1ST EDITION

SAFe® for DevOps Practitioners

Implement robust, secure, and scaled Agile solutions with the Continuous Delivery Pipeline

ROBERT WEN

Foreword by Harry Koehnemann
SAFe® Fellow and Principal Consultant

SAFe® for DevOps Practitioners

Implement robust, secure, and scaled Agile solutions with the Continuous Delivery Pipeline

Robert Wen

BIRMINGHAM—MUMBAI

SAFe® for DevOps Practitioners

Group Product Manager: Rahul Nair
Publishing Product Manager: Surbhi Suman
Senior Editor: Tanya D'cruz
Technical Editor: Arjun Varma
Copy Editor: Safis Editing
Project Coordinator: Shagun Saini and Prajakta Naik
Proofreader: Safis Editing
Indexer: Sejal Dsilva
Production Designer: Shyam Sundar Korumilli
Senior Marketing Coordinator: Nimisha Dua
Marketing Coordinator: Gaurav Christian

First published: December 2022

Production reference: 1251122

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

ISBN 978-1-80323-142-6

www.packt.com

To Jill and Charles, who have made my life better in so many ways.

And to Steve, a member of the family through years of friendship. Thanks so much for the support.

– Robert Wen

Foreword

Recently, I attended the 2022 DevOps Enterprise Summit and left with an interesting observation. Although a significant portion of the agenda continues to focus on deep technical and architectural practices, there's a growing interest in social aspects including culture, leadership, and organizational change. My takeaway? While it's true that us engineers value our technical tooling and techniques, a successful DevOps implementation requires more than just good tech.

The week after the summit, I read a preview of Robert Wen's new book, *SAFe® for DevOps Practitioners*. I was delighted to learn that he had made the same observation: For DevOps to succeed, tooling and techniques must integrate with culture, leadership, and other practices. Organizations need to think of them holistically, as we do in SAFe.

Wen masterfully blends these two subjects. In one section, he discusses DORA metrics, the seminal research connecting DevOps practices to business performance. In another area, he introduces the Westrum model of organizational culture, and explains how leaders must model the right behaviors to transform the culture to support a DevOps transformation. In another section, he dives into Continuous Integration (CI) tooling and techniques for automated building, testing, and packaging.

For large enterprises to achieve business agility, there can be no choice between SAFe *or* DevOps. Success will require both SAFe guidance for business agility *and* the technical practices and tooling espoused by DevOps. *SAFe® for DevOps Practitioners* demonstrates how SAFe change agents and DevOps practitioners can work together for overall business success.

Wen makes this connection by structuring the content around three SAFe DevOps constructs:

- Part 1, "CALMR"
- Part 2, SAFe value streams
- Part 3, the Continuous Delivery Pipeline (CDP)

Part 1 explores modern DevOps practices organized around CALMR, the acronym for Culture, Automation, Lean Flow, Measurement, and Recovery. Incorporating the guidance found in SAFe, Wen covers the principle-based elements of Culture and Lean Flow by including Westrum's culture, Deming's batch sizes, and Kersten's project-to-product approach. He also provides insights into the deep technical practices and tooling required for automation, measuring, and recovery.

Part 2 focuses on value streams, a critical element in SAFe and Lean, including chapters defining them and measuring their performance. The first chapter presents guidance on identifying value streams in SAFe and the benefits of value stream mapping. The next chapter discusses the many ways to measure value stream performance through KPIs, DORA metrics, Mik Kersten's Flow Framework, and SAFe's three measurement domains.

Part 3 presents an extensive list of Agile, Lean, SAFe, and DevOps practices, organized by the four aspects and twelve practices of SAFe's Continuous Delivery Pipeline (CDP). That organizing structure enables a deep, technical discussion of critical DevOps techniques, including Lean Startup, architecting for security, shift-left testing, chaos engineering, and many others. Bob puts them into a context that practitioners from both the SAFe and DevOps communities can easily understand.

I hope these brief descriptions give you a sense of the book's remarkable breadth. It connects the SAFe principles and practices enabling business agility to the technical methods and tooling championed by DevOps. If you are a DevOps practitioner, you will connect with the book's rich technical and tooling discussions and see how they align with SAFe's principles and practices to enable business agility across the enterprise. If you are a SAFe practitioner, you will connect with the book's organizing structure around CALMR and SAFe's Continuous Delivery Pipeline (CDP) as you learn more about relevant DevOps technical tooling and techniques.

Finally, *SAFe® for DevOps Practitioners* can either be read from cover to cover or serve as a reference tool. Each chapter provides straightforward, detailed guidance from the relevant DevOps and SAFe bodies of knowledge. Readers (particularly those familiar with SAFe's CALMR and CDP organizing structures) will find value in reading individual chapters and sections as a reference when needed.

Practitioners in the SAFe and DevOps communities help enterprises build some of the world's most important systems. SAFe change agents enable business agility in most of the Global 2000, helping those enterprises survive and thrive in the digital age. DevOps practitioners in these organizations provide the tooling and technical processes that deliver digital products faster and more reliably. *SAFe® for DevOps Practitioners* shows how SAFe change agents and DevOps practitioners work together to accelerate business success.

Harry Koehnemann

SAFe® Fellow and Principal Consultant

Contributors

About the author

Robert Wen is a DevOps consultant at ReleaseTEAM. He specializes in Continuous Integration, Continuous Deployment, Value Stream Management, and Agile Development. He has over 20 years of experience in hardware and software development, working in industries such as defense, aerospace, healthcare, and telecommunications. During his career, he has been a coach as the Scrum Master for several Agile projects and a trainer for courses on the **Scaled Agile Framework®** (**SAFe®**). He holds the following standings and certifications: **SAFe Program Consultant** (**SPC**), Atlassian Certified Expert, Certified CloudBees Jenkins Platform Engineer, GitLab Professional Services Engineer, and Flow Framework Professional.

To the guides and inspirations on my learning journey.

For DevOps, thanks go to Jim and Daphne Herron, Bart Chrabski, and Joel Lanz at Island Training; Thad West, Rodney West, Becky King, Larry Cummings, and Tracy Walton at Isos Technology; and to the present day with Shawn Doyle, Karl Todd, and my coworkers at ReleaseTEAM.

For SAFe, thanks first go to Harry Koehnemann and Cindy VanEpps, who first guided me to SAFe. The journey continued with the help of Randy Smith, Jim Camden, Nate Crews, Abdou Mballey, Terrance Knecht, Sunny Rungasamy, Tojo Joseph, and all my former students.

I dedicate this book to the memory of Jack Caine. Jack, we miss you very much and hope you know how much you have taught us.

About the reviewers

Romano Roth graduated in 2001 as a computer scientist at FHNW. In the 20 years that he has worked as a software developer, software architect, and consultant for Zuhlke, he has built a comprehensive knowledge of software development, architecture, and processes. He has worked with a variety of platforms and technologies, consulting in the financial, insurance, cyber security, electricity, medical, and aviation sectors. DevOps is his big passion. He helps companies bring people, processes, and technology together so that they can continuously deliver value to their customers. He organizes DevOps Meetup Zürich and DevOpsDays Zürich. He also has his own YouTube channel on DevOps and he regularly speaks at conferences.

I'd like to thank my family and friends who understand the time and commitment it takes to work in this fast-changing industry. Working in this field would not be possible without the supportive DevOps community that has developed over the last several years. Thank you to all the trailblazers who make this field an exciting one to work in each and every day.

Martyn Coupland is a senior consultant at Inframon, a UK System Center consultancy company. He specializes in client and cloud technologies and has knowledge of the System Center suite. He spends time with customers, helping them design and implement System Center solutions. He has a background in support, and his career, like many others, started on the service desk in IT, performing first-line support. Apart from his experience with System Center, Martyn has experience in programming languages such as C# and web programming languages such as PHP, HTML, CSS, and JavaScript.

Table of Contents

Part 2: Implement – Moving Toward Value Streams

7

Mapping Your Value Streams 131

8

Measuring Value Stream Performance 153

9

Moving to the Future with Continuous Learning 173

Part 3: Optimize – Enabling a Continuous Delivery Pipeline

10

Continuous Exploration and Finding New Features 189

11

Continuous Integration of Solution Development 211

12

Continuous Deployment to Production 237

13

Releasing on Demand to Realize Value 255

14

Avoiding Pitfalls and Diving into the Future 271

Assessment Answers 287

Preface

Both DevOps and the **Scaled Agile Framework®** (**SAFe®**) are about extending agility beyond the team.

DevOps looks to extend agility beyond the "traditional" boundary of development to include the deployment and release of the development to a production environment. This includes what to do if the new changes developed introduce failures in the production environment, activities typically done by operations teams.

SAFe® strives to extend an Agile mindset to products that require multiple teams to develop and maintain. In this paradigm, alignment and the ability to properly execute delivery and release are among the important factors for success. Both these factors are encouraged by DevOps. It is for this reason that to implement SAFe, you need to make sure that DevOps is implemented as well.

This book looks at the DevOps approach used in the SAFe. While there are practices specific to SAFe, many of the concepts, practices, and theories we introduce for DevOps are truly universal and are not limited to SAFe exclusively.

We hope you enjoy your learning journey!

Who this book is for

This book is intended for IT professionals such as DevOps and DevSecOps practitioners, SREs, and managers who are interested in implementing DevOps practices using the SAFe approach. Basic knowledge of DevOps and Agile **Software Development Lifecycle** (**SDLC**) and their methodologies will be helpful for using this book.

What this book covers

Chapter 1, *Introducing SAFe® and DevOps*, is a brief look at the history of how DevOps and SAFe came to be. We view the conditions that brought about Agile development, the evolution of Agile development to the DevOps movement, and the role SAFe can play in moving to DevOps.

Chapter 2, *Culture of Shared Responsibility*, covers the types of cultures that are present in organizations today, which are beneficial for DevOps. We also look at how to change your organization's culture to one that is needed for DevOps.

Chapter 3, *Automation for Efficiency and Quality*, explores the automation and technology used by organizations to establish a **Continuous Integration/Continuous Deployment** (**CI/CD**) pipeline. We look at tools used for monitoring and measuring the pre-production and production environments. We then finish by discussing the teams responsible for setting it up.

Chapter 4, *Leveraging Lean Flow to Keep the Work Moving*, describes the principles and methods to accomplish a Lean Flow as part of SAFe. We examine the roles that the size of the work, the length of the backlog, how busy our workers are, and the differences between items of work play in the time it takes to complete the work.

Chapter 5, *Measuring the Process and Solution*, studies the potential measurements needed to ensure value, security, and reliability of the product under development. We look at the measurements that help identify whether teams have flow in their development. We explore monitoring and observability to find the metrics that ensure the solution is secure and reliable. Finally, we look to collecting metrics that assess the product's end-user value.

Chapter 6, *Recovering from Production Failures*, outlines some methods to ensure reliability of the product in a customer-facing environment. We look at examples of famous production failures. We explore the discipline of **Site Reliability Engineering** (**SRE**), developed at Google to establish practices and ensure reliable environments. We finish our exploration by looking at Chaos Engineering, which strives to prepare for production failures by establishing experiments of failure in a production environment.

Chapter 7, *Mapping Your Value Streams*, takes a look at how to identify and establish value streams with a value stream identification workshop. We will explore how to prepare for the workshop and the mindset needed for moving to value streams. We then look at the steps needed to identify and map the Operational Value Stream. We finish by identifying and mapping the Development Value Stream.

Chapter 8, *Measuring Value Stream Performance*, delves into the metrics that are used to improve value streams. We explore the metrics that are organized by the DevOps Research and Assessment organization, known as the DORA metrics. We also explore Flow Metrics, a part of the Flow Framework created by Tasktop.

Chapter 9, *Moving to the Future with Continuous Learning*, examines how to become an organization that is continually learning. We explore the disciplines required for continuous learning as well as the practices from Lean thinking that encourage continuous learning such as the Improvement Kata.

Chapter 10, *Continuous Exploration and Finding New Features*, elaborates on the first phase of the Continuous Delivery Pipeline, Continuous Exploration. We explore the use of epics as hypotheses of potential customer value. We elaborate on the hypotheses by ensuring the architecture can allow for these new ideas and maintain the security and reliability of the product. We then look at decomposing the epics into features, ready for an Agile Release Train to develop.

Chapter 11, *Continuous Integration of Solution Development*, discusses the second phase of the Continuous Delivery Pipeline, Continuous Integration, including the start of the automation process. We look at the importance of testing, including the adoption of test-driven development and behavior-driven development. We explore the incorporation of automation in a CI/CD pipeline.

Chapter 12, *Continuous Deployment to Production*, provides an examination of the continued use of automation and practices in Continuous Deployment, the third stage of the Continuous Delivery Pipeline. We continue the exploration of automation through the CI/CD pipeline as it deploys to the production environment packages created in Continuous Integration. We also explore how testing continues in the production environment.

Chapter 13, *Releasing on Demand to Realize Value*, covers the last phase of the Continuous Delivery Pipeline, where customers receive new features through Release on Demand. We explore how teams continuously monitor the system to ensure the product is reliable and secure. We then look to see whether what is released really meets the customer's needs.

Chapter 14, *Avoiding Pitfalls and Diving into the Future*, expounds on the new trends in DevOps in terms of process and technology and some tips and tricks to get you started on your journey. We start with helping you begin your journey to DevOps or SAFe.

To get the most out of this book

For this book, knowledge of the basics of the SDLC is beneficial. Knowledge of SAFe or DevOps is not required.

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: `https://packt.link/79pAN`.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Select **System info** from the **Administration** panel."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share your thoughts

Once you've read *SAFe® for DevOps Practitioners*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781803231426`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Introducing SAFe® and DevOps

Developing products in many organizations—especially ones that work on software-based systems or complex systems involving both hardware and software and further enabled by networking technologies, known as cyber-physical systems—has changed over the past 10 to 20 years. Factors such as changes in technology, movement to geographically distributed or remote development, the push for faster **time-to-market** (**TTM**), understanding the customer needs, and pressures to reduce the occurrence and severity of production failures are opportunities, challenges, or a mixture of both that these organizations face.

