We deliver by designing, creating, and launching, measuring the impact as we learn by reflecting and improving. Finally, we adapt by discovering more, delivering more, or pivoting completely.

The Mobius Loop allows us to take the Mobius Navigator mindset and drill down into, for example, a product team and how they use Discover Maps, Options Maps, and Deliver Maps to continuously evolve products, delivering outcomes that matter with continuous measurements and learning built in. We saw this in action in PetBattle and with the many stories dotted throughout this book.

We used a Discover Map at the end of *Chapter 10*, *Setting Outcomes*, to summarize *Section 3*, *Discover It*, and capture all the learning harnessed during the Discovery Loop. We used an Options Map at the end of *Chapter 11*, *The Options Pivot*, to summarize *Section 4*, *Prioritize It*, and capture all the learning and decisions made during the Options Pivot. We used a Delivery Map at the end of *Chapter 13*, *Measure and Learn*, to summarize *Section 5*, *Deliver It*, and capture all the delivered impacts and learnings made during the Delivery Loop. If we connect these maps together, we see how the Mobius Loop is used to connect these three sections together and summarize the learning of product evolution.

There is a lot of detail in *Figure 18.2*, and you can examine individual maps either at the end of the chapters outlined above or access PDF versions in this book's GitHub repository.

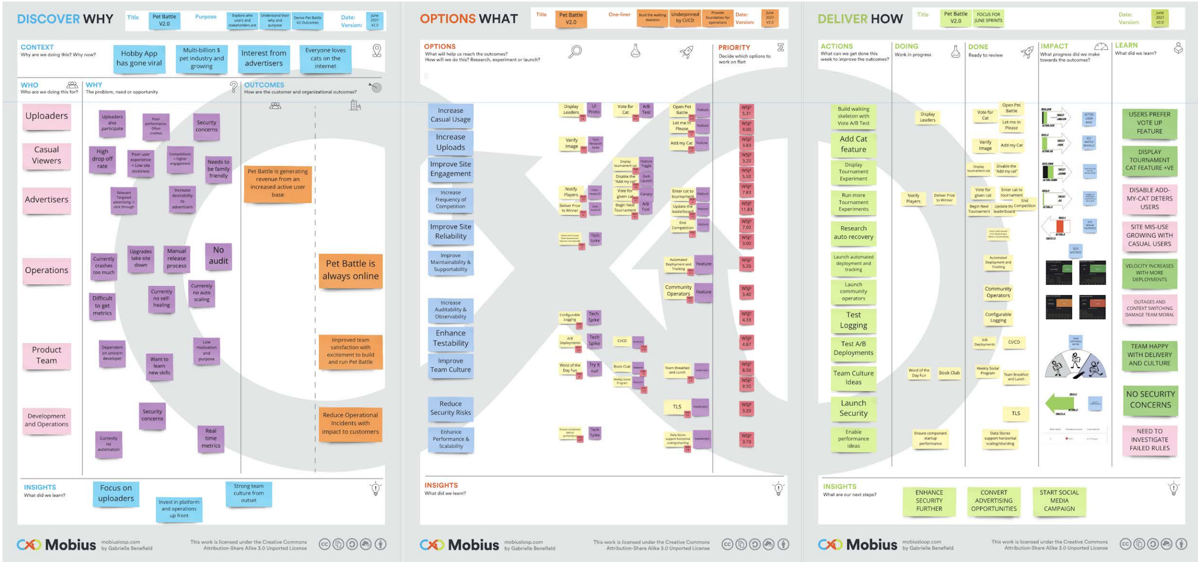


Figure 18.2: PetBattle Mobius maps: Discover, Options, Deliver

Now, what if we had 10 different product teams in an organization working this way, each with their own set of maps for a product or product area they were responsible for?

This idea becomes plausible through infectious enthusiasm, increased platform adoption, and an enhanced product mindset at different levels throughout the organization.

A side note, but the majority of this book was written during the COVID-19 pandemic when **infection** and **spreading** were very negative phenomena. It's nice to be writing about a much more positive impact of something being infectious!

Infectious Enthusiasm

One of the reasons we advocate starting with one team is to test, learn, inspect, and adapt with a small slice of the organization or department. Starting the right way with a single team and showing rapid success can be incredibly powerful before any kind of spread starts. This team is our Patient Zero.[2] By taking this one-team approach first, we are actually taking an Agile approach to moving toward a more Agile way of working.

2 https://en.wikipedia.org/wiki/Index_case

With that first team, we want to enable them to learn as fast as possible by doing everything described in this book and by adopting practices in the Open Practice Library. No training course, YouTube video, or book will ever, on its own, enable a team to instantly work in the way envisioned and achieve the outcomes anticipated by this book. We hope you've got a lot of value from this book and are armed with inspiration and information, but we don't for one moment think your team will instantly transition to this mode of working, even if they read the whole thing cover to cover.

People and teams can be enabled by going through the actual journey for themselves. They have to build their own foundation of culture and technology. They have to travel around the Discovery Loop, Options Pivot, and Delivery Loop to experience the Mobius Navigator mindset themselves. In doing this, they need an application, a system, something that delivers real business value to real end users, so they have the opportunity to Build It, Run It, Own It themselves. We advise starting with something that has tangible business value but is also relatively small, so that you can experiment and learn with one small team to start with.

Getting one application into production using all the practices and principles described in this book is a challenge and your team will almost certainly identify bottlenecks and blockers throughout the process. The great news is that the Navigator and mental model is built to address this.

Being able to improve in the way described in *Chapter 17*, *Improve It*, enables these bottlenecks and blockers to be addressed, one by one, and validated through metrics. Many of the underlying blockers come from silos in the organization that need to be broken down. The product team, through autonomy, mastery, and purpose, need to be able to silo-bust their way all the way to production!

The challenges being surfaced by such a team on a mission to show this new mode of operation need timely resolution, either through platform evolution or leadership enablement. For example, if there is poor or no Product Ownership in place, this blocker needs to be raised through a Retrospective event (as explained in *Chapter 12*, *Doing Delivery*) or the use of the Real-Time Retrospective or even a Stop-the-World event (practices described in *Chapter 4*, *Open Culture*). It's best to identify these problems early on and address them with one team, rather than accepting what we know is sub-optimal and end up scaling inferior ways of working across the organization.

Right at the beginning of this book, we explained how we have a saying that we like to *show*, *not tell*. Show, not tell, is how we create infectious enthusiasm. We have to show the world the people, the process, and the technology working together in full glory. Let's look at a few ways we do this.

Demo Day

In Red Hat, when we run Open Innovation Labs residencies (4-12 immersive engagements where our customers experience culture and practices on a real business use case), we always finish the engagement with a Demo Day event. Of course, we run demos throughout the engagement. We typically use Scrum (as described in *Chapter 12, Doing Delivery*), which means we run a Sprint Review showcase event every week (we tend to run one-week sprints).

While we encourage the idea of **inviting the world** to these weekly events, practically it is not possible for every developer, operator, leader, engineer, and anyone else connected in the organization to attend. We do record them and try other mediums to get the word out about these weekly showcases, but we find we need an extra-special Demo Day event.

Demo Day is a showcase of the entire way of working–the people, the processes, and the technology. In our residency program, it's the people from our customer's organization (also known as the **residents**) who deliver nearly all of the Demo Day showcase. By this stage, the infectious enthusiasm has already spread to them and this is their opportunity to show (not tell) everything the residents from our customer's organization have done to build, run, and own the product that's been the focus of their residency engagement. This is an opportunity to show their peers and colleagues what they've built, run, and owned; but more importantly, it is an opportunity to show the *how* of these stages.

This is a showcase of all of the practices the residents used in this time-boxed period of working together, including:

- Their cultural foundation and how they've used practices such as social contracts, Stop-the-World events and Real-Time Retrospectives. They show the team identity they've achieved, how they use team sentiment tools, and how they have created an awesome, thriving team workspace. They explain and show how they used tools such as priority sliders, Definitions of Ready and Done, and acceptance criteria to drive easy conversation, aligned and shared understanding, and a strong open culture.
- Their technical foundation and how they use containers and the OpenShift platform as the root of their technical foundation. Adding technical practices such as Everything as Code, test-driven development, test automation, CI/CD, and continuous deployment is what really provides a strong platform and foundation for continuous delivery. They show the value they get from pair programming and mob programming to achieve team collective ownership of the product. They show how they have achieved autonomy, mastery, and purpose from this foundation.

- They walk the walls and explain the Impact Maps, Empathy Maps, Event Storms, Non-Functional Maps, and Metrics-Based Process Maps and how they collectively drove the target measurable outcomes to take the organization toward its North Star.
- They show how prioritization worked during sprints using User Story Maps, Value Slice Boards, and other prioritization practices they chose to use.
- They show how the product has evolved, how the architecture has emerged, and the delivery practices used to iteratively and incrementally deliver.
- They demo the app they've built. They demo the platform capabilities being utilized and show some of the platform features they've utilized to improve operations, drive a more experimental approach to feature development, and radiate real-time metrics.
- They give some of the mind-blowing metrics that this technology, these practices, and this team have managed to achieve and how they compare to the legacy, status quo way of working.
- They share directly with their leadership what they need to sustain this way of working and what their hopes and fears are when they return to their normal place of work.
- They celebrate with pride what they've achieved and what, as a team, they've become.

The infectious enthusiasm is hard to ignore when people are sharing with such passion and energy what they've managed to achieve with the technology and practices they've adopted. Even the most introverted people have delivered incredible showcases at Demo Day. It's something you cannot write down or explain. You need to be there to feel and experience the energy and excitement.

Another aspect we often try to build into Demo Day is to show a typical day in the life or sprint in the life of the team. To do this, we create lots of videos during our engagements.

Documenting the Journey

It's hard to articulate in writing or even verbally the power of collaboration. When a team is Event Storming and has a strong open culture in place, it's just amazing to watch how the conversations align. When a team comes together to prioritize and refine Product Backlogs, the business and technical clarifications achieved and the decisions made are very powerful, especially when compared to the status quo approach to planning work. All the fun, energy, and banter that gives a team such awesome culture cannot be communicated in words.

So, we try to capture that as much as possible through photos and videos. It's why we've included lots of photos in this book to try and show, not tell, the culture and these practices in operation.

We encourage everyone to work in or near teams to photograph and video everything (with permission of those around, of course!) as much as possible. Even in a four-week engagement, we end up with literally hundreds of photos and videos.

To create infectious enthusiasm, members of our team tend to compile a weekly showcase video. This isn't a professionally made video but something fun, a couple of minutes with a montage of photos and short videos that just gives others a glimpse of what it feels like in the team space. We show these to the teams (normally on a Friday afternoon), which provides a sense of pride and reward. We get feedback that team members show these videos to their families, which tends to be a good sign that they are loving what they are doing!

We encourage our customers to share these videos and photos around their wider organization, especially with those who may benefit or be inspired to see this one team working this way and may not have a chance to visit them. Some even end up on YouTube,[3] which is a great way to show the world!

While photos and videos provide a glimpse of a team over a period of time, we also look for creative ways to summarize a longer period of time or an entire engagement. One approach that has worked really well is to sketch the experience.

---

3 https://youtube.com/playlist?list=PLnqlDDtSH2A4e3dTIGHGyhCYmwIoaYxV5

Sketching the Experience

Hopefully, you've noticed the value that sketching and doodling can bring to areas of complexity. Ilaria Doria has provided many amazing sketches throughout this book to bring the content to life a little bit more and show, not tell, the practice.

This is another approach that works really well at telling a story and documenting an experience. We have sometimes engaged graphic artists to document Demo Day as seen in *Figure* 18.3. There's a lot of detail in the image that may not all be readable in print. You can, of course, access the digital version using the book's GitHub repository at https://github.com/PacktPublishing/DevOps-Culture-and-Practice-with-OpenShift/ and zoom in.

There is also an excellent three-part series of blog posts[4] all about this particular engagement written by Eric Schabell that also includes the sketch.

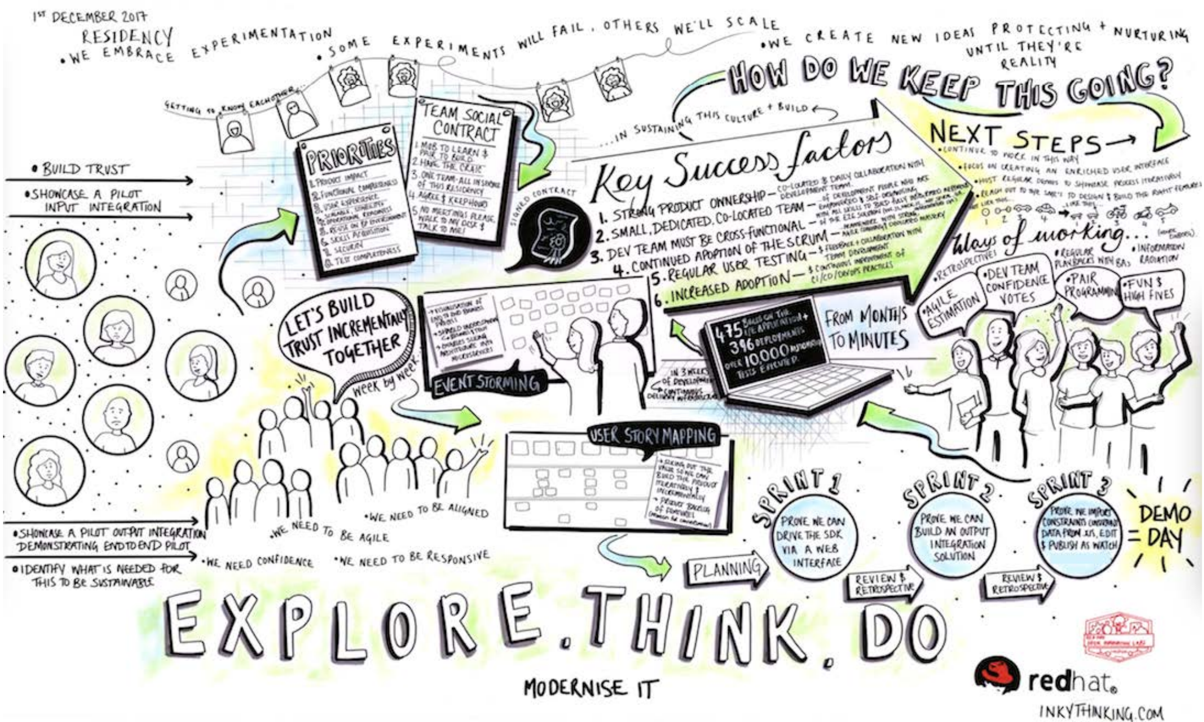


Figure 18.3: Example sketch of an Open Innovation Labs residency journey

4 https://www.schabell.org/2018/01/inside-my-open-innovation-labs-residency-part-1.html

We have also incrementally built up the pictures of these people, processes, and technology in action over the course of several weeks.

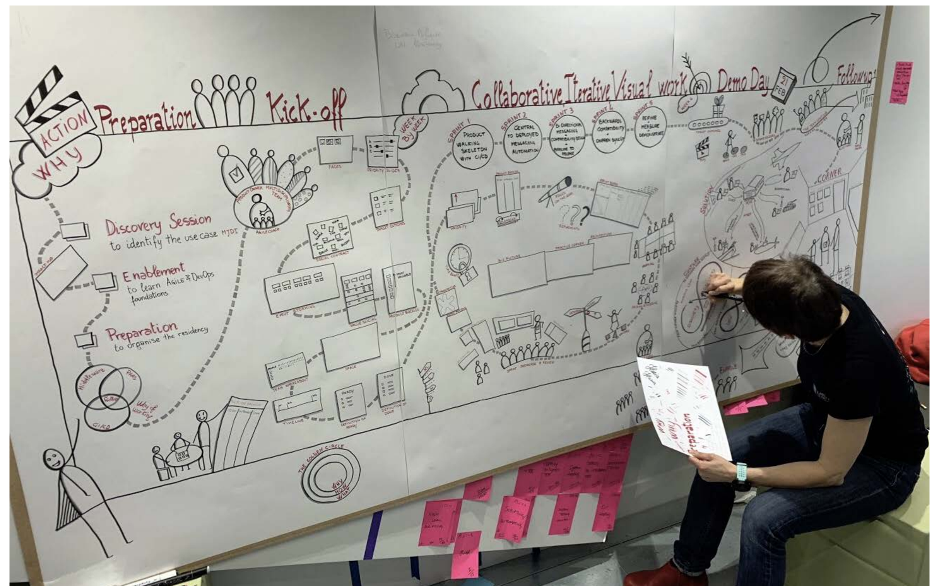


Figure 18.4: An Engagement Leader with visual facilitation skills

These sketches have hung proudly in public areas of the organization's headquarters. They stimulate interest and provide inspiration and another source of infectious enthusiasm. These visuals often encompass everything you might see if you came on a walk-the-walls tour of the team space.

Walk the Walls

One of the very best mechanisms we have found to create infectious enthusiasm is through a practice called walk the walls. In *Chapter 4*, *Open Culture*, we discussed the tremendous value we get from visualizing work and creating the many Information Radiators we have learned about in this book. In *Chapter 5*, *Open Environment and Open Leadership*, we explained the important success factors of an organization's leadership team giving teams open workspaces to show and use all of these information radiators. Walk the walls brings both of these together and delivers a truly engaging and immersive experience to anyone that might be interested in seeing this way of working happen with their own eyes. The idea is that people going on a walk-the-walls tour should be able to get all of the information they would ever want or need about the team, its product, the technology being used, and their way of working. Simply looking at the information radiator artifacts and having a conversation with whoever is giving the tour (usually a member in the team or someone close to it) provides an honest, energetic, and humbling experience to learn about the people, processes, and technology at play.

Walk-the-walls tours are not rehearsed and do not follow a script. We simply talk about what we see and tell stories about how we built what we're seeing: the Big Picture, the Practice Corner, Impact Maps, Event Storms, the North Star, Metrics-Based Process Maps, Value Slice Boards, Product Backlogs, Sprint Boards, Retrospective results, build monitors, test metrics, deployment statistics, operational dashboards, and many, many more. A great walk-the-walls experience is more than simply **Show, Not Tell**—it's **Feel and Show, not Tell**. A high-performing team has a sense of autonomy, mastery, and purpose, uses great open cultural practices, and adopts the best technology available on a world-class platform. When you see such a team practicing all of this, it's hard not to feel the culture and energy in the room. That is what is truly infectious. Any teams-to-be walking the walls and feeling this energy tend to want in—and to be as this team is.

We have tried to bring to life a couple of walk-the-walls experiences for both an in-person product team[5] and a virtual product team.[6]

Where we can't have every team and every stakeholder visit the team and walk the walls with them, we look for other mediums to help show the world the team and technology in action through, for example, what we call **Written Showcases**.

Written Showcases

Written Showcases are exactly what they say they are—Showcase events but written down! We started doing these on a weekly basis where we would be frustrated that the weekly Sprint Review Showcase event would not be as well attended as we'd hoped for. We'd invited senior leaders, stakeholders, and other teams to come and walk the walls but, in a busy world, it's hard to get everyone to visit—especially in a time-boxed engagement of perhaps just a few weeks to enable and kick-start this team.

So, we write down all the things we want to tell our stakeholders in Sprint Review showcase events and walk the walls. And we show as much as possible through colorful photos, links, and montage videos.

These written Showcases often take the form of an HTML email or a PDF (so, all the photos and videos) that includes links to artifacts and information radiators and the latest metrics collected from the evolving application, software delivery, and platform.

It's difficult to give a written document the same level of visual appeal as an engaging and immersive walk-the-walls tour. But we can try, and the great advantage is that such documents can be shared far and wide in the organization and word of mouth becomes a trigger of infectious enthusiasm.

5 https://youtu.be/70y6SEz6bas

6 https://youtu.be/TtvDJIT6RYo

Word of Mouth

Word of mouth in the field has proven to us time and time again to be a key ingredient of sustaining and growing the culture that we are seeing with one team. After a few weeks of being invited to Sprint Review Showcase events and walk-the-walls tours, receiving written showcase emails, and seeing photos and videos of what looks like some awesome team and techie stuff happening, the team in question gets talked about.

We start to hear chats in the corridor such as, "*Have you seen those folks up on the 5th floor? Sticky notes and monitors everywhere…not sure what they're doing but it sure does look interesting*" or "*We need to go and see that product team who have been in the basement the last few weeks; they seem to be doing some awesome techie shizzles and I'm sure we can learn from them*" or even, "*I've just done this walk the walls thing with that team downstairs—they're doing all the stuff you read and learn about on training courses but they're doing it so much better and for real!*"

And the conversation goes on when you see some of the metrics around software delivery and operability. Word of mouth gets around about some of the amazing outcomes that one team is achieving. How can others not want to learn more?

Mind-Blowing Metrics That Cannot Be Ignored

In *Chapter 17*, *Improve It*, we shared a story about how a Product Owner needed to prove the value of DevOps using metrics to his Leadership Team and how he used Metrics-Based Process Mapping as a practice to enable this.

The headline metrics in *Table 17.1* showed improvements in getting new features into production over 34 times faster using a long-lived product team (as opposed to a project team), new ways of working (as opposed to the legacy Waterfall project management methodology), and a world-class container platform (as opposed to VMs).

The metrics do not lie. They are collected from the underlying systems and tools that have helped effect this change. When we showcase these metrics in addition to showing all the practices and demonstrations, we are not talking about cost savings. Instead, we are shifting the focus to value creation and value realized. It leaves stakeholders and leadership thinking about the impact this can have on their business if they can get the highest-level features, applications, and capabilities into the hands of users 34 times more quickly than what they are used to.

And this is just one team. What if all teams in the organization could work like this one? This one team has provided an inspirational blueprint for change by showing the art of the possible.

Transitioning From One Team To Seven Teams

Continuing the story from *Chapter 17*, *Improve It*, about the need for metrics to explain the benefits of DevOps Culture and Practices that they'd now experienced for six weeks, what happened next?

The team had been working in a pop-up lab for this residency engagement, so the first step was to move them back to their own offices.

And it was more than just moving the people. All the boards, all the artifacts, all the information radiators, all the monitors, all the sticky notes—they all had to be moved too!

We hired removal men to wrap them up and move them.

Figure 18.5: Moving boards and information radiators to show and inspire others

Through the support of the Leadership Team of this organization, a space was secured at the site where many of their IT teams worked from.

There had been a lot of word of mouth about this team. Written showcases had been shared weekly, which included an informal fun weekly video. Thousands of photos and short videos had been captured, which, along with all the artifacts created during the residency, were all shared back to the organization's own SharePoint servers. And, of course, those metrics were going viral around the organization!

Keeping the team together and keeping their tools (including all those boards) was the most important first step. They did this.

The space they worked in became known as The Board Room!

More walk-the-walls tours, more enablement and training sessions, and many more conversations with a growing number of other teams and parts of the organization followed.

The team still had a product to evolve further and deploy into more users' hands around the world.