To address these challenges and take advantage of the opportunities, mindsets derived from Lean manufacturing began to emerge and evolve. These mindsets, combined with practices from emerging frameworks, began to allow organizations to move past the challenges and improve business outcomes.

In this chapter, we're going to highlight the historical challenges and popular mindsets and approaches that have allowed many organizations to overcome these obstacles. These challenges, approaches, and frameworks are described in the following topics:

- Challenges organizations face in product development
- An introduction to Agile
- An introduction to **DevOps**
- Scaling DevOps with SAFe®

Challenges organizations face in product development

Product development today is enabled by a marriage of technology and society. Every product is a combination of hardware and software that is further enhanced by a connection to the internet. New product enhancements are only a software release away. We truly live in the *age of software and digital*.

It is against this backdrop that we look at challenges in product development and find that not only have these challenges not changed but the challenges also grow even more daunting thanks to the reliance on technology and software. Some of these classic challenges are presented here:

- TTM pressures
- Understanding customer wants
- Security and compliance
- Ensuring quality
- Competition

These challenges don't appear in isolation. Sometimes several challenges appear, or they appear all at once. Let's examine how these challenges, alone or together, impede product development.

TTM pressures

TTM is the measure of the length of time it takes for a new product or new product feature to launch from an initial idea. It is usually seen as a measure of how innovative a company is.

A growing trend is that the length of TTM has decreased in recent years. Advances in technology have increased the pace of innovation. This increased pace of innovation has forced product development cycles to decrease from yearly cycles to 6-month or quarterly cycles. This trend will continue to happen and will force organizations to consider whether they can deliver features more frequently.

Understanding customer wants

Henry Ford is quoted as saying, "*If I had asked people what they wanted, they would have said faster horses.*" It often seems as if that statement is true today. Often, in the beginning, customers have no idea which features they want with a product. If a product requires long development cycles, the customer preferences may change, often to the point that what ultimately gets delivered is not what the customer needs or wants.

Often, what spurs a change in customer wants or requirements could be features offered by similar products provided by competitors. The added pressure from competitors provides a challenge: understand the desires of your customer and release a product or feature that meets those desires before your competition does.

Security and compliance

One challenge that organizations face doesn't come from the marketplace. Products using software face growing threats from hackers and other *bad actors* that seek to take advantage of vulnerabilities in the software. Damages, if they are able to exploit these vulnerabilities, range from reputation to money in the form of ransomware payment or litigation.

In addition, because of the threats of these bad actors, regulations intended to ensure privacy and security have been enacted. Products may have to comply with region-specific (for example, the

General Data Protection Regulation (**GDPR**)) or industry-specific (for example, the **Health Insurance Portability and Accountability Act** (**HIPAA**)) regulatory standards so that a customer's confidential data is not exposed.

Ensuring quality

One thing that is important for organizations is maintaining quality while doing product development. Organizations that do not apply rigor in ensuring that products are developed with built-in quality soon find themselves open to other challenges. Rework means longer lead times and delays in bringing products to market. Customer experience with *buggy software* or a low-quality product may drive them to competitors' products. Inadequate attention to quality may also allow security vulnerabilities to go unnoticed, allowing exploits to occur.

Vigilance toward maintaining quality during product development is ideally done by creating, setting up, and executing tests at all levels throughout development. Testing could even be done in pre-production and production environments and automated as much as possible. An *approval/inspection-based* approach only defers the discovery of problems until it may be too late or too costly to fix them.

Competition

Some of the challenges previously mentioned talked about the part that competition plays. The truth of the matter is that your competitors face the same challenges that you do. If your competition has found a way to master these challenges, they have a distinct advantage in the market.

But it's important to remember that this race isn't about being first. The challenge is to be first with the product or feature and be able to convey why this lines up with customer desires. A famous example comes from Apple. Apple was a couple of years behind other competitors in the marketplace for digital music players when it released the iPod. What made the iPod a runaway product sensation was the marketing that touted the memory size not in terms of **megabytes** (**MB**), but in terms of the number of songs. This simple message connected with the marketplace, beyond technology aficionados and music diehards, to even casual music listeners.

The incredibly successful launch of the iPod drove Apple on a path of innovation that catapulted it to its current place as one of several technology giants. The makers of the first commercial **MPEG-1 Audio Layer 3** (**MP3**) player no longer provide support for their product.

Meeting the challenges

These challenges have plagued product development since the early history of man. The exact form of challenge, however, changes with every generation and with every shift in technology.

TTM cycles will always drive when to release products; however, with the aid of technology, these cycles are shrinking. A customer's requirements may always remain a mystery, often even to the customer. Competition changes will ebb and flow, with the emergence of new disruptors and the disappearance of the *also-rans*.

As these ever-present challenges take on new forms, those organizations that have mastered these challenges do so through new mindsets and practices revolving around three areas: people, process, and technology.

A focus on people involves looking at the mindset, values, and principles that people hold in common to form an organizational culture. This culture is the most important counterpoint to challenges because it informs everybody on how they will meet these challenges.

With the culture established, a focus on the process implements practices that are modeled on the right mindset, values, and principles. Successful application of the practices promotes a feedback loop, encouraging the culture and reinforcing the mindset, values, and principles.

Finally, tools aid the process. They allow practices to be repeatable and automatic, strengthening the process, and allowing the application of the process to be successful.

The remainder of this book will highlight the combination of people, processes, and tools that have helped organizations meet these challenges seen in modern product development. These combinations are set up as frameworks, meant to be flexible enough to apply to different organizations in different industries. These combinations started with software development but warrant a look as the creation of software is prevalent in every organization today.

We begin our examination with a look at Agile development or the transition from large-scale delivery of software to incremental delivery of software in short design cycles.

An introduction to Agile

To understand the context in which these challenges lie, it is important to understand the dominant product development process, known colloquially as **Waterfall**. Waterfall was used for many years to develop cathedrals and rocket ships, but when the process was used for developing software, the strains of it were beginning to show, highlighting inadequacies against the challenges of meeting a shrinking TTM and satisfying customer needs. Something had to be done.

Next, let's look at the emergence of Agile methods from the initial attempts to incorporate Lean thinking into software development, to the creation of the Agile Manifesto and the emergence of Agile practices and frameworks.

The rise and fall of Waterfall

The method known as Waterfall had its origins in traditional product development. Workers would divide up the work into specific phases, not moving to the next phase until the current phase was completed.

In 1970, Winston W. Royce first proposed a diagram and model for this method when product development moved to software, as depicted here:

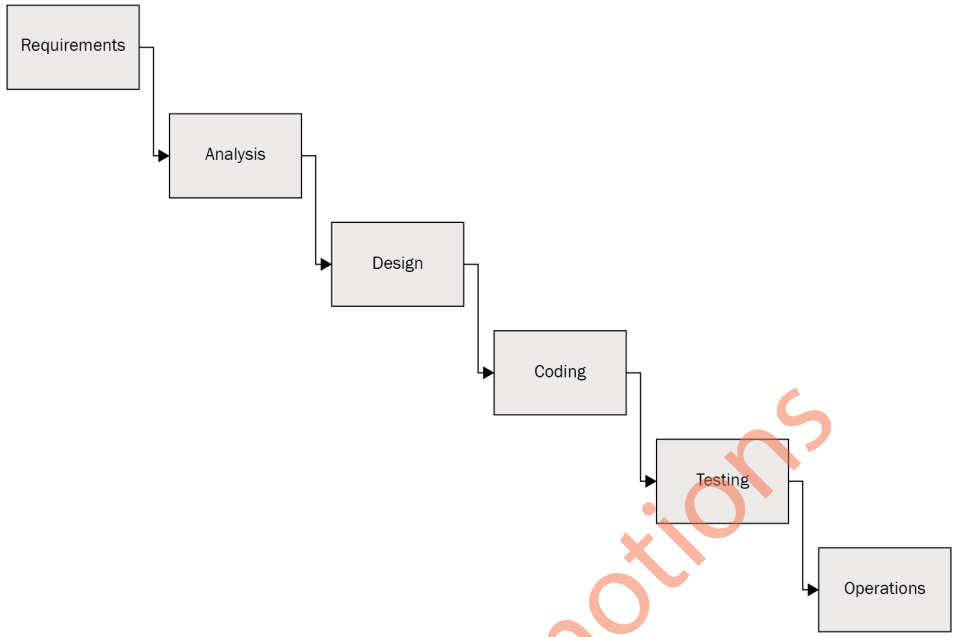


Figure 1.1 – Waterfall diagram

Although Royce never advocated this approach and actually preferred a more incremental approach to development, his diagram caught on and many in the industry called the approach *Waterfall* because the arrows from one phase to the next resembled waterfalls.

In software development, this approach began to exhibit drawbacks. If a requirement, problem, or constraint appeared in the latter phases, the additional work drove the process backward, requiring an enormous amount of rework. Many times, the end customer would not have ideas on requirements early on, leading to rework or a final product that failed to meet customer expectations.

The delays introduced by rework also put pressure on fixed-time projects. To meet the deadlines, some of the later phases (often testing) would be curtailed or eliminated to deliver a product. With the lack of testing, errors or *bugs* would remain undiscovered until the product was released, resulting in low-quality software and low customer value.

TTM pressures on product development cycles reduced the time available to create new software or update existing software. The model was falling apart, but what could be done differently?

The emergence of Agile

In the early 21st century, other methods such as **Extreme Programming** (**XP**), Scrum, and Crystal began to emerge. These methods advocated *incremental delivery*, where a bit of the intended functionality would go through all stages (requirements, design, coding, and testing) in small design cycles, often

no longer than a month. At the end of each design cycle, teams would solicit customer feedback, often incorporating that feedback into the next design cycle.

A representation of incremental delivery, containing short design cycles and delivery of packages of value, is shown in the following diagram:

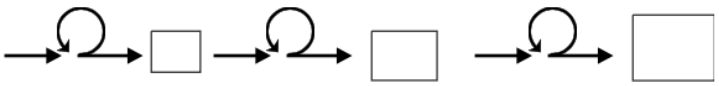


Figure 1.2 – Incremental (Agile) development diagram

Between February 11 and 13, 2001, a group of software development experts, some of whom created **XP**, **Scrum**, and the **Crystal** method met in Snowbird, Utah to ponder which alternatives existed. What resulted from this weekend was a manifesto for Agile software development, simply called the **Agile Manifesto**: `agilemanifesto.org`.

The Agile Manifesto contains a set of values and a list of principles, but it must be noted that the authors of the manifesto talk about the values as a set of preferences. It is possible for Agile teams to have processes, tools, documentation, contracts, and plans. It's only when those items interfere with the items on the left half of each statement of value that the team should re-evaluate the process, tool, or contract, and plan and adjust.

The value set shows what is important, stating the following:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions over processes and tools*
- *Working software over comprehensive documentation*
- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan*

That is, while there is value in the items on the right, we value the items on the left more.

The 12 principles of the Agile Manifesto elaborate and provide context to these values, stating the following:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for a shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

The addition of Lean

Around the same time, others were looking at other ways of developing software with shorter TTM. They looked at applying principles from **Lean manufacturing**.

Lean manufacturing looks to apply practices to reduce waste. The methods were invented by Taiichi Ohno and used to create the **Toyota Production System** (**TPS**). Along with the removal of waste, Lean manufacturing looks to build in quality and strive for **Kaizen**, or continuous improvement.

The application of the principles found in Lean manufacturing was used by Mary and Tom Poppendieck to describe Lean software development in their book, *Lean Software Development: An Agile Toolkit*. These principles are summarized as follows:

- Removing waste
- Emphasizing feedback and learning
- Waiting until the last possible moment for decisions
- Frequently delivering
- Making sure the team is empowered
- Meeting users' perceptions and expectations
- Employing systemic thinking

In addition to the work by the Poppendiecks, David J. Anderson adapted **Kanban**, another tool from the TPS for software development at Microsoft. This adaptation of Kanban was modified as a framework of practices suitable for software development. Kanban soon rose in popularity not only as an alternative to Scrum or XP—many of the practices are used in conjunction with Scrum or XP to facilitate the execution of tasks.

Software development teams did see that the change to Agile development was producing results and overcoming challenges in product development, but the results were only seen in development and not in the overall organization. Clearly, changes needed to be made in other parts of the organization. Let's examine the changes that development and operations made together in the DevOps movement.

An introduction to DevOps

As development teams began to adopt Agile methods and deliver software features incrementally, they faced a challenge in delivering value from the outside. Operations teams, the ones that maintain the development and production platforms where code executes, often do not release new packages from the development teams as they emerge. Rather, operations teams insist on collecting features and deploying them in specified *release windows* to minimize the risk that a single new change would bring down the production environment. But instead of minimizing the risk, it compounds the risk by straining the time allowed for the release windows, with mismatched configurations between development and production environments and untracked manual intervention in production environments. Bundling releases into release packages also moved delivery away from small increments to larger monoliths that may diminish customer value.

At this point, it becomes necessary to view the perspective of a typical operations team. Its job is to ensure that the organization's production environment, the one that may be producing an organization's revenue, is as operational as possible. Any change to that environment, even ones for new features, is seen as a risk to that stability.

In 2009, John Allspaw and Paul Hammond gave a talk titled *10+ Deploys per Day: Dev and Ops Cooperation at Flickr* during the *O'Reilly Velocity* conference. In this talk, they outlined the methods they used to get an unheard-of 10 deployments a day. These methods still form the basic pillars of **development-operations** (**DevOps**) today. We will talk about the incorporation of the following:

- Tools and technology
- People and process

DevOps tools and technology

During the talk, Allspaw and Hammond identified technologies and what they did with them to align both the development and operations teams. But not only were the tools noteworthy—it was also how the teams used the tools together in a collaborative way.

These technologies included the following:

- Automated infrastructure
- Common version control
- One-button builds/deployment
- Feature flags
- Shared metrics
- **Instant messaging** (**IM**) robots on shared channels

The use of tools and technologies continues to play a key role in DevOps today. We'll explore how these tools enabled a quick release process into production and quick resolutions if problems happened in production.