About six months later, I received an email from the same Product Owner who we talked about in *Chapter 17*, *Improve It*, with the need for metrics. First, he talked about the app:

"*Yes, [the application] is live in production globally on all 89 markets. It's working quite well and aside from minor issues it's been running flawlessly.*"

He went on to explain how their CI/CD practices have continued to improve—"*We've continued to push code to production on a more or less daily basis; our record at the moment is 3 deploys within two hours and the pipeline execution time is down to about 20 minutes with a whopping total of 128 test cases part of the regression.*"

And, finally, he told me how this approach was organically scaling, through infectious enthusiasm to other teams—"*It's super exciting to be part of transforming a bigger part of the organization. I'm now responsible for 7 teams. Feels like we've hit the ground running and are making good progress even if we still have some ways to go.*"

One more thing. He gave me an updated version of the slide he'd previously used at Demo Day. The process improvement of getting new functionality done into production was no longer 34 times better than the legacy practices and legacy technology. It was now 105 times faster.

Total lead time from idea to first delivery to users	Prioritized bug	Functionality done confirmation	Critical bug fix
Current Way of Working	Current Way of Working	Current Way of Working	Current Way of Working
38 Weeks	23 Weeks	4 Weeks	10 Weeks
New Way of Working	New Way of Working	New Way of Working	New Way of Working
7 weeks	3 weeks	1.5 Hours	30 min
Improvement	Improvement	Improvement	Improvement
31 weeks reduction 5.4x faster	20 weeks reduction 7.6x faster	158 hours reduction 105x faster	9.5 hours reduction 20x faster

Table 18.1: Metrics-based benefits of using new DevOps Culture and Practices

This is a great example of continuous improvement!

What is great about this story is that the culture and practices were subject to immediate scaling. We saw the journey of ensuring the product team continued to perform and improve underpinned by the metrics. I believe several impediments and blockers had to be fixed by leadership to enable this. The proof point was scaling the team's application operationally across 89 markets while interest and infectious enthusiasm grew across the business.

And eventually, some of the key people involved in making that *one team, one dream* a success progressed to enabling seven teams to work, behave, and think in the same way.

More Teams, More Application Products

When we are ready to unleash that infectious enthusiasm and enable more Application Product Teams to work in the same mode as the first one has, what does that look like?

We've seen throughout this book how a team built its own foundation—cultural foundation, enabled by leadership, and a technical foundation. We've seen how that team journeyed around the Discovery Loop through the Options Pivot and onto the Delivery Loop, and based on learning either looped around Delivery again, returned to the Discovery Loop, or returned to the Options Pivot. This journey continues forever.

This is what five Application Product Teams' approaches would look like, as seen in *Figure 18.6*. Each of them is an autonomous team with mastery and purpose, focused on their individual product (or product area if they're delivering into a larger product). So, they each have their own Mobius Loop to journey around. They each have their own foundation of culture, their unique team culture. They each have their own technical foundation that they can continuously improve.

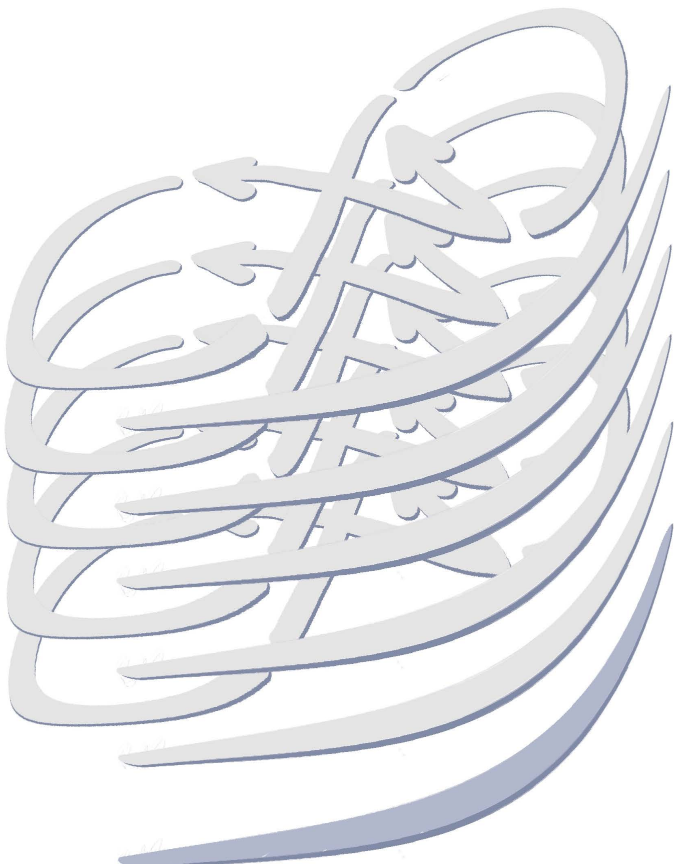


Figure 18.6: Lots of Mobius Loops representing lots of products in an organization

There is one addition. All the teams need to be supported by an underpinning foundation. This foundation is also made up of a cultural and a technical part. The cultural part enables teams to safely collaborate, share, and learn together using techniques such as inner-source and communities of practice. The technical part of the underpinning foundation is heavily contributed to by the platform, such as OpenShift, which underpins all the technical practices being applied by the adopting teams. How do we bring additional teams into this whole ecosystem? How do we add them to the platform?

First, every team needs to go through some kind of enablement. They need to build their own foundation of culture and technology. They need to understand their Why. They need to be involved in their own Discovery Loop and Delivery Loops. We strongly believe that there is a need for three iterations in any team's enablement.

The Power of Three Iterations in Enablement

It always seems to be three. When we do something new and then repeat it and then repeat it again, it's the third iteration where things start to click. Sometimes it takes longer and more iterations (especially where complexity is high), but three is the minimum number of iterations that any team should consider to go through an immersive, learn-by-doing, enablement experience. This is why our Open Innovation Labs residencies are always between 4 and 12 weeks in duration. A four-week residency means the team can take a few days to build their foundation, a few days to go round the Discovery Loop and Options Pivot, and then do three one-week Delivery Loop iterations using Scrum. Each iteration of the Delivery Loop will deliver an increment of working software that is potentially shippable to production. These iterations are kick-starting the continuous incremental delivery of the application product.

Now, this may not be enough to achieve outcomes and many customers will want to get back to Discovery and/or do more Delivery Loop iterations, hence six- and eight-week residencies are more common. But it's always in that third Delivery Loop or third Sprint where people really start to get it. They realize that one of these Showcases is going to happen every Wednesday at 3 PM. They realize that there'll be a written Showcase with a fun video on Friday afternoon. They realized that that will explain what has been loaded into the new sprint that was planned on Thursday mornings. It becomes a pattern and a rhythm that team members and stakeholders can feel increasingly confident about. And, most importantly, we start to see the impact of continuous improvement in metrics after three iterations.

The App Does Something This Week That It Didn't Do Last Week!

Working with a global oil customer in 2017, we used a four-week Open Innovation Labs residency to reimagine one of their pricing applications using DevOps Culture and Practices. Of course, this meant bringing the user into the process much earlier, and with much more collaboration.

I managed to get the time and support of an asset economist who was one of the key users of the application in question. She had never met a software development team, never mind worked with them, so this was a whole new world to her.

She didn't attend the Sprint 1 Showcase. She did attend the Sprint 2 Showcase and was hugely frustrated because so much of the functionality that she expected to see was missing from this replacement app that she relied on in her job. She, of course, had not really grasped that this was an early product increment and still very early in development. But she did provide feedback on a few things she saw demonstrated that were wrong.

The following week was the last Sprint Review showcase and she attended again. This time, she saw some of her feedback had been addressed and some (not all) of the new features were starting to go into the application.

The penny dropped and all became clear: she realized that the app was evolving with her involved. This week, it did stuff it didn't do last week. And the team had managed to take in her feedback and address it in less than a week! A very different experience to the usual user acceptance testing sign-offs she would use after several months of siloed development.

It was the third iteration where this positive realization took place.

So, we've established that every new team needs to go round the Mobius Loop at least three times to be engaged and immersed in using the platform and DevOps Culture and Practices. What about the foundation?

Bolster the Foundations

This is where we do *not* do a copy and paste of that first team's foundation. Teams need to build their own foundations. They also need to regularly nurture and bolster their own foundations. And where they are connected by culture and technology, they all need to contribute to their connected underlying foundation.

When we introduced the Open Practice Library in *Chapter 2*, *Introducing DevOps and Some Tools*, we explained that this is a community-driven repository of tools. Throughout the book, we've pulled out around 50 of our favorite tools in the Open Practice Library toolbox and explained how we use them together to deliver metrics-based outcomes that matter.

These are not necessarily the practices that every team should use. Just because a set of practices works so well for one team does not mean they will work equally well for every subsequent team. Practices used to build the foundation and navigate around the Mobius Loop need to emerge and react to the specific context and needs of the team at the time in question.

This means every team may mix up their practices compared to any other team.

To draw a comparison, let's think about what happens when your toilet breaks in your house. It can be a very stressful experience losing a piece of equipment so vital in the home! Assuming you're not an experienced toilet-fixer, you will probably call a plumber. Now, when the plumber arrives at your house, we suspect you don't greet the plumber by telling him or her what tools they're going to use and in what order and for how long. You expect the plumber to arrive with a van and/or a toolbox of everything he or she will need to investigate the problem, confirm the exact nature of the cause and what is needed to correct it, assess the options for making the fix (perhaps with some recommendations), make the fix, test it and ensure the problem is fixed and won't happen again, and clean up. The plumber will more than likely need a few tools to do this. You don't tell them what tools to use. You expect them, the experts, to do that!

It's the same with our product teams. This is why we shouldn't give teams a prescriptive, step-by-step set of tools and methods and expect them just to act on them. With autonomy, mastery, and purpose, we need them to pick the tools and practices they want to use and we expect them to arrive with the very best ones to get the job done to the best of their ability. And if a tool or practice isn't working out (either at first or after some time), throw it back in the toolbox and try another, a better one! Through the sharing of experience and forming communities across teams, organizations can find the very best tools and practices as informed by the teams on the ground.

As we explained in *Chapter 12*, *Doing Delivery*, we also need to look for when to bring the guardrails down that some of these practices provide. Just because a product team started using practices associated with the Scrum framework and worked really well for a few months, doesn't mean that these practices should be used forever for that team. Teams seeking to continuously improve need to inspect themselves and recognize when a practice they've been using successfully for some time might actually be counterproductive and a bottleneck.

Having a common underpinning foundation of culture and technology is what allows many teams to successfully travel around the Mobius Loop. This means creating a culture where teams learn from each other, and infectious enthusiasm continues to evolve—not just from the first team but by subsequent teams and organizations. Much of our experience comes from open source communities where sometimes many thousands of people connect to serve a common purpose and iteratively and incrementally deliver on this. We're seeing and helping organizations use the principles of open source to form communities inside organizations. The practice of inner-sourcing allows collaboration, contribution, and openness across the organization. They achieve this by forming Communities of Practice and Communities of Interest. **Communities of Practice** (**CoPs**) provide a space where people from different teams who hold particular specialist skillsets can collaborate, share, ideate, and inspire. Good examples include UI/UX CoPs, Product Ownership CoPs, or an OpenShift CoP. **Communities of Interest** (**CoIs**) can be on any topic of interest to anyone and related to something inside or outside the organization. Inner-sourcing and CoPs help to provide and bolster the underpinning foundation of culture that multiple teams can leverage sustainably and gain from.

Let's look at some ideas on how we might sustain technology.

Sustaining the Technology

Anyone who has a career in IT will tell you that one of the biggest challenges is keeping up to date with all of the technology. In the world today, a very large percentage of us are always connected to the internet.[7] The growth and numbers are staggering, especially when you consider that back at the start of the 1990s (when the authors of this book were leaving university), only ~5% of the developed world were connected to the internet. The pace of change is endless.

7 https://en.wikipedia.org/wiki/Global_Internet_usage

It is with this backdrop in mind that we evolve our applications and services atop a platform that has grown from its first commit in 2014, shown in *Figure* 18.7, to the most prevalent container orchestration platform today.

```
virt:~/git/kubernetes ⎇ master#5ad79eae2dc$ git log 2c4b3a562ce34cddc3f8218a2c4d11c7310e6d56
commit 2c4b3a562ce34cddc3f8218a2c4d11c7310e6d56
Author: Joe Beda <joe.github@bedafamily.com>
Date:   Fri Jun 6 16:40:48 2014 -0700

    First commit
```

Figure 18.7: First Kubernetes commit

For technologists, the challenge is keeping pace with this rate of change. To *not* change is one option. However, it can be a fatal option; think Betamax and cassette tapes and Perl (so sad)! We have said a few times in this book that we are optimizing a certain thing to minimize for the cost of change. We need to minimize the amount of time it takes to modify or change a software deployment pipeline, to enhance it, or to give us faster feedback. We need to minimize the time it takes to get ideas from our customers and business into working software solutions in production. We need to minimize the time it takes to update the platform. These drives to minimize time are based on the realization that *time is money* in business and organizations, and that it is often better to be *fast* than *right*. Gaining faster feedback allows teams to pivot and change faster based on their business and customer needs.

So, how can technologists sustain this? We do this by continually learning, evolving the long-lived team's shared knowledge and understanding of technology and solutions that are continuously evolving and adapting. Practices such as pair programming and mobbing bring to life this continual learning experience. It takes technical excellence that can be shared and understood by the team, combined with the people and processes practices described in this book, to win.

Businesses small and large are building their business platforms atop a myriad of vendor platforms. This platform mindset is critical in today's hybrid cloud. One of the main reasons for the success of the platform mindset is that platforms can become a place where shared and common best practices are codified among different teams. If the public cloud has taught us anything, it is that the right platform and infrastructure is a real sustainable competitive advantage. OpenShift is positioned to be the platform your organization and teams develop and continuously deploy business application services upon, wherever your infrastructure is deployed.

Lifecycle managing these platform architectures is a core skill that every organization has to deal with. Many of the intricacies are managed *as a service* in the hybrid cloud; this lowers the operational burden for your product teams. In OpenShift, with the operator pattern, all of the expert knowledge can be codified, shared, and updated over time to help automate and manage the individual pieces that make up the platform. Perhaps most importantly, the operator pattern was used to unify the operating system with the platform, making the OpenShift platform installation and life cycle far easier from a user's perspective. Practices have also evolved to take on this life cycle challenge. Site reliability engineering is a discipline born from traditional IT operations and infrastructure that codifies best practices for teams managing these platforms in all their possible guises.

To adopt a learning mentality is to accept that what you know today is not necessarily going to be enough for the changes and challenges that lie ahead. For us, innovation, research, and development happens in the open source communities of the world. You can tap into this and become part of this through commercial open source, which in turn becomes a competitive advantage to your organization. Commercial open source software projects are owned by a single company that derives a direct and significant revenue stream from the software through subscriptions, services, training, and support.

Being adaptive to change is critical in any DevOps or transformation agenda. So many times have we come across DevOps done wrong, and often they come down to these seven items. Watch out for these anti-patterns and know how to respond to them appropriately as, if you don't, they can lead you down a dark path:

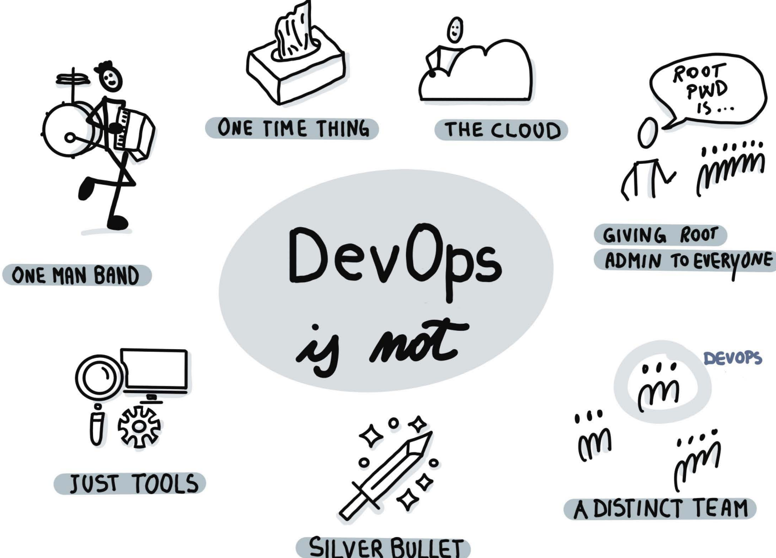


Figure 18.8: Seven anti-patterns and what DevOps is NOT

1. DevOps is **not** a one-person band: "*Yeah, we do DevOps, Mary is here Tuesdays and Thursdays and she manages our pipelines*." So often do we come across phrases like "*The DevOps person*," but this is an anti-pattern. DevOps is not just one team either; it's a desire to want to work a certain way. To want to take end-to-end responsibility for the software's journey and break down walls that impede feedback loops. It's not some gun for hire that swans in and fixes everything before saddling up and heading back out of town.
2. DevOps is **not** a one-time thing: Following on from this gun-for-hire analogy, DevOps is not a one-time thing. It's not just a matter of configuring a pipeline once and it's static forever. Like all technology, these things need to be nurtured and evolved. As new components get added to your architecture or new software becomes available, your pipeline and process need to evolve too.
3. DevOps is **not** the cloud: Using services from the cloud or just deploying a Kubernetes cluster does not mean you're now doing DevOps. It's how you use the platform as an enabler and view it as a product that can accelerate any transformational journey.
4. DevOps is **not** giving root/admin access to everyone: This one should feel cut and dry, but doing DevOps does not just mean giving the access to production and to the developers. It's about developers and site reliability engineers working together, listening to each other's concerns, and optimizing your delivery capabilities with trusted and reliable pipelines.
5. DevOps is **not** just the tools: Just because you're using Jenkins or Tekton does not mean you're doing DevOps. The tools are only a vehicle that can enable teams to work together on a shared platform and space. The real goal is creating ownership of the software stack and a feedback loop that connects users through to development to increase speed to market.
6. DevOps is **not** a silver bullet: It's all about *the people, the process, and the technology* combined and balanced. Doing DevOps is not going to solve all your problems; it's just one aspect.
7. DevOps is **not** a distinct team: If your **software development life cycle** (**SDLC**) consists of developers building code and then throwing it over the wall to the DevOps team to deploy/manage, then you need to reevaluate your SDLC.

Now, if the idea of *the DevOps Team* is such an anti-pattern, the question needs to be asked, "*Is the same true for the idea of a* **Platform Team**?"

Our first inclination may be that it is. If we truly want to bring down all silos and have a fully cross-functional product team, surely that should mean platform expertise is included in the team? Surely one or two of the T-shaped or M-shaped people we defined in *Chapter 1*, *Introduction – Start with Why*, should be specialists in the platform?

This has been a debate we've been having for several years, and perspectives do start to change when we start to consider many product teams all working on a common platform. This is a problem of scale. Of course, one of the goals of having a platform in the first place is to solve this problem of scaling and reuse. It provides an underpinning technical foundation for many teams to accelerate their development with. A single product team like the PetBattle team will get many benefits from a platform. But, if PetBattle were to get acquired by Pet Planet—a new Pet Tech firm with 50 product teams—the economies-of-scale benefits offered by the platform grow exponentially.

Recently, we've been inspired by and resonate heavily with the work of Matthew Skelton and Manuel Pais, who wrote the book *Team Topologies*. They introduce different team topologies, which include:

1. The Enabling Team, who help train, coach, and guide a team to learn new practices and acquire new skills, much like we do on our Open Innovation Labs residency program.
2. The Stream Aligned Team, which is a team aligned to a single, valuable stream of work. The Application Product Teams we have referred to throughout this book align closely to this.
3. The Platform Team, which has a purpose to enable stream-aligned teams to deliver work with substantial autonomy.

The reason we support the idea of a Platform Team and don't see it as an anti-pattern like the DevOps team is because the platform is a product and should be treated as a product. Where Skelton and Pais differentiate between Stream Aligned Teams and Platform Teams, we see both of these teams as Product Teams.

This book has been all about how Product Teams should operate and what is involved in taking a product mentality to a team's subject. We visualize any product mental model using the Mobius Loop and foundation.

The Double Mobius Loop – Platform and Application Product

So, how does the platform Mobius Loop work? Turn to *Page 1* of this book! It's exactly the same! We treat the platform as a Complex product, which means we perform continuous discovery, Options Pivoting, and continuous delivery.

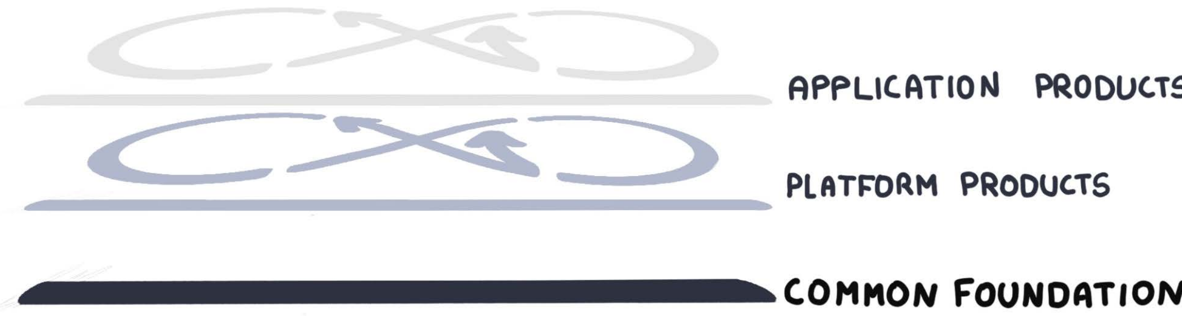


Figure 18.9: Double Mobius Loop of application products and platform products

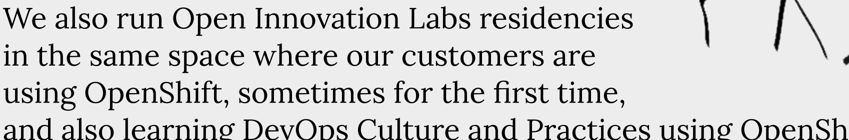
A Platform Team can use the same Discovery practices as we introduced in *Section 3, Discover It*. Rather than discovering a business application, they are using these practices to discover the platform. For example:

- A Platform Team can have its own North Star metric, as described in *Chapter 8, Discovering the Why and Who*. This gives a single metric that best captures the core value that the platform delivers to its customers or, more specifically, Application Product Teams.
- A Platform Team can have its own Impact Map, as described in *Chapter 8, Discovering the Why and Who*, deliverables that can be built into the platform, and the measurable impact we hypothesize these would have on actors (such as developers) to help achieve the platform's goal.
- A Platform Team can use Empathy Mapping and other human-centered design techniques to build empathy with, say, developers in Application Product Teams. This is very powerful in avoiding a common anti-pattern being mandated across the organization; it creates a pull from the development community who want a platform that will help them and solve pain points.
- A Platform Team can use Event Storming as described in *Chapter 9, Discovering the How*, to map out the processes developers use or want to use to get best use out of the platform. This enables the platform architecture to emerge. We can also use practices such as Non-Functional Mapping and Metrics-Based Process Mapping to trigger a continuous delivery approach with measures on the platform.