Automated infrastructure

As software became more complex, the environments to execute them became more complex as well. Operations teams were faced with configuring servers that grew over time from tens to hundreds or thousands. The task required automation. **Configuration management** (**CM**) tools such as Chef and Puppet began to emerge.

CM allowed operations teams to standardize environmental configurations such as versions of an operating system, software applications, and code libraries. It allowed them to easily find those machines that did not have the standard configuration and correct them. As servers moved from physical hardware to **virtual machines** (**VMs**), CM allowed for the creation and maintenance of standard images that were differentiated by what role they played in the server environment.

CM also helps developers. Automated CM can make every server provisioned uniform between the multiple environments an organization has for software development and release. Consistency between development, staging, and production environments eliminated a problem colloquially known as "*works on my machine*" where even subtle differences between development, testing, and production environments led to the possibility that code would work on development but would fail in production.

Common version control

A version control systems (**VCS**) such as Git are is a popular tool in software development for managing source code. With version control, developers can make changes to source code on a private sandbox called a branch. When ready to share source code changes, developers can merge their changes back to the main branch of the source code repository, making sure those changes do not conflict with other changes from other developers. The VCS records all changes that happen to the source code, including those from other branches. Because the version control contains a comprehensive history of the source code's evolution, it is possible to find a version of the source code from a specific point in time.

Soon, version control became important for storing more than source code. Testing tools required test scripts and test data that could be version-controlled as the tests changed. CM tools used text files to define ideal configurations for servers and VMs. Operations could also have scripts that would perform the configuration tasks automatically. All these could be version-controlled to record the evolution of the environment.

Making sure that both development and operations were not only using version control but using the *same* version control tool became important. Having all versions of the artifacts (code, tests, configuration files, scripting, and so on) developed allowed for an easy understanding of which version of which artifact was used in a release using tags or labels. A common version control tool ensured that one side (development or operations) was not denied access to view this understanding if a problem occurred.

One-button builds/deployment

Building a release from source code could be a time-intensive task for developers. They would have to pull their changes from version control, add in the required code libraries, compile the changes together into a build package, and upload that build package into an environment to test whether the changes worked. A smart developer would often automate these tasks by setting up build scripts, but could this process be easier?

Continuous integration (**CI**) tools such as Hudson (later named Jenkins) emerged that allowed developers to go through all the steps of a build process and execute build scripts by just pushing a button. A page could easily show not only build success but if a build failure occurred, it could also show in which step of the process that failure occurred. Automating the build through CI also made sure that the build process between developers was consistent by ensuring that all steps were followed and a step was not omitted.

Could that same consistency be applied to operations teams when they deployed releases? **Continuous deployment** (**CD**) tools take a build package and run tests against it in the current level environment, and if they pass, apply it to a specific environment. These tools can also connect with **CM** tools to create instances of the environment with the new build package "on the fly." Any new deployment would be recorded showing who pushed the button, when it was pushed, and which artifacts and artifact changes were deployed to a particular environment.

Can a tool used for CI also be used for CD? This is a common way of implementing the same automation that can be used by both development and operations.

Feature flags

Flickr, the company Allspaw and Hammond worked for, was a photo-sharing and rating website. Its software differed from traditional desktop-based software because it was concerned with supporting only one release: the release on its production environment. The company did not have to worry about supporting multiple versions of the released software. This allowed it to have the main branch of its version code repository as the specific version it would support and examine if problems arose.

To handle problems introduced by buggy new features, it set up conditional branches in code called **feature flags**. Based on the value of a variable, the code for a new feature would be visible or invisible. The feature flag acts as an *on-off switch*, indicating which code is released, and thus visible, as illustrated in the following diagram:

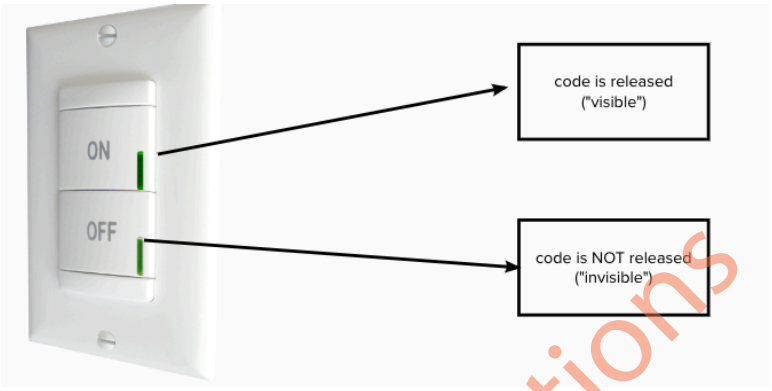


Figure 1.3 – Illustration of a feature flag

Having feature flags in the code allowed for deployments to production environments to be more flexible. Newly deployed code could be present in production but not be seen until thoroughly tested. "Dark launches" could result from this where operations could evaluate the performance of the new feature with the existing software against production data and load. Test customers could evaluate new features in a subset of the production environment where those feature flags were activated. Finally, the behavior of the environment could be quickly changed by changing the value of the feature flag(s), propagating the change through CI and CD, and allowing the change in production. This method of recovery is called *roll-forward* or *fix-forward*.

Shared metrics

To ensure stability, operations collects the performance of every environment and reviews the metrics such data collection produces. These metrics can be displayed as specific views on a dashboard. Dashboard views not only give an indication of performance now but also allow operations to identify trends and take corrective action.

Flickr made these dashboards not only for operations but also for developers as well. Developers could see the performance of the application in the context of the environment. Allowing developers access to this contextual data ensured they could see the effects of their new features and whether those features provided value.

Shared metrics also allowed for adaptive feedback loops to occur in the environment. The performance metric could be evaluated by the application, and the evaluation could generate a notification that additional resources would be required.

IM robots on shared channels

Communication between development and operations was paramount. The use of standard email was discouraged in favor of instant messaging and chat mechanisms that allowed for ongoing real-time communications of the systems between development and operations. Notifications about development events such as build status and operations events—for example, deployment status, system alerts, and monitoring messages—could be inserted into the channel by chat robots to give both development and operations personnel notice of specific events that occurred. Chats could also be searchable to provide a timeline of events for troubleshooting when problems arose.

DevOps people and processes

It's worth noting that other organizations besides Flickr were using the same tools and technologies. What made a difference to Flickr was how the people from distinct groups worked together in shared processes to leverage the tools and technologies. These people and processes formed an organizational culture that allowed them to deploy rapidly.

Allspaw and Hammond made note of specific touchpoints during the talk. These included the following:

- Respect
- Trust
- Learning from failure
- No "fingerpointing"

Having these touchpoints form that organizational culture was just as important as the application of tools and technology.

Respect

It was important that people from different groups within Flickr operated from a place of respect. That respect meant moving past stereotypes about developers or operations people and looking at common goals.

The respect extended to other people's expertise, opinions, and recommendations. There's a fundamental understanding that different people have different backgrounds and experiences that shape their opinions and responsibilities. A key part of problem-solving is to listen to those different perspectives that may give different and better solutions to a problem. Understanding the differing responsibilities allows you to understand another person's perspective.

Another important extension of that respect that Allspaw and Hammond highlighted was not just giving a response but understanding the reasons and motivations others would have for solving these problems. It's not enough to answer a question—you should also understand why the question is being asked before giving an answer. Allowing everyone to understand the context allows the group to create unique solutions to problems.

To have this respect shown, there must be transparency. Hiding information between groups does not allow for the free exchange required to create innovative solutions to problems. Also, at some point, whatever you hide will be found out, creating conflict.

An important part of respect is empathy. Knowing what effects to operations there may be from a code change is important before having that discussion with operations personnel. This allows room for any hidden assumptions to be unearthed and for creative solutions to flow.

Trust

Armed with transparency and empathy to build respect, people from one group need to trust the other groups. If a development person has that understanding of what impact to operations their feature will have, it is then incumbent on them to have that conversation with operations personnel to confirm those impacts or at least make them aware of potential impacts.

Conversely, operations people need to have developers involved to discuss together what effects any infrastructure changes will have on current or future features.

Ultimately, this comes to an understanding that everyone should trust that everyone else is doing their best for the good of the business.

Examples of the manifestation of trust are not only the sharing of data through version control, IM chat mechanisms, and metrics/dashboards, but also lie in the construction of shared runbooks and escalation plans that are created when readying a new release. The construction of these plans allows discussion to flow on risks, impacts, and responsibilities.

Finally, including mechanisms to allow the other group to operate is an important part of leveraging that trust. For developers, that meant setting up controls in software for operations people to manipulate. For operations people, that meant allowing appropriate access to the production environment so that developers could directly see the effects new changes had in the production environment.

Learning from failure

Failures will happen. How an organization deals with that failure is the difference between a successful organization and an organization that will not remain operational for long. Successful organizations focus more on how to respond to failures, foreseen and unforeseen, more than expending energy to prevent the next failure.

Preparation for responding to failure is a responsibility that falls on everyone. Each person, developer, or operations team member must know how they would react in an emergency. Ways of practicing emergencies include having junior employees "shadow" senior employees to see how they would react. At Flickr, those junior employees were put in the same "what-if" scenario as the exact outage occurred to see which solutions they could develop.

No "fingerpointing"

At Flickr, they discovered that when people were afraid of getting blamed for production failures, the first reaction would be to try to individually fix the problem, find who to blame, or conceal the evidence. That always led to a delay in finding a solution to the problem. They instituted a *no fngerpointing* rule.

The results were dramatic. Resolution times to fix problems rapidly decreased. The focus then shifted from who caused the problem to what the solution was.

The DevOps movement begins

The response to Allspaw and Hammond's talk was swift and impactful. People started to look at ways to better align development and operations. Patrick Debois, having missed the *O'Reilly Velocity* conference where Allspaw and Hammond gave their talk, organized the first *DevOpsDays* conference in Ghent, Belgium to keep the conversation happening. The conversation continues and has become a movement through successive DevOpsDays conferences, messages on Twitter highlighted with "#DevOps," blogs, and meetups.

The response drives the creation of new tools for version control, change management, CI, CD, CM, automated testing, and artifact management. Technology evolves from VMs to containers to reduce the differences between development, test, staging, and production environments.

The DevOps movement continues to grow. As with the adoption of Agile, DevOps is open to all and decentralized. There is no one way to "do DevOps." DevOps can be applied to environments of any type such as legacy mainframes, physical hardware, cloud environments, containers, and Kubernetes clusters. DevOps works in any industry, whether finance, manufacturing, or aerospace.

Since the original talk by Allspaw and Hammond, organizations that have adopted DevOps principles and practices have seen incredible gains in deployment frequency, while also being able to reduce the probability of production failures and recovery times when an errant production failure does occur. According to the 2021 *State of DevOps* report, "*elite*" organizations can release on-demand, which may happen multiple times a day. This is 973 times more frequent than organizations rated as "*low*." Elite organizations are also a third less likely to release failures and are 6,570 times faster at recovering from a failure should it occur.

Some organizations are medium-to-large-sized companies working in industries such as finance, aerospace, manufacturing, and insurance. The products they create may be systems of systems of systems. They may not know how to incorporate Agile and DevOps approaches. For these companies, one framework to consider is the **Scaled Agile Framework®** (SAFe®).

Scaling DevOps with SAFe®

SAFe® is one of the more popular adopted platforms used to incorporate the Agile mindset and practices according to recent *State of Agile* surveys, taken annually. As stated on `scaledagileframework.com`

by Scaled Agile Inc, the creator and maintainer of SAFe®, the framework is "*a knowledge base of proven integrated principles, practices, and competencies for achieving business agility using Lean, Agile, and DevOps*."

Organizations can choose to operate in one of four SAFe® configurations. Almost all organizations start with a foundational configuration called *Essential SAFe*. In Essential SAFe, 5 to 12 teams—each comprised of a Scrum Master, Product Owner, and 3 to 9 additional team members—join together to form a *team of teams* called an **Agile Release Train** (**ART**). The ART works to develop a product or solution. Guiding the work on the ART are three special roles, as outlined here:

- **Release Train Engineer** (**RTE**): This is the *Chief Scrum Master* of the ART. The RTE acts to remove impediments, facilitate ART events, and ensure that the train is executing.
- **Product Management** (**PM**): PM is responsible for guiding the evolution of the product by creating and maintaining a product vision and guiding the creation of features that go in a prioritized program backlog.
- **System Architect** (**SA**): The SA maintains the architecture of the product by creating architectural work called enablers. They act as the focal point for teams on the ART in terms of balancing emergent design from the teams with the intentional architecture beginnings of the product.

The following diagram illustrates an ART and the roles within it:

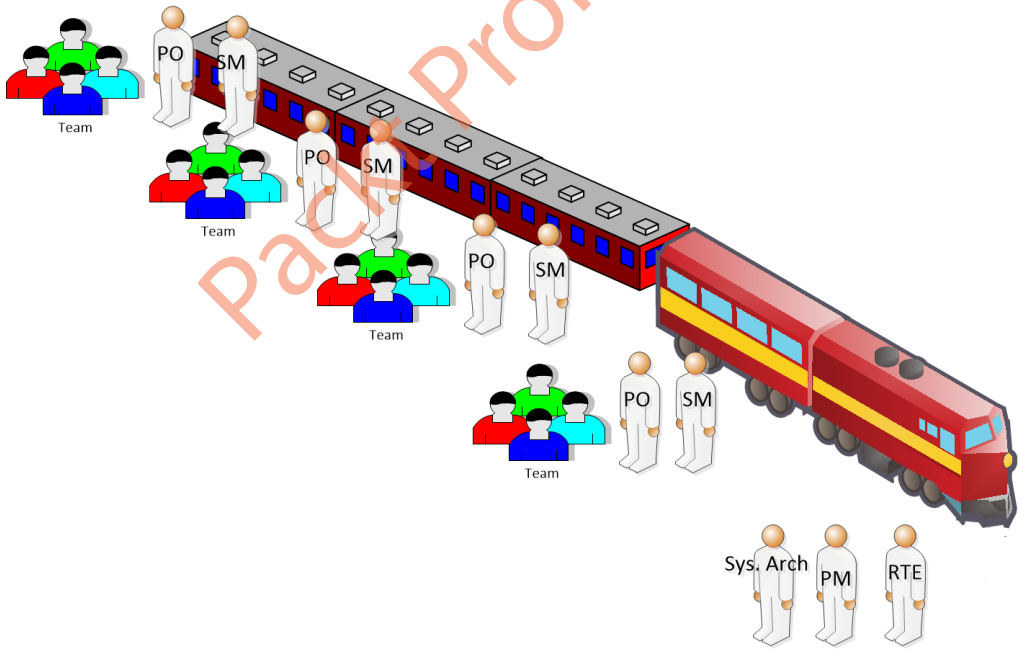


Figure 1.4 – An ART with main roles

Just as with stories, the work a team performs is timeboxed on a Scrum team, and the work of an ART is timeboxed as well. Features should be completed within a **Program Increment** (**PI**), which is a period of time between 8 to 12 weeks. The PI is a grouping of sprints where teams in the ART perform by breaking down features into stories and delivering those stories, sprint after sprint in the PI. You can see an illustration of this in the following diagram:

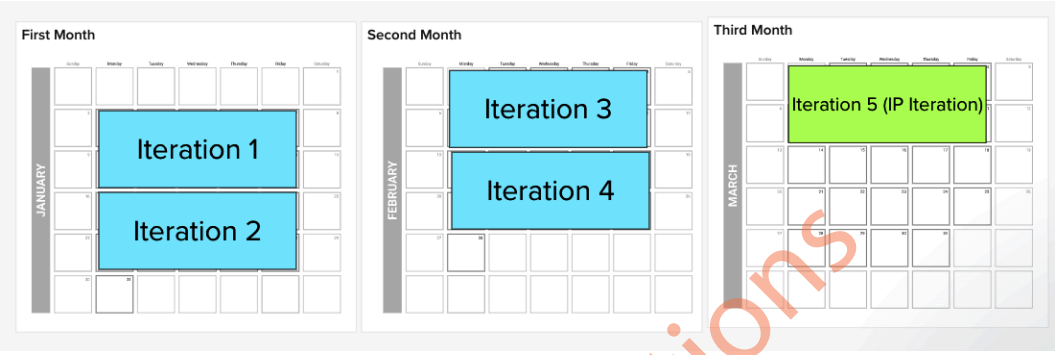


Figure 1.5 – A 10-week PI with five iterations (sprints)

It is against this backdrop of Essential SAFe and the ART that we apply DevOps. ARTs look at adopting the same practices that Allspaw and Hammond mentioned in their talk in 2009, as well as newer practices that have emerged since then. This book will cover the approach to DevOps as outlined in the SAFe. The aspects of this include the following:

- Modeling the DevOps approach using **Culture, Automation, Lean Flow, Measurement, and Recovery** (**CALMR**)
- Setting up and maintaining value streams
- Applying the **CD** pipeline against the value stream
- Including built-in quality and security in the process

Looking at CALMR

After Allspaw and Hammond's talk, people tried to organize the practices mentioned and create a model that would exemplify the DevOps approach. During *DevOpsDays 2010*, John Willis and Damon Edwards coined a **CAMS** approach, where each letter signified a significant factor or *pillar* to DevOps. The letters, and the factors they represent, are set out here:

- (**C**)ulture
- (**A**)utomation
- (**M**)easurement
- (**S**)haring

Later, Jez Humble added an **L**, for **Lean Flow**, to evolve this to the **CALMS** approach.

Scaled Agile, realizing that the desired culture would have **Sharing** as a key component, removed the **S** from its model, and replaced it with an **R**, for **Recovery**. The **CALMR** model can be summarized as such:

- **Culture**: Create a culture of shared responsibility among all groups (development, operations, security, business, and others).
- **Automation**: Leverage automation as much as you can on your Continuous Delivery Pipeline.
- **Lean Flow**: Work with small batch sizes, visualize all your work, and avoid too much **Work in Progress** (**WIP**).
- **Measurement**: Measure your flow, your quality, and your performance in all environments, and whether you are achieving value.
- **Recovery**: Create low-risk releases. Devote energy to preparing how to recover from failure.

Part 1: Approach – A Look at SAFe® and DevOps through CALMR will examine each factor of the CALMR approach and see how the teams and the entire ART use values, principles, and practices to implement these factors.

Mapping your value stream

Value streams are a concept from Lean manufacturing where you look at the holistic process of creating a product from initial conception to delivery. For a value stream, you evaluate the steps needed in the process and the people and resources involved at each step. Each step in the process has its lead time (time waiting before the step can begin) and its cycle time (time spent on each step).

The first part of organizing value streams is to identify the present state of the value stream. Each step—as well as the people, resources, and lead and cycle times—is determined to identify and map the entire value stream. After this identification, questions are raised on solutions that can be employed to reduce time through the Value Stream steps.

After the initial identification, the next step is to identify and amplify the feedback loops that each step may require. Metrics play a valuable part here to see whether the value stream, as realized by an ART, is executing its process, whether a solution under development has problems in any environment, and whether a solution is delivering its promised value.

At this point, taking the first step of value-stream identification and mapping and the second step of finding feedback for each step yields the third step of value-stream management. During the initial value-stream mapping exercise, a potential "future state" or optimized value stream may be identified. It's up to the ART to make incremental changes to obtain this optimal value stream. Only by adopting an attitude of continuous learning and working toward continuous improvement can they do that.

Part 2: Implement – Moving towards Value Streams dives into the three ways of doing value-stream management, modeled after the *Three Ways* identified in *The Phoenix Project*. We will examine the steps to identify a value stream and map an initial and potential future value stream. We will see how metrics form feedback loops for steps in the value stream. Finally, we will evaluate tools and techniques from Lean thinking to improve the value stream in the context of Continuous Improvement.

Running your value stream through the Continuous Delivery Pipeline

A Continuous Delivery Pipeline is the implementation of the value stream for an ART. It marries the people and their functions, with the process of delivering products from initial concept to release, and includes the tools to automate tasks, mostly in the form of a CI/CD pipeline. In SAFe, the Continuous Delivery Pipeline is divided up into four aspects. Each aspect is run concurrently by members of the ART throughout each PI, as illustrated in the following diagram:

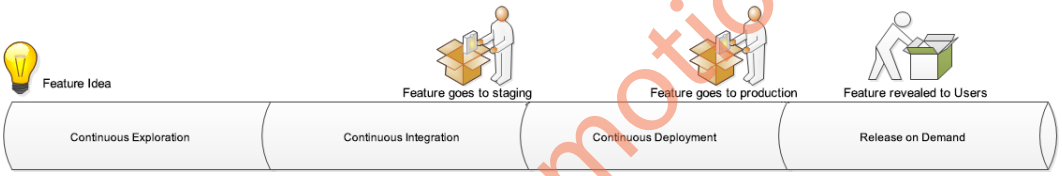


Figure 1.6 – The Continuous Delivery Pipeline

The first phase is **Continuous Exploration** (**CE**). In this phase, PM works with customers, stakeholders, UX, the SA, and other groups such as compliance and security to determine upcoming features based on hypotheses of benefits to the customer that will show real value. These features are examined to determine feasibility and what, if any, changes to the architecture are required to meet **Non-Functional Requirements** (**NFRs**) such as security, reliability, performance, and compliance. After this definition and refinement, the feature is placed in the Program Backlog and prioritized for inclusion in an upcoming PI.

During the execution of a PI, development teams will incorporate the second phase of the Continuous Delivery Pipeline: **CI**. Once code changes have entered version control, the CI/CD pipeline comes into play. The pipeline may run several layers of testing, including linting to examine code quality and unit testing to examine proper functionality. If the tests pass, the code changes may be merged into a higher-level branch and a package created. That package may undergo additional testing to validate the correct behavior of the system. Passing that testing, the package that includes changes may be deployed—automatically, if possible—into a staging environment that resembles production.

Depending on the organization, the third stage, **CD**, may be automated and take over. Deployment of the package created in the CI phase may be performed automatically in the production environment. Feature flags may prevent changes from being released as testing continues to ensure the changes

will work with existing functionality and in the production environment. Measurements continue to be taken in the production environment by continuous monitoring to verify proper operation and response to problems in production happen here.

Finally, in the fourth phase, **Release on Demand**, the new feature is enabled, allowing customers to take advantage of the changes. The production environment continues to be monitored for adverse effects, and the ART continues to respond to any deployment failures that happen. Measurement here includes the evaluation of leading indicator metrics to evaluate the amount of value truly delivered by the new feature and whether the initial hypothesis is true or false. Finally, the ART reflects and applies lessons learned to improve both the Continuous Delivery Pipeline, and the value stream.

We will examine all four stages of the Continuous Delivery Pipeline in *Part 3: Optimize – Enabling a Continuous Delivery Pipeline*. We will examine the people and processes that happen with CE. We will investigate the tools and technologies that make up CI and CD. Finally, we will see how Release on Demand closes the loop for our ART in delivering value to our customers.

Including security in the process

There is a growing realization that collaborating with and involving other groups in the organization can improve product quality and speed up product development. One such group collaborating with development and operations is security.

DevSecOps is a growing trend in DevOps circles where information security practices are folded in throughout the continuous delivery process. In SAFe, we include the information security practices espoused by DevSecOps so that security is not considered an afterthought.

Throughout *Parts 1 to 3*, you will see where security has active involvement and where continuous testing is performed so that solutions comply with any approvals that are mandated at the end of development. These secure solutions have long been incorporated into product design, development, and deployment.

Summary

In this chapter, we looked at the problems many organizations face when developing products today. We saw how the modern pressures of faster TTM, changing requirements and unknown customer desires, and problems in production deployments wear out both the development and operations groups. We also looked at the responses that development and operations groups have created.

Development began to look at incorporating an Agile mindset to allow for quick, frequent releases of small increments of value that would allow customer feedback to drive the next development increment. This outlook required an examination of values and principles to change the mindset as well as the incorporation of Lean thinking from the manufacturing world.

As development began to reap the benefits of the change to an Agile mindset and incorporate Agile practices, the bottlenecks for the release of new products and new functionality fell to operations—those that maintained the existing production environment. A new way of working together, the DevOps movement, sought to tear down the walls of confusion between development and operations through the application of tools and technology and by setting up a culture based on respect, trust, empathy, and transparency.

One approach that incorporates DevOps principles and practices in development is SAFe. DevOps using SAFe is employed for a *team of teams* or an ART. The ART embraces DevOps by adopting the CALMR model. The team identifies the way it works and maps that as a value stream. Finally, it employs a Continuous Delivery Pipeline to deliver its work to production, measure its worth, and improve the value stream.

In our next chapter, we will begin our examination of the CALMR approach by looking at the first and most key factor: culture.

Questions

Test your knowledge of the concepts in this chapter by answering these questions.

1. What does the M stand for in CALMR?

   A. Monitoring
   B. Multitasking
   C. Measurement
   D. Mission

2. Which are phases in the Continuous Delivery Pipeline (pick two)?

   A. Continuous Improvement
   B. Release on Time
   C. Continuous Exploration
   D. Release on Demand
   E. Continuous Delivery

3. What kind of culture is important in CALMR?

   A. Independent
   B. Shared responsibility
   C. Bureaucratic
   D. Open

4. What has been the traditional focus for operations?

   A. Revenue
   B. Stability
   C. Velocity
   D. Compliance

5. Which term describes incorporating information security practices in CD?

   A. SecureOps
   B. DevSecurity
   C. DevSecOps
   D. OpSec

Further reading

For more information, refer to the following resources:

- `https://leadinganswers.typepad.com/leading_answers/files/original_waterfall_paper_winston_royce.pdf`—The original paper by Winston W. Royce diagramming what has come to be known as the Waterfall method. Note that in the paper, he advocates alternate paths to allow for additional testing and customer feedback.
- `https://agilemanifesto.org`—Manifesto for Agile Software Development, or quite simply, the Agile Manifesto.
- *Lean Software Development: An Agile Toolkit* by *Mary Poppendieck and Tom Poppendieck*.
- *Kanban: Successful Evolutionary Change for Your Technology Business* by *David J. Anderson*.
- `https://www.youtube.com/watch?v=LdOe18KhtT4`—A recording of the *10+ Deploys Per Day: Dev and Ops Cooperation at Flickr* talk given by John Allspaw and Paul Hammond at the 2009 *O'Reilly Velocity* conference that ushered in the DevOps movement.
- `https://cloud.google.com/blog/products/devops-sre/announcing-dora-2021-accelerate-state-of-devops-report`—Findings from the *2021 Accelerate State of DevOps* report.

Part 1
Approach – A Look at DevOps and SAFe® through CALMR

A year after the Flickr presentation at the O'Reilly Velocity conference and the initial DevOpsDays conference, John Willis and Damon Edwards tried to define the important elements of this new movement called DevOps at the 2010 DevOpsDays conference. They eventually settled on **Culture, Automation, Measuring, and Sharing** (**CAMS**).

CAMS remained the approach to DevOps until Jez Humble decided that a Lean Flow was also essential to DevOps and needed to be added to the model. CAMS then became CALMS.

Scaled Agile incorporated DevOps into the Scaled Agile Framework® in 2018. When they did this, they evaluated the current CALMS model and made a realization: sharing is an important part of culture. By establishing that the Culture was one of shared responsibility, it defined the kind of Culture needed for DevOps. It also allowed Recovery to be added to the model, which became CALMR.

In *Part 1*, we will examine Scaled Agile's CALMR approach to DevOps. We will examine what characteristics form a Culture of Shared Responsibility. We will then examine the kinds of technology used for automation and who is responsible for setting them up. We will look at how a Lean Flow allows us to deploy quickly and with a high level of quality. We will then look at ensuring quality and security by continuously measuring the progress, correctness, and value of the product under development. Finally, we will look at preventing production failures and what corrective measures we can take if a production failure does occur.