- A Platform Team can set target measurable outcomes as outlined in *Chapter 10*, *Setting Outcomes*.
- A Platform Team can organize and prioritize its work using practices such as User Story Mapping, Value Slicing, Impact and Effort Prioritization, How-Now-Wow Prioritization, and Weight-Shortest-Job-First to build a platform Product Backlog prioritized by value delivered to Application Product Teams. Design of experiments, design sprints, and even considering the advanced deployment concepts to roll out new platform capabilities can all be used by Platform Teams.
- Platforms are complex systems as defined by Cynefin and explained in *Chapter 12*, *Doing Delivery*. This means they benefit from being delivered iteratively and incrementally with learning and feedback built in. An Agile delivery approach such as Scrum or Kanban allows platforms to be showcased to developers and operators iteratively and allows metrics and learning to be collected.
- Platform Teams need to have their own foundation. Platform Teams need to have autonomy, mastery, and purpose themselves. They need their own technical foundation so they can apply practices such as Everything as Code, the Big Picture, and CI/CD as explained in *Section 2*, *Establishing the Foundation*. The technical foundation of the Platform Team is twice as important; not only does it provide a foundation for emerging platform development, but it morphs into the technical foundation of all Application Product Teams.
- Platform Teams can work in spaces full of information radiation, collaboration, and openness just like Application Product Teams. In fact, the most important collaboration is with those Application Product Teams.

How do Platform and Application Product Teams interconnect? How do their respective loops interconnect? Developers and other members of Application Product Teams are the users and the stakeholders of the Platform Team. The decision and motivation to use a platform has to come from the power of those users and not a top-down management decision. Platform Teams evolve the platform and infrastructure to serve needs and opportunities and address problems that developers and Application Product Teams are having.

Connecting Many Levels of Product Teams



One of the best feedback loops I witnessed between Platform Teams and Application Teams is in our Open Innovation Labs space in London, where we often have engineers from the Red Hat OpenShift business unit working in the space.

We also run Open Innovation Labs residencies in the same space where our customers are using OpenShift, sometimes for the first time, and also learning DevOps Culture and Practices using OpenShift.

Where there is an opportunity to connect OpenShift engineering to our customers, we seize it. There is a two-fold benefit. Our customers get to meet the engineers behind the core platform product they are using, can hear directly from them on upcoming features, roadmap items, and more; and our OpenShift engineers get to see real customers using their product! They get to see how and why they're using it, what's working well, and what isn't. It's an awesome feedback loop.

Two of the worst anti-patterns we see are where management makes a product decision without talking to users, and where management makes a platform decision without talking to developers and operators.

So, what can management take away from this and what can Leadership Teams learn from the double Mobius mental model to platform and application product continuous discovery and continuous delivery? What if organizational leadership and strategy was also treated as a complex product?

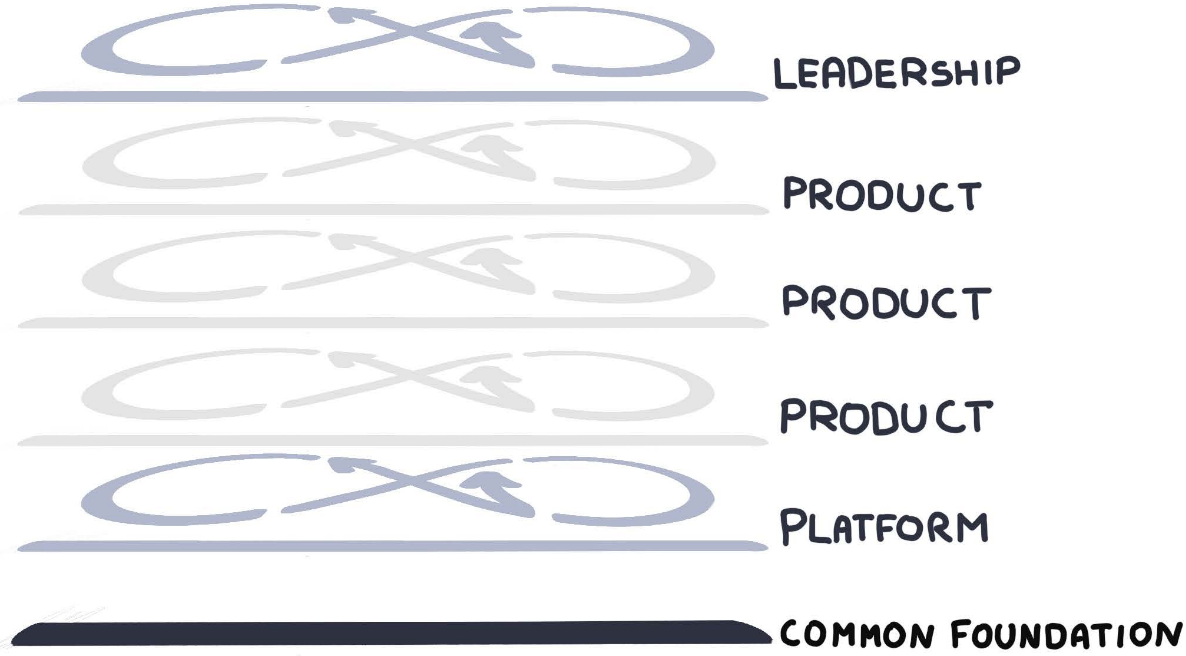


Figure 18.10: Mobius ecosystem of leadership, products, and platform

If an organization's strategy is to be treated as a complex product, that means leadership taking a continuous discovery, Options Pivot, and continuous delivery approach.

Product Ownership of strategy needs to understand the context and motivation of the market, informed by information visualized by product teams (just as Product Owners of Application Product Teams need to understand the context and motivations of users and business stakeholders; just as Product Owners of Platform Teams need to understand the context and motivations of developers and operators).

Leadership needs its own foundation and needs to inject stability into the product and platform foundation. In some cases, this means removing the *rotting foundation* from legacy processes. Examples we see of this include the way HR, financial, legal, and risk processes are conducted. All of these benefit from a product mindset and being tightly connected to their users' empathy.

If we revisit once again the practices introduced in the different sections of this book, we can see how they can be applied by a Leadership Team taking a product mindset toward its strategy. For example:

- Every strategy should have a North Star.
- Impact Mapping has been used by many organizations to drive the organization's company strategy, articulated by a goal, actors, target impacts, and deliverables.
- Practice such as Event Storming, Empathy Mapping, and other human-centered design can be used to gain shared understanding and empathy with shareholders, financial stakeholders, owners, competitors, and other market forces.
- The Options Pivot practices in *Section 5*, *Deliver It*, can be used to prioritize work, design experiments, and even consider advanced deployment of products into the market.
- The Delivery practices in *Section 6*, *Build It*, *Run It*, *Own It*, can be used to deliver increments of the strategy in an Agile way and run Showcases (to the company and the market) and Retrospectives to continuously improve. Measure what matters and continuously learn. Capture and radiate organizational metrics and drive learning and continuous improvement.

The outcome of Leadership Strategy as a Product is to set the intent for Application Product teams. Product Teams set the intent to Platform Teams. All teams are informed by measures, learning, and visualization—both to improve themselves and to radiate up from platform to product, and then to leadership.

The final practice from the toolbox of practices to help connect these is called **Objectives and Key Results**, or **OKRs**.

OKRs comprise an **objective**—a clearly defined goal—and 3-5 **key results** that are specific measures used to track the achievement of that goal. The goal of OKRs is to define how to achieve objectives through concrete, specific, and measurable actions. Key results can be measured on a scale of 0-100%. Objectives should also be supported by initiatives, which are the plans and activities that help to achieve the objective and move forward the key results. OKRs can be shared across the organization, with the intention of providing teams with the visibility of goals and to align and focus effort. OKRs are typically set at the strategic, team, and personal levels.

A nice way to start OKRs is to link it to the Impact Mapping practice introduced in *Chapter 8*, *Discovering the Why and Who*. The goal statement in the Impact Map translates to an **Objective** (**O**). The measurable impact statement connected to the goal translates to **Key Results** (**KR**). So, the Impact Mapping practice being used in the Discovery Loop of Strategy and Leadership helps set the intent for products, which can each have their own Impact Maps and OKRs. A Platform Team can also use the Impact Mapping practice to capture and radiate their *Why* and build OKRs to support the application products and strategy above.

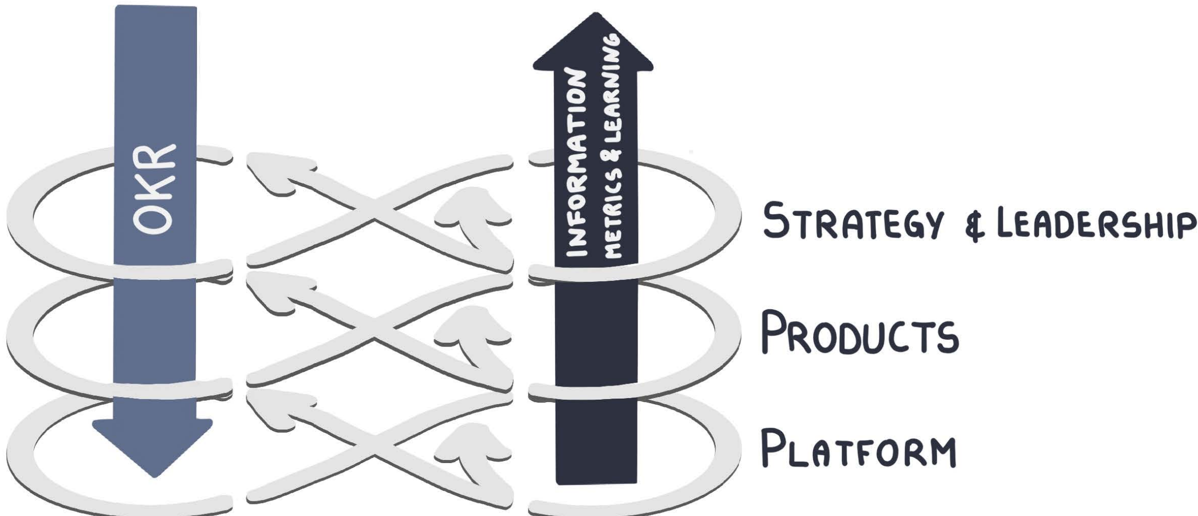


Figure 18.11: Mobius Loops connected by OKRs and information, metrics, and learning

The double loop learning applied throughout this book using Mobius to evolve products applies to platforms and leadership strategies. They are all connected through OKRs, which flow down the organization and change based on information, metrics, and learning flowing up from the platform and products.

Conclusion

In this chapter, we explored how we *Sustain It*—how we sustain everything we've covered in this book and potentially even grow it!

A lot of this comes down to infectious enthusiasm—infectious enthusiasm for the technology and platform, for the way of working and the practices, and for the team and culture. We looked at different ways that infectious enthusiasm is generated and some stories on how the *one team*, *one dream* mantra evolved to multiple Application Product Teams and, eventually, the need for a Platform Team.

Whether teams in your organization are the first, second, tenth, or hundredth Application Product, Platform, or Leadership Team, treating the subject being delivered as a product is the success factor.