This part of the book comprises the following chapters:

- *Chapter 2, Culture of Shared Responsibility*
- *Chapter 3, Automation for Efficiency and Quality*
- *Chapter 4, Leveraging Lean Flow to Keep the Work Moving*
- *Chapter 5, Measuring the Process and Solution*
- *Chapter 6, Recovering from Production Failures*

2

Culture of Shared Responsibility

"*Culture eats strategy for breakfast.*"

This quote from management expert Peter Drucker emphasized the impact culture has on what organizations strive to achieve. In the previous chapter, we saw how culture (people, their behavior, and the structures that encourage their behavior in organizations) provides the underpinnings of processes and tools. So, if culture is so important, what's the best culture and how do we attain that if we find that our present culture is lacking?

We find that structure and behavior will determine culture. We will first examine different cultures based on a classic organizational culture model. This examination of organizational cultures will include traits and characteristics and how each handles information flow. We then look at how to effect cultural change toward a more desired culture by changing behavior.

Armed with this knowledge of the ideal organizational culture and how to effect change, we will examine the structure of culture that the SAFe® promotes. The first part is identifying how both Lean thinking and the Agile Manifesto play a role in developing the proper mindset.

With this mindset at play, we will take a close look at the principles in SAFe® that provide context not only to the structure but also to the practices put into play to optimize Lean and Agile development.

Finally, we will take an initial look at value streams: the structures that will foster that ideal culture. We will see how value streams build upon the Lean-Agile mindset and SAFe principles to allow for effective cultural change.

In a nutshell, we will be covering the following topics:

- A culture for organizational change
- Lean-Agile mindset
- SAFe principles
- Value streams

A culture for organizational change

Every community of people, from the smallest of teams to the largest of nations, will have a culture—a thread that serves as the identity of the community. A community uses its culture to identify its norms and indicate what makes that community different from other communities.

It is the organization's responsibility to determine whether its culture is servicing the needs of the organization and allowing it to grow and prosper. The first part of this is the self-inspection of whether the culture is beneficial to the organization. After that analysis, the organization can decide on actions to change the culture.

What kind of culture?

In 1988, Ron Westrum was studying how to improve safety on medical teams when he came upon the idea of examining the culture of those teams. He looked at how these organizations handled information and came up with a typology consisting of three types of cultures. The cultures are identified as the following:

- Pathological
- Bureaucratic
- Generative

According to Westrum, a pathological culture is patterned by a preoccupation with personal power and glory for the organization's leader. Information is used as leverage for political power. Fear and threats are commonly used as motivation by leaders to accomplish (the leader's) goals.

In a bureaucratic culture, the organization looks to rules, positions, and departmental turf as its main qualities. Information may be shared through prescribed channels and procedures, but only within local boundaries.

An organization with a generative culture is focused on alignment with the organization's mission. Information flows freely to whoever can assist with the mission.

Westrum identified examples of the characteristics of how different cultures deal with different kinds of information. These are shown here:

	Pathological	**Bureaucratic**	**Generative**
Focus	Power-oriented	Rule-oriented	Performance-oriented
Cooperation	Low	Medium	High
How do we deal with messengers of bad news	Blamed	Neglected	Trained

Handling responsibilities/risks	Shirked	Narrowed	Shared
Communication across organizations	Discouraged	Tolerated	Encouraged
How failure is handled	Scapegoating	Seek justice	Inquiry and learning
How new information is applied	Crushed	Discouraged as it may lead to problems	Implemented

Table 2.1 – How organizations handle information by culture

A key part of Westrum's model of information flow dealt with how different cultures handle anomalies or problems discovered. How do organizations react to a negative discovery? Westrum identified six responses, as follows:

- **Suppression**: Preventing the person from spreading the word about the discovery
- **Encapsulation**: Ignoring the person who discovered the bad finding
- **Public relations**: Minimizing the impact of the discovery
- **Local fix**: Fixing only the immediate problem with no investigation of related issues
- **Global fix**: Fixing the problem wherever it occurs
- **Inquiry**: Thorough investigation of the root cause

The range of responses and the cultures that typically generate them are seen in the following diagram:

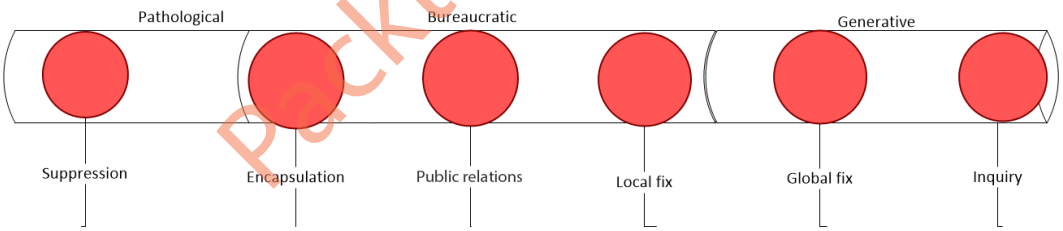


Figure 2.1 – Range of responses to anomalies by culture

According to Westrum, the free flow of information in a generative culture fosters three things: alignment, awareness, and empowerment. These three factors are important in setting a mission and working toward goals.

Because information flows freely to all members in a generative culture, awareness comes easily. The organization's goals are transparent and visible to all. Actions by other members within the organization and outside of the organization are communicated so that dependencies between efforts are effectively managed. So, this awareness is not only local to the organization but also exhibits a broader point of view.

Alignment comes from this awareness by having a visible goal communicated throughout the organization. This allows all people, at all levels in a generative culture, to *buy into* the goals of the culture.

With a generative culture where everyone knows what the organization's mission is and all members are aligned toward achieving the organization's goals, everyone is encouraged to speak up, think outside of their roles and responsibilities, and fully participate in continuous inquiry.

In the end, Westrum concludes that a generative culture, with a free information flow and information processing, allows for fundamental long-lasting improvements to the system as opposed to *quick fixes*. He notes that culture is mutable and that organizations can move from one type of culture to another.

A generative culture with the traits identified by Westrum has benefits that relate to a **development-operations** (**DevOps**) approach. In *Accelerate: Building and Scaling High Performing Technology Organizations*, the authors discovered that an organizational culture that was generative produced higher levels of software delivery performance and organizational performance. In addition, organizations with a generative culture experienced higher levels of job satisfaction.

How do we change the culture?

Culture follows the structure of an organization as well as its behavior. To change an organization's culture, it stands to reason those changes have to be made both in the structure of the organization and in the behavioral norms that the organization demonstrates.

Although it is important to recognize the values and principles that determine the practices, true behavioral change can only come from applying new practices and allowing continued repetition so that they become habits. When these habits are rewarded and continued, they become a behavior.

One model of fostering the behavioral changes that move cultures comes from *Leading Change* by John Kotter. In the book, Kotter proposes an eight-step method for driving the transformation of a culture. This method is outlined next.

Creating a sense of urgency

Changing a culture is never easy. People may be set on certain behaviors and unwilling to adopt new behaviors. Often, there needs to be a reason to change, one that is urgent enough to overcome all barriers. These urgent reasons may comprise the following:

- **A burning platform**: A realization that the current methods have ceased to work and that a change is necessary
- **Proactive leadership**: New or forward-thinking leadership that is marshaling the change to a better future

In 2006, Ford Motor Company was beset with problems, not only from reduced market share from Japanese and Korean imports but also from internal squabbles. They had lost **United States dollars** (**USD**)

$17 billion that year and their debts were rated at *junk* status. Bill Ford, the great-grandson of the founder, Henry Ford, called in Alan Mulally, the current **chief executive officer** (**CEO**) of Boeing to see whether he could turn the company around.

Mulally set to work. He created a weekly **Business Plan Review** (**BPR**) where leaders would share whether they were *green*, *yellow*, or *red* on their top five business priorities. Every leader indicated they were green until Mulally said: "*We are about to lose $17 billion this year, and you are saying that everything is OK? Did we plan to lose $17 billion this year?*" His push for transparency shook the leaders at Ford to allow further transformative actions.

Guiding a powerful coalition

People cannot make changes on their own. They need to find or create allies that see the same problems and will align on the changes needed to overcome those problems. These people may have come to the same conclusion or are willing to be early adopters.

As Alan Mulally was getting started as the CEO of Ford, he was determined not to *clean the house* of the present Ford executives. Some members of his executive team were longstanding employees of Ford who had ideas that meshed with the changes that Mulally was trying to make. Notable additions included Derrick Kuzak, who would become the **vice president** (**VP**) of Global Product Development, and Bennie Fowler, the eventual VP of Global Quality.

Creating a vision

What does the future state look like? Which things are important to have in that future state? The idea is not only to understand why change is necessary but what you are changing to. Kotter notes that establishing that vision for change is the responsibility of leadership. Creating a vision serves three purposes, as set out here:

- **Mission**: This sets the *why* and provides clarity for everyone
- **Motivation**: This moves people in the proper direction
- **Alignment**: The movements of people are coordinated toward the goal

One of the first actions that Alan Mulally detailed was what he wanted the Ford Motor Company to be. He eventually called this plan *One Ford*. The *One Ford* plan wanted to accomplish the following:

- Radically restructure Ford's manufacturing capability to match the capacity to the demand
- Quickly design new vehicles that sated consumer appetites
- Ensure the plan was funded and would ensure economic viability
- Work at a global level as one Ford Motor Company, in contrast to the regional silos at the time

Communicating that vision

Once you have the vision, it's important that everyone in the organization gets the same message. That message should be clear and free from jargon. Use evocative images and metaphors to fire up imagination. Consistent repetition sets the tone and gives permanence. Be prepared for feedback. Leaders will also be held accountable for behavior that may seem hypocritical in the face of that vision.

Ford had several audiences to which Alan Mulally had to communicate his *One Ford* vision. He had to broadly send out his message to employees, to the network of Ford dealerships, and to his suppliers and stockholders, which included the descendants of Henry Ford. Mulally accomplished this using several methods, from speeches and press conferences, to make sure each employee received a *One Ford* wallet card with all the points of that vision.

Empowering others to enact the vision

Once the vision is public, it's up to those in the organization to determine how to enact that vision. Leaders give them the autonomy to make those behavioral changes and apply new practices that line up with that vision. Leaders can also enable support for such changes including training. Leaders also remove any structural barriers that would encourage resistance or prevent change.

Even as Mulally was preparing his *One Ford* vision, needed actions were already being accomplished. Mark Fields, then president of the Americas, implemented *The Way Forward*, which was accelerated when Mulally became CEO. This plan resized Ford by not only closing plants—which allowed Ford to shed inefficient and unprofitable car models—but also re-aligning production lines. Derrick Kuzak set up a new design process called the **Global Product Development System** (**GPDS**), which allowed the creation of new car models utilizing a global platform.

Generating short-term wins

The changes will bear fruit. It's up to leadership to identify and celebrate the wins that emerge, no matter how small. According to Kotter, acknowledging short-term wins has the following effects:

- Provides evidence of the efforts
- Rewards those responsible for the change
- Allows adjustment of the strategies
- Silences cynics and resisters
- Keeps leadership on board
- Builds momentum

Consolidating wins to drive more change

At this point, be careful not to slip into complacency. Work still must be continued; otherwise, the long-term effort will stall. Kotter identified the following hallmarks needed at this point for sustained long-range change:

- More change, not less
- Additional help brought in
- More leadership coming from senior management
- Leadership also coming from further down in the hierarchy
- Eliminating unnecessary interdependencies

As Mulally's plan to turn around Ford was implemented and started to return results, a global recession caused by the mortgage crisis threatened the entire auto industry. Ford continued with its plan for survival and was the only American auto manufacturer not to accept a government bailout.

Anchoring new changes in the culture

The changes that come and the successes generated are building blocks to a new culture. Each new acknowledgment turns changes into habits and builds habits into desired behavior until it becomes a part of the culture.

The changes introduced by Alan Mulally have stayed with Ford, long after Mulally retired in 2014. The mindset of *One Ford* still echoes throughout Ford. The development process is a global, unified view as opposed to the regionalized views of different **business units** (**BUs**). One of the practices that Mulally introduced, the BPR, spread down from the executive level to teams and is still being used.

Now that we've seen a successful way to change to a generative culture, let's take a closer look at the parts of that culture that an **Agile Release Train** (**ART**) will exhibit.

Lean-Agile mindset

An important part of the culture in SAFe that the practices try to engender is a combination of Lean thinking with Agile development. This combination is known as the Lean-Agile mindset.

One of the two parts of this mindset looks at how an organization can eliminate waste to focus on what's necessary. This part looks at Lean thinking to accomplish that.

The second part of this mindset looks to ensure incremental delivery happens. For this, we look at the Manifesto for Agile Software Development for guidance. Meeting the needs of an organization in SAFe requires a close examination and adjustment of the Agile Manifesto. We'll soon look at these adjustments.

With this mindset in place, we need to examine what is important in SAFe. To that end, we will examine the core values of SAFe.

The SAFe House of Lean

Lean thinking has its roots in the **Toyota Production System** (**TPS**), created by Taiichi Ohno, with some inspiration from W. Edwards Deming. In the TPS, important practices and priorities are arranged like

a house, to emphasize the concepts and practices that serve as the foundation, as well as the support. The goal of the TPS serves as the roof:

"Best Quality-Lowest Cost-Shortest Lead Time-Best Safety-High Morale"

The SAFe summarizes its Lean thinking approach using a similar House of Lean model paradigm. This model is shown here:

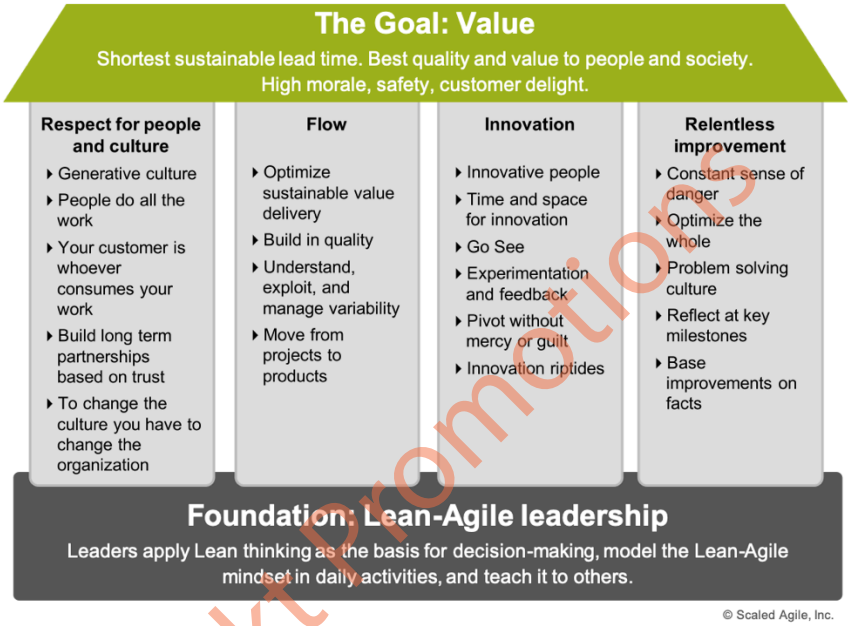


Figure 2.2 – The SAFe House of Lean (©Scaled Agile, Inc. All rights reserved)

Let's look at each part of this house, as follows:

- **The roof (value)**: The goal we are trying to achieve is value. This value is accomplished in the shortest lead time with the highest quality. This brings delight to the customer and may even change society for the better. Employees benefit from an environment that has high morale and concern for their safety.
- **The pillars**:
  - **Respect for people and culture**: We look to work in collaboration with others in a generative culture.
  - **Flow**: We look to set up a continuous flow of work so that we can continuously deliver value.

  - **Innovation**: We need time and space to be creative, to let our imaginations soar, and to examine *what-if* scenarios with our solutions. Without this time and space, our thinking becomes stunted with concern for the next emergency. This is often referred to as *tyranny of the urgent*.
  - **Relentless improvement**: We look to improve. We understand there is a *hidden sense of danger* from competition that's not only established but also represented by the next *disruptor* that comes with the advent of new technology.

- **The foundation (leadership)**: We cannot set up our house without effective leadership, one that creates a generative culture where everyone has a voice, encourages the flow, sets a time and place for innovation, and looks for opportunities to relentlessly improve.

We've seen what's involved in the Lean aspect of our Lean-Agile mindset. Let's look at the other half by seeing whether any adjustments need to be made to the Agile Manifesto.

Adjusting the Agile Manifesto

In *Chapter 1*, *Introducing SAFe® and DevOps*, we had an initial look at the Agile Manifesto. Given that the original context of the manifesto was initially aimed at small teams of developers, it makes sense that in a differing context of a team of teams, the values and principles may need to be re-examined and adjusted if necessary.

In examining the value statements, we find that the truths that lie with them don't change with the context. We still value the items on the left more than the items on the right. This is true for a small team or a larger ART.

Some principles may need some further consideration. In particular, SAFe asks you to consider the applicability of principles 2, 4, 6, and 11 from the Agile Manifesto.

Principle 2 of the Agile Manifesto states: "*Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage*." However, products that use a combination of hardware and software may require balance in accepting changing requirements late in development. Customizable software may allow for changing requirements, but requirement changes that need new hardware may be difficult and costly to fulfill once deployed.

Principle 4 of the Agile Manifesto states: "*Business people and developers must work together daily throughout the project.*" In addition to the Product Owner role that works with the other team members on an Agile team, SAFe includes other roles from the business side. **Product Management** (**PM**) works with every team's Product Owner to define a solution for the ART. Business Owners act as key stakeholders for the ART.

Principle 6 of the Agile Manifesto states: "*The most efficient and effective method of conveying information to and within a development is face-to-face conversation.*" Many of the events held by an ART such as **Program Increment** (**PI**) Planning and Inspect and Adapt were originally held primarily as face-to-

face events. With the proliferation of globally distributed development and in the face of a historic global pandemic, technological alternatives using web and video conferencing are starting to emerge. As life returns to normal, hybrid methods of meeting and collaborating may emerge.

And finally, *Principle 11* of the Agile Manifesto states: "*The best architectures, requirements, and designs emerge from self-organizing teams.*" Here, when developing complex systems of systems, where the ART has 5 to 12 teams, there is a need for balance between 5 to 12 emergent architectures and a single intentional architectural voice. That balance is provided by the System Architect, the role responsible for a product's architecture and the guide for all the enablers that teams work on.

With the proper mindset identified, we will now take a look at core values important to an ART.

The SAFe core values

SAFe has four important core values enabled by practitioners on an ART with a Lean-Agile mindset. These values are listed here:

- **Alignment**: In a generative culture with a mission, all participants work together to achieve that mission. An ART that has a generative culture will have teams that will align with other teams on that train.
- **Transparency**: A generative culture creates transparency by default. This transparency is a key part of ensuring alignment and fostering a generative culture.
- **Program execution**: The generative culture is focused on the mission. The entire ART that is transparent and aligned will work to fulfill the ART's vision and deliver the right solution.
- **Built-in quality**: Defects and failures impair the ability of the ART to reliably deliver on the solution and keep the ART's vision. A vigilant attitude to maintaining quality by detecting and removing defects throughout development is necessary to keep the ART on track.

We now know the important qualities that are important for an ART. Let's examine how to apply those values in terms of principles.

SAFe principles

The SAFe can be applied to different organizations in all kinds of industries. Differences between organizations and industries may be various and may make it difficult to move toward a generative culture. We may need a guide to align our practices so that the core values are developed and maintained.

10 SAFe principles are there to align practices to the core values. Let's see how they can be applied to DevOps.

Taking an economic view

If in our SAFe House of Lean, we are looking at a goal of value, so we want to make sure we obtain that value growth incrementally and consistently. To do that, we need to make sure our decisions come from an economic context.

Donald Reinertsen identified an economic framework needed for incremental value delivery in his book, *The Principles of Product Development Flow: Second Generation Lean Product Development*. Major components adopted by SAFe include the following:

- **Operating within Lean budgets and guardrails**: Decisions should be made by those closest to the information, but boundaries regarding the decisions can be made at higher levels to ensure the necessary governance and oversight are still applied.
- **Understanding economic trade-offs**: As the ART needs to make decisions, it should be aware of the considerations that these decisions will have. These factors are development expense, lead time, product cost, value, and risk.
- **Leveraging suppliers**: Often, a decision may come to *build versus buy*. An organization may look to vendors for reasons such as insufficient capacity in its workforce, or that vendor may have a specialized skill set or be the only supplier of a specific component.
- **Sequencing jobs for maximum benefit**: ARTs cannot do everything at once. They need to prioritize what gives the maximum value soonest. Reinertsen recommends using **Cost of Delay** (**CoD**), the cost the organization incurs if it doesn't deliver the value at the right time, over other measures such as **Return on Investment** (**ROI**) or executive decision, often lampooned as the **Highest-Paid Person's Opinion** (**HiPPO**). SAFe takes this a step further by dividing the CoD by the size or duration of the job to give a *bang-for-the-buck* measure called **Weighted Shortest Job First** (**WSJF**).

Applying systems thinking

When we look at developing complex products encompassing systems of systems such as cyber-physical solutions, we need to employ a holistic view of the product. But that's not the only system needing attention.

What's often ignored is the system that is the organization. In 1967, Melvin Conway proposed the following idea, which is now known as Conway's Law:

"*An organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.*"

In other words, the way an organization develops a product will be mirrored in the architecture of the finished product.

Because of the applicability of Conway's Law, to optimize the final architecture, we need to look at a better way of developing the product. That allows us to look at and optimize our value stream.

Assuming variability and preserving options

We want to make sure we keep the spirit of accepting requirements anytime during development, as found in *Principle 2* of the Agile Manifesto. A question then emerges: how can we keep the spirit of this principle intact?

We want to look at how requirements commonly change. Usually, at the beginning of development, there are a lot of unknowns. As development progresses, learning happens and the unknowns become knowns.

Learning also helps with identifying design options that will work with the requirements and those that won't. The combinations of requirements, design options, and time result in a *cone of uncertainty*, as seen here:

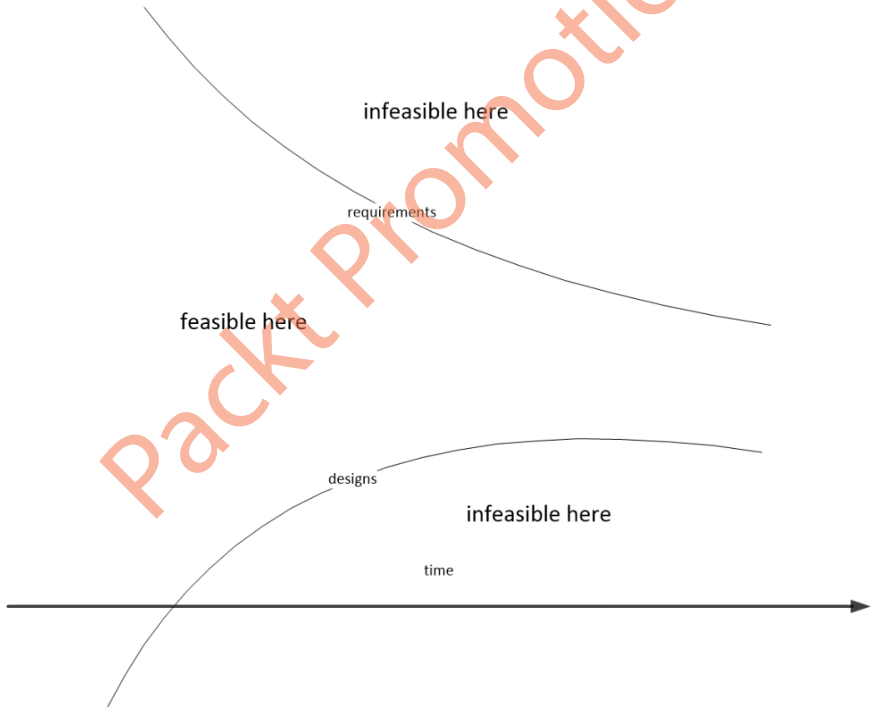


Figure 2.3 – Cone of uncertainty

To navigate the cone of uncertainty, it's best to keep requirements flexible and have several design options (commonly referred to as **set-based design** (S**BD**)) available in the early stages. As time progresses, the organization learns and finds which requirements don't apply and which design options are infeasible.

Building incrementally with fast, integrated learning cycles

Incremental delivery is ultimately about learning. Producing an increment of value allows feedback from customers, which allows organizations to move on track to deliver more value. This incremental cycle of learning also allows the organization to find which of its design options are infeasible according to new discoveries in the environment.

Each team on an ART learns through its development cycles. To unify that learning, it needs to frequently integrate its work, test the integration, and seek feedback from the entire system. This integration should happen at least as frequently as its learning cycles.

Basing milestones on an objective evaluation of working systems

Many organizations that use the Waterfall method set up phase-gate milestones to ensure that everything is ready for the next phase and reduce risks. To move on to the next phase, the milestone is there to see whether the deliverables for that phase are ready and complete.

Phase-gate milestones do not handle risks because phase-gates will emphasize working as much as possible on a single-design approach. The rush to pass a milestone does not allow learning to happen to ensure that a solution is still within the cone of uncertainty. You may not realize that a solution is infeasible until it is too late.

SAFe looks at having periodic milestones. These milestones are based on the feedback and learning of each increment of value and of the integrated solution at that point in time.

Visualizing and limiting WIP, reducing batch sizes, and managing queue lengths

As we arrange the development using value streams, we want to make sure that flow occurs on the value stream to ensure continuous delivery of value. Ensuring flow relies on three key actions, as set out here:

- Visualizing and limiting **Work in Progress/Process** (**WIP**)
- Making sure we have small batch sizes
- Managing our queue lengths

We will examine these factors and look at achieving flow in more detail in *Chapter 4*, *Leveraging Lean Flow to Keep the Work Moving*.

Applying cadence – synchronizing with cross-domain planning

One of the SAFe core values we identified before was alignment. This value is important as we have multiple teams on our ART, and we want to make sure every member of every team is working in lockstep toward the ART's vision.

To do this, teams on the ART apply both cadence and synchronization. Both are required to make sure that there is a balance between the inherent uncertainty in development and the current plan of the ART, allowing necessary pivots.

Cadence on the ART means that the teams have learning/development cycles of identical lengths. This constant length serves as a *drumbeat* of development. With a constant length, things can happen at a routine and predictable pace.

Synchronization on the ART means that the teams start and stop their learning/development cycles at the same time. This allows for the integration of the entire system to happen between teams.

The application of cadence and synchronization is illustrated in the following diagram:

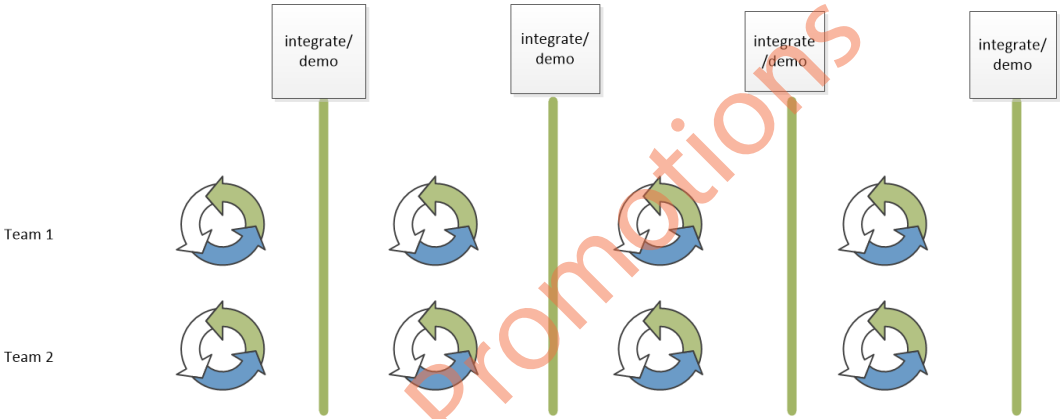


Figure 2.4 – Cadence and synchronization with multiple teams

One final part of applying cadence and synchronization is cross-domain planning, which on an ART occurs at the beginning of every PI. There, all teams meet, along with the business stakeholders, shared services, architects, PM, and the **Release Train Engineer** (**RTE**) to align with the ART's mission for the PI. Having this planning occur at a cadence allows all teams to inspect where reality deviates from what was originally planned and to adjust. This limits variability, which leads to rework and other waste, to a single learning cycle.