Let's leave you with our top 10 tips for transformation towards DevOps Culture and Practice with OpenShift:

1. Never forget the foundation–building a foundation of culture and technology practices enables business agility.
2. Never stop improving–whether it is the foundation, the technology, or the products.
3. Never stop learning–the entire system of work is focused on being able to learn at every level. Whether it be a product, a team, a platform, or an organization's strategy, continuous learning should be central to everything.
4. Never done. We call this **#neverdone** because we are working in infinite loops–double loops of learning that are continuous and never end, right up until product, platform, or organizational shutdown.
5. Show, not tell. This is one that might seem over-said, but one of the true measures is working software. We need it to get feedback and to go around the loops again.
6. Culture is key. Culture can be hard to define, but it's the sense of camaraderie that a team has. It's the in-jokes, the banter that leads to pulling together when things are tough. Or being willing to do that rubbish task this sprint because you know in the next one, someone else will take their turn.
7. Invest in your people. A culture is hard to build but easy to break. Demotivation can creep in and erode trust.
8. Listen to the engineers; they also need to listen to and understand the needs of the business and users, however.
9. Learn by doing–experiment, experiment, experiment. Fail safely, fail fast, and, most importantly, learn from it. Learn by doing.
10. Have fun!

Appendix A – OpenShift Sizing Requirements for Exercises

OpenShift clusters support many types of footprints – cloud, datacenter, and local. You can discover the best way to install OpenShift in these different environments by browsing to this link: https://cloud.redhat.com/openshift/create/.

There is a distinction between *Red Hat supported* and *community* versions of OpenShift. You may wish to get full enterprise support for the Red Hat versions from Red Hat! Especially if you are doing anything work related, this is highly recommended. The community versions are supported by the open source community, you can read about them here https://www.okd.io/.

We have detailed using **CodeReady Containers** (**CRC**) on your local machine as the way to get started with OpenShift. There is both a supported version of CRC, https://cloud.redhat.com/openshift/create/local, and a community version, https://www.okd.io/crc.html, available. To run CRC with the default settings, execute the following:

```
crc start
```

In an effort to constrain the system resources used, CRC does not contain all of OpenShift's features. Running CRC locally will take a lot longer if executing the techie bits of this book. You may find frequent timeouts and have to *try some commands twice*. So, either be patient or perhaps use a Cloud hosted OpenShift which has fewer resource constraints. If you have real money to spend on cloud hosting or you have a virtualized infrastructure available to you, you can install an OpenShift 4 cluster using the binary installer:

```
openshift-install cluster create
```

This will give you a running cluster with multiple master and worker nodes and you can configure different options prior to installing. Read the docs here: https://docs.openshift.com/container-platform/4.7/installing/index.html.

CRC can be started with various options. Importantly, you can set the following:

- `-c, --cpus int`: The number of CPU cores to allocate to the OpenShift cluster (the default is 4)
- `-m, --memory int`: The amount of memory (in MiB) to allocate to the OpenShift cluster (the default is 9216)
- `-d, --disk-size uint`: The total size (in GiB) of the disk used by the OpenShift cluster (the default is 31)

You can also configure the IP address – check out `crc start --help` for details.

We have tested the following scenarios from the book using CRC and recommend the following minimum sizing. If you have more resources available locally, we recommend you use them! This will improve the performance and usability when running through the code.

Book Chapter Heading	CRC command	Comments
Chapter 6, Open Technical Practices – Beginnings, Starting Right # Install Jenkins using Helm	`crc start -c 4 -m 10240`	If you choose to build more agents, use more disk for CRC (the `-d` argument). All agents require a minimum 60 GB disk.
Chapter 14, Build It # Running PetBattle	`crc start -c 4 -m 12288`	Helm installation of the PetBattle applications
Chapter 7, Open Technical Practices – The Midpoint # Implementing GitOps	`crc start -c 4 -m 16384 -d 50`	ArgoCD tooling exploration and bootstrapping.
Chapter 14, Build It	`crc start -c 4 -m 16384 -d 50`	If you want to run all the examples, choose this configuration.

Table Appendix A.1: Minimum sizing for various book scenarios

CRC is optimized to run on a single machine and so has metrics and monitoring disabled by default. This can be enabled by running the following command:

```
crc config set enable-cluster-monitoring true
```

Be aware that you will require more memory than listed above to be able to run the monitoring stack, CI/CD, and PetBattle. For the full documentation, options, and troubleshooting for CRC, please see https://code-ready.github.io/crc/.

How To Resize Storage in Your CRC Virtual Machine

To grow the image disk size, you need to stop and restart CRC to properly resize the image disk. This seems to be a bug/issue with CRC.[1][2]

On Linux, perform the following to resize the disk:

```
$ crc start -c 4 -m 16384 -d 50
$ CRC_MACHINE_IMAGE=${HOME}/.crc/machines/crc/crc.qcow2
$ crc stop
$ cp ${CRC_MACHINE_IMAGE} ${CRC_MACHINE_IMAGE}.ORIGINAL
$ virt-resize --expand /dev/vda4 \
$ {CRC_MACHINE_IMAGE}.ORIGINAL ${CRC_MACHINE_IMAGE}
$ rm -f ${CRC_MACHINE_IMAGE}.ORIGINAL

crc start -c 4 -m 16384 -d 50
```

Tekton Persistent Storage

Depending on your environment, the type of storage class available to your OpenShift cluster will vary.

CRC specifics

Currently, there are two Kustomize overlays for `PersistentVolumeClaims` used in OpenShift. This is because there are different storage classes available in each environment. In *Chapter 7*, *Open Technical Practices – The Midpoint*, we bootstrap the tooling using ArgoCD. The Ubiquitous Journey file `tekton/kustomization.yaml` needs to be set accordingly by changing the following commented line:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

bases:
- persistent-volume-claims/aws
# - persistent-volume-claims/crc
- rolebindings
- tasks
- pipelines
- templates
- triggers
```

1 https://access.redhat.com/solutions/4969811

2 https://github.com/code-ready/crc/issues/127

Cloud specifics

In AWS, we make use of `aws-efs` (https://docs.openshift.com/container-platform/4.4/storage/persistent_storage/persistent-storage-efs.html) for RWX storage. An updated EFS driver is in the works. If this is not available in your environment, you will still be able to run the examples but not all pipeline jobs will function currently if they are run in parallel.

Edit the `tekton/persistent-volume-claims` files to use a storage class available in your cluster (for example, gp2 is the default in AWS). You can find the storage class name by running the following command:

```
oc get storageclass
```

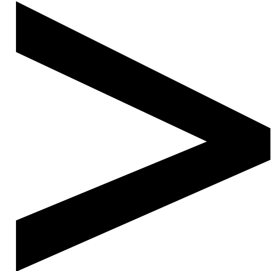
Appendix B – Additional Learning Resources

In this book, we've mentioned tools and techniques that we have used while writing it, with the intention that you will be able to use them in your own domain.

However, this list isn't exhaustive and there are others that we've used that deserve mention as well:

- https://www.konveyor.io/: The Konveyor community is working on projects to help users re-host, re-platform, and refactor their applications to Kubernetes.
- https://OpenShift.tv: Red Hat OpenShift streaming. Experience live, unfiltered demos (with no safety nets).
- https://www.telepresence.io/: This is an open source tool that lets you run a single service locally while connecting that service to a remote Kubernetes cluster. Debug your Kubernetes service locally using your favorite debugging tool.
- https://www.eclipse.org/jkube/: This is a collection of plugins and libraries that are used for building container images using Docker, JIB, or S2I build strategies. JKube also provides a set of tools, including watch, debug, and log, to improve your developer experience.
- https://www.openpolicyagent.org/: This is an open source, general-purpose policy engine that unifies policy enforcement across the stack. **Open Policy Agent** (**OPA**) provides a high-level declarative language that lets you specify policy as code and simple APIs to offload policy decision-making from your software. You can use OPA to enforce policies in microservices, Kubernetes, CI/CD pipelines, API gateways, and more.

- http://uncontained.io/: This began as a project in the Red Hat Container Community of Practice to share knowledge about OpenShift adoption with members of Red Hat's Consulting organization. We quickly realized that the guidance and knowledge we were sharing were worth publishing, meaning we could share it with customers, partners, and members of the community.
- https://opendatahub.io/: Open Data Hub is a blueprint for building an AI as a service platform on Red Hat's Kubernetes-based OpenShift® Container Platform (https://www.openshift.com/) and Ceph Object Storage (https://www.redhat.com/en/technologies/storage/ceph). It inherits from upstream efforts such as Kafka/Strimzi (https://strimzi.io/) and Kubeflow (https://www.kubeflow.org/) and is the foundation for Red Hat's internal data science and AI platform.
- https://developers.redhat.com/: All things developer-related at Red Hat – blogs, e-books, and so on.
- https://github.com/openshift/odo: **OpenShift Do** (**ODO**) is a developer-focused CLI for Kubernetes and OpenShift.
- https://strimzi.io/: Strimzi provides a way to run an Apache Kafka cluster on Kubernetes in various deployment configurations.
- https://github.com/wercker/stern: Stern allows you to tail multiple Pods on Kubernetes and multiple containers within the Pods.
- https://learn.openshift.com/: An interactive learning portal for OpenShift.
- https://www.redhat.com/en/services/training/learning-subscription/: Red Hat Learning Subscription



Index

About

All major keywords used in this book are captured alphabetically in this section. Each one is accompanied by the page number of where they appear.

Symbols

3-5-3 formation 441, 465
- Daily Scrum Event 453, 454
- Development Team Role 444
- Product Backlog Artifact 445, 446
- Product Increment Artifact 450
- Product Owner Role 442, 443
- ScrumMaster Role 443
- Sprint Backlog Artifact 447-449
- Sprint Planning Event 451-453
- Sprint Retrospective Event 458-462
- Sprint Review Event 455-458

@QuarkusTestResource 616

A

A/B Testing 401, 402, 649
- deploying 652-655
- experiment 649
- Matomo 650-652
- reference link 402

acceptance criteria 437, 438, 451

actors 239, 240-243, 247, 717, 747

adaptability 106, 217

adaptation 96

adaptive 510, 744

advanced deployment strategies 401
- A/B Testing 401
- Blue/Green Deployments 402
- Canary Releases 403
- case studies 408, 409
- considerations 406
- Dark Launches 404
- Feature Flags 405

Affinity Mapping 65

Agile 426, 427 , 431, 712, 748
- history 424
- traits 426

Agile coach 9, 66, 201

Agile Manifesto 21, 424
- principles 425, 450
- versus Waterfall approach 426

Alberto Brandolini 271

alert 678
- examples 678
- managing 680
- need for 678, 679
- types 679

alerting 678, 701

Alertmanager 684

alert fatigue 685

Alistair Cockburn 56

Allure 598

Amazon 97, 215, 274

Amy Edmondson 54

analysis paralysis 708

Andon Cord 70, 79

Angular 177, 549, 556

AngularJS 129

annotations 613, 676, 677

Ansible 138, 153, 699, 715

Apache Bench 627

Apache JMeter 628

APIs 147, 148, 196, 617, 693

app chassis 41

application metrics 669, 680
- measuring 511, 512

Application Products 738, 747

Application Product Teams' approaches 738

App-of-Apps pattern 546-563

architecture 217

Argo CD 181-183, 537, 543, 544, 545
- App-of-Apps pattern, anatomy 546, 547, 548
- trunk-based development and environments 545, 546

artificial intelligence 600

automate 621, 643

automation
- improving 713

Automation Test Pyramid 611

autonomy 58

aws-efs
- reference link 759

B

bake 550-557, 591, 622
Barry Boehm 423
baseline metrics 39, 316, 319, 333, 515
Behavior-Driven Development (BDD) 199-203
behavioral change 243, 322, 349
big bang software releases 611
Big Picture 172, 173
- building 184-194
- creating 179
- drawing 174
- example 172
- reference link 180

Big Picture, CI/CD for PetBattle 549, 550, 558
- BUILD component 551, 552
- DEPLOY component 553, 554
- Jenkins 559
- pipeline stages 556, 557
- PROMOTE component 555
- SYSTEM TEST component 554

Bitnami 568
BizDesDevSecOps 23
BizDevOps 23
blast radius 645
Blue/Green Deployments 402, 403, 656-658
- previews 658, 659
- reference link 403

branching 165-167, 282, 579
build 128, 170
BuildConfig 591, 695
build information 677, 678
business agility 711
business analysts 153, 273, 274
Business Model Generation 28
business outcomes 25, 253, 404

C

Canary deployment 404, 648
Canary Releases 403
- reference link 404

catastrophic system failure 213
Ceph Object Storage
- reference link 762

Cert-Utils 696, 698
chain builds 41
chaos 433, 644
Chaos Engineering 644, 645
Chaos Mesh 645
Checkstyle 621, 639
Chief Information Security Officer's (CISO's) 507
CI/CD, for PetBattle 549
- Big Picture 549, 550

CI/CD pipeline 191, 712, 717
CI/CD tooling landscape 596-598
CIO 9, 122
circuit breaker 664, 665, 685
circuit breaker pattern 664, 665
Clair 636
clear board 129, 740
cloud-native 9, 181, 182
cluster resources 533
cluster-scoped 688, 696
Code Coverage 207, 490, 639
code quality 141, 142, 170
CodeReady Containers (CRC) 533, 755-757
- specifics 758

CodeReady Containers (CRC), troubleshooting
- reference link 757

co-located 43, 234
command 278
Common Expression Language (CEL) 594
Communities of Interest (CoIs) 742
Communities of Practice (CoPs) 742
complex domain 594, 698
complicated domain 195, 428
ConfigMaps 680, 694
configuration-as-code 128
Conftest 633, 635
container
- history 133, 134
- reference link 137

registries 135
working 134-136
Containerfile 552
Continuous Delivery (CD) 146, 308
reference link 149
Continuous Deployment (CD) 149, 151
reference link 151
continuous improvement 27, 59, 120, 706, 718, 722, 737
Continuous Integration (CI) 144
feature branches 168
reference link 146
continuous learning 27, 717, 722
continuous metrics inspection 342
control loops, Kubernetes
working 695, 696
conventional organization
versus open organization 102
Cost of Delay 393
risk reduction and opportunity enablement 393
using 393
Cost of Delay, components
direct business value 393
time criticality 393
COVID-19 43, 85, 182, 726
Cowboys 721
CRC Virtual Machine
storage, resizing 758
CRDs 596
Cucumber 202, 620
Cucumber-style BDD 618
culture 57, 58
measuring 510
culture and practice
explaining 34, 36
customer requirements 16
customers
alerting 518
Custom Resource (CR) 693
Custom Resource Definition (CRD) 694
CVE 506, 626
Cynefin 428-430, 748
Cypress 617

D

Daily Scrum 441, 453, 454
Daily Stand-Up 441
Daniel Jones 16
Daniel Roos 16
Dark Launches 404
reference link 405
data designs 19
day 2 operation 41
Decision-Making Contexts 428
Definition of Done
defining 478-481
reference link 481
Definition of Ready 435-438
reference link 440
deliverables 233-250
Delivery Loop 50
examining 708
Delivery Map
creating 522
delivery practices 710, 729
Demo Day 728, 729
dental checkup 611
deployment configuration 762
deployment pain
measuring 509
deployment pipeline
aim 532
deployment previews 658, 659
Design of Experiments 373-376
reference link 376
design sprints 29, 382-385
reference link 384
using 382
design thinking 331, 337
DesOps 23
detail design 19
Developer-Driven Testing (DDT) 199, 200
development teams 136, 153, 156, 165
DevOps 13, 14, 24
anti-patterns 745
DevOps team 24
DevSecOps 23, 41
digital Big Picture 174
digital templates 44
Discovery Loop 50, 221, 222, 349
Discovery map 343, 346
discovery practices 224, 225 , 330, 337, 343

distributed teams 43, 44
Dockerfile 552
documentation process 730
Domain-Driven Design (DDD) 215, 272
domain-specific language (DSL) 572
Domino Effect 664
DORA report 499
dot voting 65, 115
double-loop learning 42
Double Mobius Loop 747-751

E

Eastman 103, 105
Elasticsearch 688
emerging architecture 39, 72, 211, 218
Emily Webber 76
empathy 60, 179, 225
empathy mapping 29
enabling outcomes 338, 339, 346
end-to-end testing 617-620
Envoy 607, 686
events 273
Event Storm Flow 275
event storming 29, 270-273, 297, 376, 434, 446
- architecture approach 297, 298
- ingredients 273, 274
- recipe 274-287

everything-as-code 128, 152
- implementing, approaches 153
- need for 152, 153
- reference link 155

Example Mapping 204, 205
- example 206
- reference link 207

executives 20, 43
experiments 27
eXtreme Programming (XP) 21
extroverts 239

F

failures
- radiating 93, 95

Fault Tolerance 664
Feature Flags 405, 406
- reference link 406

feature implementations 406
Feature Toggles 405
feedback loops 489, 490, 713
Ferrari 430
Fibonacci 394
Fist of five 65, 115
Fluentd 665
Flux 181
foundation 48, 51
- encouraging 741, 742

frameworks 414, 426, 431, 440
Freddie Mercury 686
Frederick Taylor 426

G

Gabrielle Benefield 26, 325, 338
Gatekeeper 633
Gateways 688, 689
Gaussian 626
Gene Kim 71, 110, 340
George Spafford 71
Git 165
GitFlow 167, 168
- 165, 166
- Develop 166
- feature 168
- Master 166

GitHub 165, 411
- secret, creating 566-568

GitHub Flow 166
GitLab 502, 518, 582
GitOps 180, 537, 595
- implementing 181
- versus infrastructure-as-code (IaC) 180

GitOpsify 571
GitOpsy 568
goal 710, 745, 746
Gojko Adzic 233
Golang 584
Golden Hammer pattern 213
good architecture 211, 212
good experiments
- success points 375

good old fashioned chat 115

good service design 215
achieving, principles 215
Grafana
used, for visualizing metrics 672
GrafanaDashboard 672
GrafanaDataSource GrafanaDashboard 672
green from go 39, 129, 130
guerrilla testing 496
reference link 497

H

hammer 196, 213, 508, 585
happy customers 14, 25
heat map voting 115
Helm
Jenkins, installing with 160-164
overview 158, 159
Helm charts 158-162
YAML files 159
helm install 162, 579
helm tool
installation link 159
Henrik Kniberg 396, 397
Hero Developer 130
hey tool 627, 628
high availability 137, 307, 504
Highest Paid Person's Opinion (HIPPO) 392
high-level design (hld) 19
Hirotaka Takeuchi 440
Horizontal Pod Autoscaler (HPA) 510, 630
Hosier 423
how-might-we 29
How-Now-Wow Prioritization 379-381
emerged idea groups 380
reference link 380
human-centered design 18, 255, 331
Think, Make, Check practice 258
human-centered design practices
reference link 266
hypothesis 324, 338-351

I

Ikujiro Nonaka 440
Ilaria Doria 34, 731
image classification model 600
image scanning 636-638
image tagging 34, 41
Impact and Effort Prioritization Matrix 377, 378
advantages 377
emerged idea groups 378
reference link 378
Impact-Effort Prioritization 376
impact map 233-250
actor's behavior change, identifying 241-243
actors, identifying 239-241
deliverables 243, 244
example 253
goals 238
need for 235
reference link 254
SMART goals 236
summarizing 247, 248
impact mapping 29, 39
impacts 369, 403
implement 50
infectious enthusiasm
creating 726, 727
Infinispan 536, 616, 673
information leakage 216
information radiator 56, 86-88
setting, at home 88, 89
information refrigerator 56
InfoSec 505
InfraOps teams 136
infrastructure as code 7, 30, 51, 128
infrastructure engineer 9, 33
infrastructure platform costs
measuring 512, 513
initial Product Backlog
forming 385-387
Inno-versity 110
inspection 96
instrumentation 34, 41
internal registry 579
invest 44

INVEST criteria 437
investigative alert 679
I-shaped 10
Istio 686-688
  resources, for traffic management 688
Istio OpenShift Routing (IOR) 688
IT leader 9, 33

J

JaCoCo 625
Jaeger 666
James Womack 16
jasmine 625
Java 613-625
Java archive (jar) 678
JavaScript 141, 177, 208
Jeff Patton 365
Jeff Sutherland 435, 440, 454
Jenkins 559 , 581, 582
  Argo CD, connecting to CD 560-562
  Argo CD, connecting to Git 559
  feature development pipeline 583
  installing, with Helm 160, 161, 163, 164
Jenkins Blue Ocean 620
Jenkinsfile
  anatomy 572-578
  branching 579
  future 584, 585
Jenkinsfile DSL
  aspects 574