Unlocking the intrinsic motivation of knowledge workers

A generative culture welcomes the input and initiative of everyone in the organization. Empowered individuals working together toward a mission build that generative culture. But what does that empowered individual look like, and is a generative culture good for them?

Peter Drucker defines knowledge workers as "*individuals who know more about the work that they perform than their bosses*." They are often the ones closest to the information of whether a solution

will give value to a customer. These are just the kind of people our generative culture will need. How can we ensure they will stay and be actively engaged?

Daniel Pink, in his book *Drive: The Surprising Truth of What Motivates Us*, upended expectations when he found that monetary compensation worked, but only up to a certain point. After money, what really motivated people were these things:

- **Autonomy**: People want to have the freedom to explore the best solutions and self-direct the work they want to do
- **Mastery**: People want to improve their skills and build expertise
- **Purpose**: People want validation that the work they do has meaning

By being cognizant of the people in our organization and what really motivates them, we can ensure we build toward that generative culture.

Decentralizing decision-making

When optimizing lead time, organizations become cognizant of things that may cause delays. One of these sources of delay can be the escalation of problems and waiting for a decision.

With a generative culture with empowered individuals, it can be possible for most decisions to be made by the teams. Some decisions that are strategic in nature do have to be made at higher levels in the organization. SAFe recommends that these decisions to be escalated are like this:

- **Infrequent**: These decisions are not made often
- **Long-lasting**: The impact of the decision will last for a long time
- **Incorporate significant economies of scale**: The decision may affect the entire organization

Then come those decisions that are more of a tactical nature and that teams can make, and shouldn't be constantly made by leadership, where the responsibility of leadership is making strategic decisions. In short, decentralized decisions are like this:

- **Frequent**: These are common decisions that must be made often
- **Time-critical**: These decisions have a high CoD
- **Require local information**: These decisions have information that is readily available to the team in the environment

Organizing around value

After going through the first nine SAFe principles, we want to set up the structure of our organization so that all the principles are reflected, allowing for the creation of a generative culture.

We've seen that we need to look at the economic factors when looking at delivering value. We've also seen that for our system, we must be cognizant of the communication links that mirror our architecture. We want to motivate our knowledge workers and allow them to make the necessary tactical decisions to accelerate value delivery.

The structure we want should mirror the development process while ensuring that handoffs occur with minimal delay. The structure also should work in small learning cycles and ensure a flow of value occurs.

In short, we want value streams instead of traditional hierarchical silos. In large organizations, there may be different departments organized to create stability. This stability is still necessary to handle large efficiencies of scale. How can this paradox still be maintained?

In SAFe, the value stream is seen as a *network*, getting the needed people and binding them together so that they work together to deliver value in the quickest way possible. The organization's hierarchy is still there as a *second operating system*. The two structures, modeled after the *dual operating system* discussed in John Kotter's book, *Accelerate (XLR8): Building Strategic Agility for a Faster-Moving World*, stand together as equals in the organization, each needed for different reasons.

We look at that *network*, the value stream, in the rest of this book. We will see how to identify and map our value stream, how to turn that into an ART, and then how to use a **continuous delivery** (**CD**) pipeline to assist the value stream. Let's start by taking a close look at what a value stream is.

Value streams

We saw that SAFe principle *#10* talked about organizing around value. To do that, our organization must set up and use value streams to make sure we consistently deliver value to the customer. But what precisely is a value stream?

The classic definition of a value stream comes from Lean thinking It describes the steps, the people that perform those steps, and the times associated with each step. It then becomes a platform for optimization so that an organization can reduce lead times.

SAFe takes this classic definition and applies it in two contexts. The first context is an operational value stream that describes the interaction the customer or end user has with the organization and which products are used in that interaction. The second context is a development value stream that describes the process of development of a product or solution.

We will first examine the classic definition and then move to how they have evolved into operational and development value streams.

Classic value streams

If we trace the origins of Lean thinking to manufacturing, a value stream was a set of assembly steps in a factory to produce a product that will leave the factory and be sold to a customer.

Applying value streams to product development, we change the start point to the initial request for a new feature or the initial idea for a new product. We then outline the major steps and who does them. The endpoint for the value stream is when that new feature or product is released to the customer. A representative example is shown here:

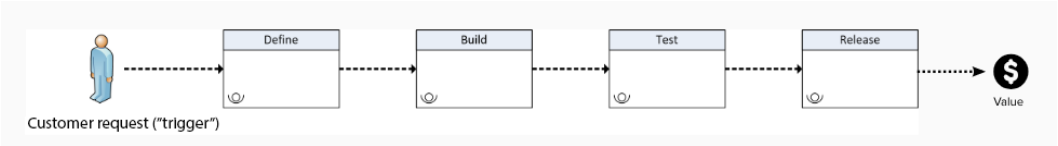


Figure 2.5 – Value stream

Value streams are a key part of the Lean methodology. James P. Womack and Daniel Jones identified a five-step process in their book, *Lean Thinking*, incorporating value streams. This process is summarized here:

1. Work with the customer to identify value.
2. Identify the value stream that details the activity from request to delivery.
3. Ensure that flow occurs on the value stream created.
4. When regular flow occurs, ensure customers can pull desired changes through the value stream.
5. Continuously improve and optimize the value stream.

Operational and development value streams

The classic value streams introduced in the previous section certainly provide a model for a development process from initial concept to release to the customer. In SAFe, such a value stream is defined as a development value stream.

SAFe also looks at the ways customers use an organization's products and services to achieve value. This view from the customer vantage point is defined as an operational value stream. The products and services touched upon in the operational value stream are developed and maintained by development value streams.

Let's look at how operational value streams and development value streams work together to give the customer value with an example incorporating a fictional media streaming service.

The new customer wants to view programs on the streaming service. They will go onto the web portal (developed by the Portal ART) to set up an account, including any necessary billing. They may want to set up the streaming service on their smart TV using the mobile interface (developed by the Mobile ART). Finally, they may want to watch original programming exclusive to the streaming service (developed by the Content ART).

This customer scenario is shown here as an operational value stream with intersecting development value streams:

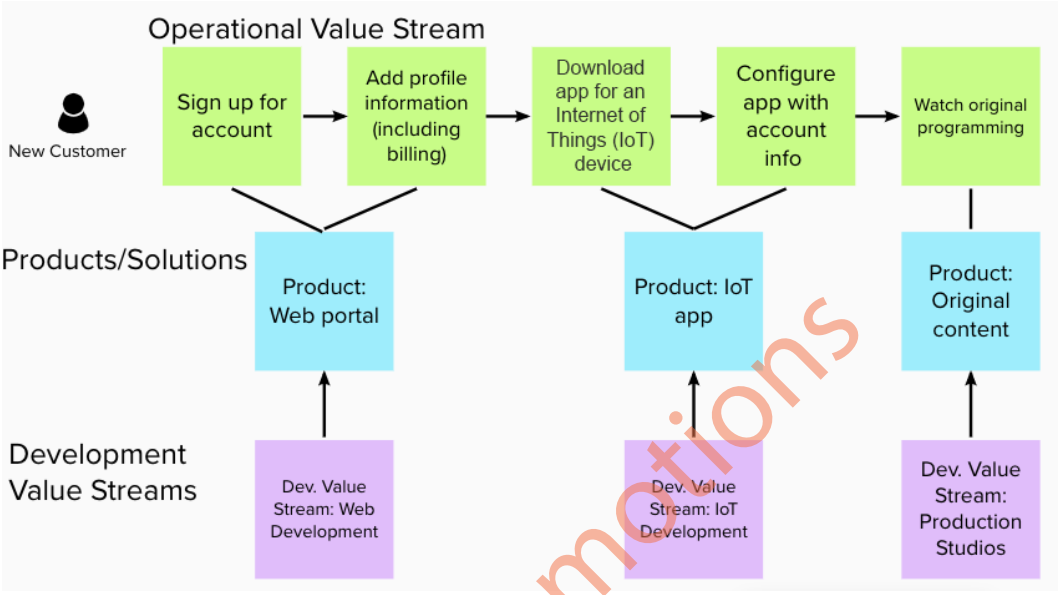


Figure 2.6 – Organizational value stream with development value streams

Value streams represent a key Lean practice. Adopting value streams and organizing work around value streams allows for an ART to create behaviors that affirm the Lean-Agile mindset, the core values, and the SAFe principles. As these behaviors settle in, the culture moves toward becoming generative, aligning everyone to the mission.

Summary

In this chapter, we started our examination of the **Culture, Automation, Lean Flow, Measurement, and Recovery** (**CALMR**) approach by looking at the first and most important factor: culture. We examined the work of Ron Westrum, which talked about three types of organizational cultures: pathological, bureaucratic, and generative. After looking at the characteristics of each of these cultures, we found that the one that really empowered our teams was a generative culture.

Once we settled on the ideal organizational culture, we then looked at how to move to that ideal culture. To help on this journey, we looked at the eight steps of the Change model created by John Kotter. We also saw these steps in action with examples from Ford Motor Company under Alan Mulally.

We identified the mindset that drives the behaviors we want to exhibit in our generative culture. The sources of this mindset come from both bodies of knowledge regarding Lean thinking the Agile Manifesto, and the SAFe core values. We also identified those principles from the Agile Manifesto that may need special attention when working with a team of teams or a scaled environment.

Also driving our behaviors and providing additional perspective are the 10 SAFe principles. We saw how these principles can act as guidance between core values, mindset, and practices.

Finally, because culture is built upon both structure and behavior, we took a close look at an ideal structure to create a generative culture. We further saw how value streams started out in Lean, and how SAFe evolves the classic definition of value streams to operational and development value streams.

In this chapter, we saw the importance of culture and ways to attain that culture through behavioral changes. Technology can help enable that change to a generative culture. In the next chapter, we'll examine the technology that forms the automation aspect of CALMR, which performs that enablement.

Questions

Test your knowledge of the concepts in this chapter by answering these questions.

1. In a generative culture, messengers are:

   A. blamed.
   B. trained.
   C. ignored.
   D. celebrated.

2. What serves as the *roof* in the SAFe House of Lean?

   A. Time
   B. Value
   C. Leadership
   D. Innovation

3. For teams on an ART to achieve alignment, they need to practice both cadence and _____.

   A. flow
   B. culture
   C. synchronization
   D. independence

4. Identify two SAFe core values.

   A. Built-in quality
   B. Adaptation
   C. Empowerment
   D. Flow
   E. Transparency

5. Value streams identify the _____, the people involved, and the time it takes to deliver something of value to the customer from the initial idea.

   A. tools
   B. culture
   C. steps
   D. products

6. In the Kotter model for change, what comes after *generating short-term wins*?

   A. Celebrating and acknowledging the wins
   B. Empowering others to enact the vision
   C. Consolidating wins to drive more change
   D. Connecting the wins to the vision

Further reading

For more information, refer to the following resources:

- `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1765804/`—"*A typology of organisational cultures*" article by Ron Westrum, identifying the three types of organizational cultures, their characteristics, and their effects.
- *Accelerate: Building and Scaling High Performing Technology Organizations* by *Nicole Forsgren PhD*, *Jez Humble*, and *Gene Kim*—This is a research-based approach to the effectiveness of DevOps principles and practices.
- *Leading Change* by *John Kotter*—This is the book that describes the eight-step model to change an organizational culture.
- *American Icon: Alan Mulally and the Fight to Save Ford Motor Company* by *Bryce G. Hoffman*—A behind-the-scenes look at the changes Alan Mulally and others set in motion that saved Ford Motor Company.

- `https://www.scaledagileframework.com/lean-agile-mindset/`—An introduction to the Lean-Agile mindset in SAFe, featuring the SAFe House of Lean and the Agile Manifesto.
- *The Principles of Product Development Flow: Second Generation Lean Product Development* by *Donald Reinertsen*—This book is the basis for the 10 SAFe principles. An interesting read for introducing flow.
- *Drive: The Surprising Truth of What Motivates Us* by *Daniel Pink*—A glimpse into what motivates knowledge workers.
- *Accelerate (XLR8): Building Strategic Agility for a Faster-Moving World* by *John Kotter*—Another masterwork by Kotter on changing organizations by setting up the "Dual Operating System."
- *Lean Thinking* by *James P. Womack and Daniel Jones*—An examination of Lean principles using value streams. The majority of this book is based upon an analysis of the TPS.

3

Automation for Efficiency and Quality

Of the factors in the CALMR (Culture, Automation, Lean Flow, Measuring, Recovery) approach, automation is the one most associated with the DevOps approach. A great deal of energy is devoted by DevOps practitioners to keeping current on trends in technology for environments and tooling. These tools, with different functions, are tied together to form a toolchain or pipeline.

We start our look at different types of tools in our pipeline by looking at the foundational tool types every pipeline needs. This includes Agile project management, version control systems, and review/documentation tools.

Continuous Integration (**CI**) tools stem from build management utilities. We will examine tools that create builds and other types of tools that run when a build is executed. These include automated testing tools, packaging tools, and artifact repositories.

An extension of CI is the deployment of build packages to staging and production environments. We will examine the tool types used in **Continuous Deployment** (**CD**), including configuration management, **Infrastructure as Code** (**IaC**), and vulnerability scanning tools.

Automation still relies on people. We will have a look at the ways development teams and operations teams can align to create the necessary automation and environments using DevOps topologies.

Finally, we'll see how people create the automation for the Continuous Delivery Pipeline in SAFe® by examining what the system team does in the **Agile Release Train** (**ART**).