Jenkins Templating Engine (JTE) 585
Jest 613
  using, for service and component testing 613-616
Jez Humble 137, 340
Jidoka 71
Jimmy Janlen 95
John Willis 70
Josh Arnold 392
Joshua Seiden 323
JUnit
  unit testing with 612, 613
just enough 343, 705, 711

K

Kafka/Strimzi
  reference link 762
Kanban 475, 476, 748
  example 477
  reference link 478
Kanban board 476
Ken Shwaber 21, 440
Kevin Behr 71
KeyCloak 301, 534-538, 541
Key Results (KR) 752
Key-Value (KV) 695
Kiali 688, 692, 696
Knative 41, 528, 601
Knative Eventing 601
Knative Serving 601
knowledge distribution 223
kn tool 605
Kodak problem 103, 104
Kodak story
  learnings 105
Konveyor community
  reference link 761
Kourier 607
Kraken 645
Kubeflow 602
  reference link 762
kubelet 662
KubeLinter 633
Kubernetes
  logging on 665
Kustomize 546, 547, 603

L

labels 673-676
label selectors 670, 673
lagging indicators 503
Language Quality Profile 624
leadership 110, 111
  roles 108
lean 13, 22, 598
Lean Metrics
  measuring 502, 503
Lean startup 401, 485, 498
Lean UX 28, 255
legacy systems 37
Linkerd 685
linting 141
Linux 133, 160, 758
Linux Container 552
Litmus Chaos 645
liveness 664

liveness probes 663
logging 665, 666
logical architecture 19, 297, 400
Log In feature
  acceptance criteria 437
Loudest Voice Dominates (LVD) 392
Luke Skywalker 58

M

Maistra 688
Marcus Maestri 70
mastery 58
Matomo 650, 655
  installing 651
Maven 556, 587, 622
Mean Time to Recovery (MTTR) 645
Mean Time to Repair (MTTR) 341, 645
measurable outcomes
  visualizing 516
measure-and-learn 29, 498
  collecting 487
  experiment results 492
  inspecting 488
  retrospective 488-492
  usability testing 493
  user testing 493
metadata 34, 159
Metric-Based Process Mapping (MBPM) 308-310
  case studies 313-319
  improvements 311
  improving, through iteration 312, 313
  Lead Time (LT) 310
  Percentage Complete and Accurate (PCA) 310
  Process Time (PT) 310
metrics 666, 667
  gathering, in application 667, 668
  improving 713
Metrics-Based Process Map 714
metrics-driven transformation 486, 487
Michael Cohn 195
Micrometer 667, 680
microservices 72, 272, 761
mind-blowing metrics 729, 734
mindset 724-751
minimal viable space 120, 123
MinIO 601-605
Miro 44, 68, 116
mobbing 131, 132, 133
Mobius Loop 6, 27, 28, 49, 127, 128, 432, 433, 725
  advantages 28
  foundation, adding to 30
  reference link 26, 28
  warnings 708-710
Mobius Navigator 724
  core elements 27
  principles 27
mob programming 130, 131, 170, 475, 728
mocha 625
MongoDB 600, 615, 629, 656
M-shaped 11, 719, 746
MultiBranchPipeline 545
Mural 44, 68, 123
Mutual TLS (mTLS) 686

N

namespaces 176, 182, 696
namespace-scoped 696, 697
native object store 601
nearshore 43
network mapping 81, 82
  reference link 83
Nginx 535
Nicole Forsgren 340
Node Package Manager (npm) 551
non-functional map 304, 305, 308
  requirements 304
non-functional testing 207-211, 621
  chaos engineering 644-647
  code coverage 639-644
  linting 638, 639
  types 638
North Star 705, 747, 751
North Star Framework 225-232
  canvas 229
  checklist points 225
  example 228
notification 517, 518, 522

Not Safe For Families (NSFF) component 600
as serverless 600, 601
invoking 607-610
Knative Serving Services 604-606
OpenShift Serverless Operator 603, 604
pre-trained model, generating 601, 602
pre-trained model, obtaining 602
principles 611
Not Safe for Families (NSFF) process 294

O

Objectives and Key Results (OKRs) 751
observability
661, 662
observations 213
offshore 43
open culture 44, 54, 57
creating 59
Open Data Hub
reference link 762
Open Data Hub community
URL 601
Open Innovation Labs 44, 49
open leadership 105, 106
open organization 63, 101, 102, 106
Open Policy Agent (OPA) 41, 153, 633, 761
URL 761
Open Practice Library 31, 32
open practices 724
OpenSCAP 638
OpenShift 427
installation link 755
metrics, visualizing 671
OpenShift Alertmanager 684
OpenShift Container Platform (OCP) 9, 49
reference link 762
OpenShift, deployment strategies
Blue/Green deployment 648
canary deployment 648
recreate deployment 648
rolling deployment 648
OpenShift Do (ODO)
reference link 762
OpenShift, learning portal
reference link 762
OpenShift Serverless Operator 603
open source 724
open technology 6, 44, 723
OpenTracing 666
operational performance 499
operator 693-695
in PetBattle 697, 698, 699
scopes 696, 697
Operator Docs, for Argo CD 187
OperatorFramework 699
Operator Hub 695
Operator Lifecycle Manager (OLM) 596, 695
Operator pattern 694
operators 661, 686, 699
Operator Scopes 696
Opossum 685
Options Map 409, 412
case study 411
creating 410
Options Pivot 50, 353
scene, setting 350
organization
sustainable change 107
organizational chart
changing 106, 107
outcome-driven thinking 28
outcomes 322, 323
versus outputs 323
OWASP Dependency-Check 626
OWASP Zed Attack Proxy (ZAP) 643

P

PagerDuty 684
pairing 131
pair programming 130, 131
advantages 131-133
patterns 685, 718
Pelorus 502
performance
improving 10, 756
measuring 507

performance metrics 34, 41
perf testing 627-633
Personal Access Token (PAT) 567
PetBattle
  running 538-543
PetBattle architecture 533
  components 534, 535
  issue 533
  Pet service 536
  Plan of Attack 537
  Tournament Service 536
  user management 536
PetBattle components
  user interface 535
physical architecture 19, 712
pipeline 41, 72, 129, 137
pivot events 277
placing bets 239, 248
platform 228
platform adoption metrics 341
Pod Disruption Budget (PDB) 632
POJO 613
Polly 685
poor architecture
  characteristics 211
Post-its 204
practice 5, 685
  attributes 5
predictions 257, 609
prioritization 391
  Cost of Delay, using 392
  value, versus risk 391
  WSJF, using 392, 394, 396
priority sliders 75, 112, 113, 728
  examples 114, 115
  for distributed people 116, 117
proactive notification 517, 522
probes 662-664
  used, for performing actions 662
probes, Kubernetes
  liveness 663
  readiness 662
  startup 663
process isolation 134
product backlog 705, 708, 711
product backlog refinement 29, 389, 390
  activities 390
product increment 441, 450, 740
product owner 451, 455, 465
  iteration goals, prioritizing 399
Product Ownership 396, 397, 399
product team 23, 543
program specifications 19
project manager 28, 33, 55
Prometheus
  configuring, to retrieve metrics from application 669, 670
  used, for querying 671
Prometheus query language (PromQL) 671
promote stage 619
prototypes 256, 257, 710
Protractor 612, 617
psychological safety 54
Pub Retro 462, 463
purpose 58
PyBreaker 685
Python 685

Q

qualitative feedback 42, 374
qualitative user research 256
Quality Gates 621, 622
quality of software 611
quantitative characteristics 666
quantitative feedback 374
quantitative metrics 42
Quarkus testing 612
Quarkus 669, 680
Quay 518, 564, 636
Quay.io 564-566
  URL 563
question key 277

R

rainforests 431
reactive alert 679
readiness 307
real-time retrospective 53, 75-79, 728
  reference link 79
recommended labels 675
recreate deployment 648
recurring patterns/approaches 213
Red Hat 636
  approaches 718
Red Hat Community 634
Red Hat CoP 643
Red Hat Learning Subscription
  reference link 762
Red Hat OpenShift streaming
  reference link 761
Red Hat's culture 6
RED method
  duration 515
  errors 515
  rate 515
Rego 633, 634
Rego playground 634
release candidate 619, 659
release continuously 478
release early 478
release often 478
replication controller 510, 630
research 705
residencies 739
resiliency 645, 685
resources
  links 761, 762
resources and services
  measuring 514, 515
resource validation 633, 634, 635
REST 613
REST Assured 615
  using, for service and component testing 613-616
resume driven development 214
Ripley deployment 648
Rockstar Developer 130
role-based access control (RBAC) 186
rolling deployment 648
Ryan Shriver 26, 338, 415

S

S3 storage 602
Sakichi Toyoda 71
sam's code 208
sandbox builds 579, 658
Scaled Agile Framework (SAFe) 392
scale to zero 601
Scrum 440
  3-5-3 Formation 441
  framework 441
  using, with distributed people 472-475
  URL 440
ScrumMaster 441, 714
SDO 489
SealedSecrets 568, 569
  deploying 568-572
Sean Ellis 225
secrets 559, 563, 565
Secure Hash Algorithm (SHA) 548
security
  measuring 505-507
Security Context Constraint (SCC) 651
Security Experts 712
security vulnerabilities 626, 643
Seldon 601
Selenium Grid 617
Semantic Versioning (SemVer) 544
serverless 600, 603
Service Availability 501
service delivery 256, 702
Service Delivery and Operational Performance (SDO) 499
  measuring 500
service design 215
service levels (SL) 504
  measuring 503
Service Mesh 534, 537
  components 688
service mesh resource
  destination rule 690, 691, 692
  using, in PetBattle 688, 689
  virtual services 690
ServiceMonitor 669, 680
Service Operation Metrics 501

Service Serving Certificate Secrets 700, 701
service testing
with Testcontainers 616, 617
Shabnoor Shah 105
shifting security left 505
short feedback loop 169
short-lived feature branches 169
showcase 734, 739
show, not tell 129, 727-733
sidecar 665, 686
sidecar containers 686, 687
Simon Sinek 107
Single Sign-On (SSO) 535
sketching 382, 731
SLA 217, 314, 503, 504
benefit 504
SLI 503, 504
slicing value 366
SLO 504
SmallRye Fault Tolerance 685
SMART 236
social contracts 7, 61, 62, 728
example 64
reference link 70
retrospective learnings 66
using, by distributed team 68, 69
social media driven development 215
Software as a Service (SaaS) 596
software delivery metrics 169, 341
Software Deployment Metrics
Change Failure Rate 501
Time to Restore 501
software development life cycle (SDLC) 745
Software Development Metrics
deployment frequency 501
lead time, measuring 501
software pipeline 137
software traceability 676, 677
SonarQube 550, 589, 590, 621-626
source-2-image 552
source control managers (SCMs) 594
Source-to-Image (S2I) 163
reference link 163
space 117, 118
examples 118
open workspace 119, 120
Spring Cloud Circuit Breaker 685
sprint 21
Sprint Backlog 447-452, 468
sprint planning 29, 441, 443, 473
sprint retrospective 448-470
Sprint Retrospective Event
reference link 465
sprint review 441, 450, 470-474
Startup 485, 535
start with why 3, 4, 38, 719
Stefano Picozzi 7
Stern
reference link 762
Stop the World event 70-74
reference link 75
Strimzi
reference link 762
subscription 541
Subversion (svn) 165
SuperGloo 685
system designing
considerations 216
system integrators 43
System Test 176, 509, 554, 555

T

Target Outcomes
capturing 325-327
example 329
examples 327, 328
need for 324
scenarios 324
visualizing 329
Target Outcomes practice 322
capturing 326
targets
measuring 487

team
- building 63, 65
- iterations, for enablement 739, 740
- socializing within 80, 81
- success factor 719, 720
- success factors 722

team identity 79
- creating, with distributed people 85, 86

team logo
- creating 83-85

team name
- creating 83-85

team sentiment 89, 90
- blending, with other practices 92
- blending, with other processes 90, 91
- reference link 93
- with distributed people 93

technical debt 426, 481

technology
- sustaining 742-746

tech spikes 406

Tekton 585
- basics 585, 586, 587, 588
- designing 589-593
- GitOps 595
- reusable pipelines 588
- triggers flow 593-595

Tekton Hub 585, 586

Tekton persistent storage 758

templates 44, 68, 158

TensorFlow 600, 601

TensorFlow Serving 602, 606

test automation 30, 51, 728

Test Automation Pyramid 195, 196
- testing 196, 197
- tests, measuring 197
- tiers, service tests 196
- tiers, unit tests 195
- User Interface (UI) 196

Test Containers
- service testing with 617
- used, for service testing 616

Test-Driven Development (TDD) 199
- reference link 200

testing 611-617

testing is an activity, not a role 611

testing phase 617

the big picture 172, 173

The Build 182, 183

Thomas Johnson 499

throughput 507-510

tkn 592

Tom Geraghty 54

topologies 746

Toyota System Corporation 71

traceability 152

tracing 456, 666

Traefik 685

traffic management 685, 688

triggers 211, 455, 593

Trunk-Based Development 167, 168, 545
- reference link 167

T-shaped 10

TSLint/ESLint 638

Typescript 621

U

Ubiquitous Journey 621, 622, 651

UNICEF 450

Unicorn 130, 132

Unicorn Developers 130

unit testing
- with JUnit 612, 613

usability testing 151, 493

USE method
- errors 514
- saturation 514
- utilization 514

user-defined alerts 680-686

User Experience Analytics 515, 516

user experience designer 7, 9, 494

user research 42, 296

User Story Mapping 305, 360
- challenges 372, 373

user testing 376, 455

UX Design 10
- components 256
- empathy mapping 257

V

value chain 13-16, 308
  gaps 16-23
Value Slice Map 357
  setting up 360
value slicing 355, 356, 362
  reference link 372
  setting up 357, 358, 359
value stream 16, 232
Value Stream Mapping (vsm) 308
Val Yonchev 118
version control system (VCS) 165
Vincent Driessen 165
virtual machines (VMs) 134
virtual residency 182
virtual service 685
virtual spaces 123, 124
visualization 667, 708, 713, 716
visualization of work 95, 227, 358
voting 456, 469

W

walking skeleton 399, 400
walk the walls 732, 733
Waterfall 21, 421-423, 431
watermelon effect 55
webhooks 521, 548, 580
Weighted Shortest Job First (WSJF) 392
What You See Is What You Get (WYSIWYG) 543
William Kahn 54
wizards 721
word of mouth 734
Written Showcases 733, 736

Y

Yahoo's Open NSFW Classifier 601

Z

Zalando 617
Zalenium
  URL 617