In a nutshell, the following topics will be covered in this chapter:

- Pipelines and toolchains
- Continuous integration
- Continuous deployment
- DevOps topologies
- The system team

Pipelines and toolchains

A toolchain is the set of tools used in DevOps practices in the product development life cycle. The classic representation of the toolchain used in DevOps is an infinity loop, broken up into a number of functions. Each function or stage is enhanced by automation. A representation of this infinity loop, created by Kharnagy, and licensed under the Creative Commons Attribution ShareAlike license, is shown in the following figure:

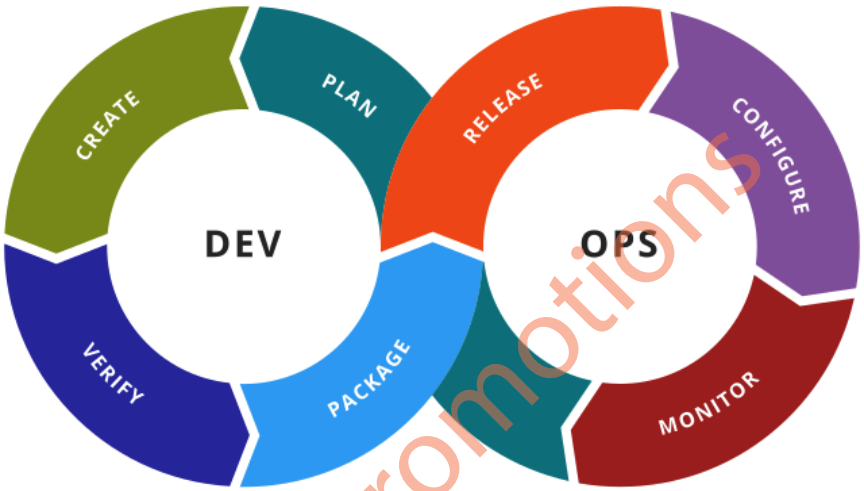


Figure 3.1 – DevOps toolchain

If we separate the ends of this infinity loop, we see the basis of our pipeline. The pipeline orchestrates the operation of all the stages with the exception of the monitoring stage. This begins our look at each pipeline stage, as shown in the following figure:

Figure 3.2 – DevOps pipeline

We begin our examination of the pipeline by looking at the activities whose artifacts set the pipeline in motion: plan and create. These foundational steps are illustrated in *Figure 3.3*:

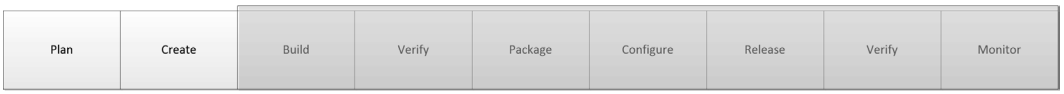


Figure 3.3 – Pipeline foundations

Let's start our examination of the CI/CD pipeline by looking at foundational tools. We will look at tools that can assist with planning in our value stream and monitoring the progress of the overall development process. We'll also examine the tools that act as repositories for the code, tests, configuration scripts, and documentation.

Planning with Agile project management tools

To look at where we are from request to release, we need to find a way to understand what we must do, and what the progress is of those steps. There are a large variety of methods to achieve this, from physical Kanban boards to Excel spreadsheets. As teams deal with remote and geographically distributed ways of working, Agile project management tools are a popular method for doing this.

Agile project management tools allow for the creation and update of work items. Progress on the work items is displayed on either a Kanban board or a list of issues. Recording the work items and their progress allows for easy collection of progress metrics, such as lead time.

In addition, work items can be linked to branches in version control and executions in a CI/CD pipeline tool. This allows for a trail of when a change was released throughout the entire pipeline.

Leading Agile project management tools include Jira and Trello, both by Atlassian, Azure DevOps by Microsoft, Digital.ai Agility (formerly known as VersionOne), IBM Engineering Work Management (formerly known as IBM Team Concert), and Broadcom Rally. In addition, many version control tools such as GitHub and GitLab include Agile project management functions.

Creating code and documentation

Version control has been an important part of software development since the 1990s. With version control, multiple developers can work on the same code base without fear of deleting each other's changes. To accomplish this, developers create a branch that contains their changes. When the time comes to share these changes, they merge the changes back into a shared branch where any differences are resolved. Merges can also be effective points for other developers to review any code changes going to the shared branch.

These days, code is not the only artifact kept in version control. Testing scripts used for automated testing can be kept in version control. Text files that are used for configuring staging and production environments are also kept in version control. In short, anything that is text that refers to any change or release is kept in version control. As we saw in *Chapter 1*, *Introducing SAFe® and DevOps*, when looking at Flickr, a common version control system between Dev and Ops is best.

The most prevalent version control system for code is Git, invented by Linus Torvalds, which was used as the repository for the Linux operating system. Git is a distributed version control system that allows copies of the entire repository to be easily replicated, even to developers. Even with the ease of replication, there are Git hosting solutions available that allow organizations to centralize the repositories to an *origin*. The most popular Git hosting products include Bitbucket by Atlassian, GitHub, GitLab, and Azure DevOps.

Documentation is another important artifact created for product development. **Non-Functional Requirements** (**NFRs**) may be detailed in specifications, architecture may be specified in terms of models and diagrams, and **user interface/user experience** (**UI/UX**) guidelines may be depicted as wireframes and sketches. These initial designs may start from planning and continue in the iterative learning cycles.

Document repositories and wiki software are used to store requirements specifications, architectural models, UI wireframes, and product and user documentation. Popular repositories include Confluence from Atlassian and GitLab Pages.

Once changes have been added to the repository in version control, the work of the CI/CD pipeline can begin. Let's take a look at the activities that make up continuous integration.

Continuous integration

When code changes are ready, automation can begin building the necessary packages for use in staging and production environments. As part of the build process, tests can be run to determine the correct function as well as security. When testing indicates correct and secure functionality, a package is created and stored in artifact repositories based on the technology used.

This part of the pipeline is illustrated in the following figure:

Figure 3.4 – Pipeline: Continuous integration

Let's look at how the CI portion of the pipeline manages a build, executes the initial-level testing, and packages the build. We will begin with a definition of continuous integration, continuous delivery, and continuous deployment.

Continuous integration versus continuous delivery versus continuous deployment

We will see that continuous integration captures the activities that can be automatically run once a change has been committed to the version control system. Code, including any changes, can be compiled or packaged to a form computers can use. Tests are run after the build step to ensure no bugs or security vulnerabilities have been introduced. Notifications can be created upon success or failure. Upon success, the code changes can be merged with the existing code base. We will examine these steps in detail when we look at the continuous integration stage of the Continuous Delivery Pipeline in *Chapter 11*, *Continuous Integration of Solution Development*.

Continuous delivery takes the continuous integration steps further by allowing the newly-merged changes to be packaged and delivered to a staging environment, a test environment that is as similar to production as possible, or to production. Once delivered to the environment, further tests can be run to verify the correctness of new features or to perform a deeper security scan. The success of these tests allows the organization to release the changes when they're ready. Detailed steps outlining continuous delivery (labeled as continuous deployment) will be listed in *Chapter 12*, *Continuous Deployment to Production*.

Continuous deployment is continuous delivery with one further step: when the tests are complete in the production environment, the new features are automatically released to allow customers to use them immediately.

Regardless of whether your final stop in automation is continuous integration, continuous delivery, or you completely automate a release through continuous deployment, you will typically use the same tool to establish your pipeline. Let's look at that category of tools now.

Orchestrating the change

Pipeline orchestration tools (commonly referred to as CI/CD tools) begin as build management tools. These tools execute build scripts and perform additional actions when triggered manually or automatically when a commit occurs in the version control system.

Earlier versions of CI/CD tools maintained the jobs to be done as part of the UI. CI/CD tools today allow jobs to be defined through a text file using YAML or other formats.

The power of the CI/CD tools lies in their flexibility. Easy integration with other tools to perform other functions, such as automated testing and deployment, has enabled overall success in the DevOps movement. Scalability in execution through the incorporation of agent software in worker nodes is another important factor, allowing jobs to be created in any environment.

CI/CD tools can be set up in *on-premises* environments, on private clouds, or as **Software-as-a-Service** (**SaaS**) products. The most popular CI/CD tool for on-premises or private cloud environments continues to be Jenkins, an open source project that started as Hudson. Other popular tools include CircleCI and Bamboo from Atlassian. Many Git hosting products have rolled out CI/CD pipeline extensions as part of their system, including GitLab, GitHub Actions on GitHub, Azure DevOps, and Bitbucket Pipelines on Bitbucket Cloud.

Verifying quality

By far, one of the most important functions a pipeline can do is set up and execute automated testing. Automated testing is gaining attention due to the *shift-left* philosophy with the realization that the earlier and more often you do testing, the better the quality of the finished product. The DevSecOps movement advocates for earlier and more frequent automated testing as a way of establishing *continuous security*. Early testing can be done without requiring the code to be executed in an environment either through simulated inputs and evaluating outputs or through an examination of the code.

These *first-level* tests are known as unit tests and static analyses. Let's take a detailed look at them now.

Unit tests (test-driven development)

Unit tests are scripts written to verify that functions in code produce the desired output when given simulated input. Unit test frameworks such as JUnit and NUnit are specific to the language used to create the code. Unit tests can run directly from the pipeline as a defined stage.

Test management software can also be used to execute unit tests. Each unit test is saved as a test case in the test management software, and the results are recorded. Test management software can also set up an integration to Agile project management tools to record defects when tests fail.

Popular test management software includes Engineering Test Management from IBM, XRay from XBlend, and Zephyr from SmartBear.

Static analysis

Static analysis is the examination of code without executing it. Typically, tools are used to analyze and audit the code. Static analysis has other names depending on the expected outputs:

- Linting is a static analysis done with a specific tool (lint). Linting examines code looking for possible code errors and can be used to enforce coding standards.
- **Static application security testing** (**SAST**) is static analysis applied to searching for possible security vulnerabilities in code.
- Dependency scanning looks at the dependencies of libraries called by code to review whether known security vulnerabilities exist.
- License scanning looks at the dependencies called by code to review the type of open source licensing the libraries use. This helps keep the organization compliant with the types of open source licenses used and if attribution and distribution of changes are required.

Tooling that can perform the analysis described, including SAST, includes SonarQube from SonarSource, Snyk, Coverity from Synopsys, mend.io (formerly WhiteSource), Klocwork offered by Perforce, as well as GitLab.

Packaging for deployment

After the first level of the tests pass, the pipeline can then prepare the code changes. Packaging the changes is dependent upon several factors, including the language and the technology used for deployment.

Artifact repository tools allow for version control of large package images. These may pose problems with storage on the version control software mentioned previously because these artifacts are large binary files. These binary images may range from standard packages such as WAR files in Java or NPM images in Node.js, to **virtual machine** (**VM**) images. The popularity of Docker as a deployment

technology has created a need to identify and version control Docker images in private repositories, resulting in additional capabilities for artifact repository tools.

Popular artifact repository tools include Artifactory by JFrog and Nexus by Sonatype. In addition, GitLab and Azure DevOps include the ability to act as an artifact repository for binary images.

Continuous deployment

During the continuous integration phase of the pipeline, we saw the last step as the packaging of changes into a binary image. Continuous deployment continues from that step to the application of that image into testing and production environments.

Automation may play a role in adding or updating resources in these environments. IaC tools allow the configuration of these resources.

Now that code changes are in an environment, testing can be done in further detail to find problems with quality and security. Here, the tests may also look at how changes affect the performance and validation of the desired changes.

As changes are added to environments, we need to be aware of the effects of these changes. To that end, we will measure the performance of the overall environment including the storage and analysis of logs.

The continuous deployment stage is illustrated in the following figure:

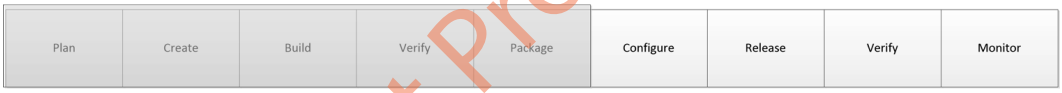


Figure 3.5 – Pipeline: Continuous deployment

Let's look at these activities carried out in the environments. We may need to configure the environment to set up new features. Then comes the actual deployment of changes into the environment. Finally, more and deeper testing can be performed in the environment to ensure the correct function, security, and value.

Configuring environments with IaC

Often, changes may involve creating new resources in an environment. Part of the configurations in configuration management tools may invoke other tools that allow the automatic creation of resources. The creation of these resources is guided by a script, often in YAML format. Due to the reliance on these scripts, the tools are described as IaC.

The emergence of public cloud environments, such as **Amazon Web Services** (**AWS**), Azure from Microsoft, and Google Cloud Platform, has introduced tools associated with each cloud environment. The most notable of these is CloudFormation, which works with AWS.

Other vendors offer IaC tools that are more flexible, working in a variety of physical servers, private cloud, and public cloud environments. The most notable of these is Terraform by Hashicorp.

Releasing with configuration management and feature flags

Configuration management tools are responsible for identifying and setting the configuration of development and production environments. A pipeline can invoke the configuration management tool to introduce a build package that has passed the continuous integration stage.

Originally, configuration management tools specified the configuration for physical (bare metal) servers or VM images. They have grown to include Docker containers and Kubernetes clusters.

Descriptions of configurations are often specified in terms of the desired configuration state but do not elaborate on the necessary steps to achieve the desired state. This helps to achieve idempotence in the system.

Popular configuration management tools include Chef by Progress Chef, Puppet, and Ansible by Red Hat. Ansible has an advantage over both Chef and Puppet in that it connects to the environment resources through **Secure Shell** (**SSH**), which removes the need to install agent software on the resource.

Release visibility with feature flags

Even as code changes make their way into production environments, those changes may not be visible to the end users or affect existing functionality. This may be due to code switches or *feature flags* that prevent the visibility of the code changes. This allows for a gradual rollout of changes, such as canary deployments. This also allows for a quick reversion to the previous state by deactivating the applicable feature flags.

Popular feature flag tools include LaunchDarkly, Flagsmith, and CloudBees Feature Management.

Additional verification through advanced testing in the environment

Now that changes are built, packaged, and placed in environments, testing can work on deeper levels. Test inputs can be placed in the environment to determine whether the code works as expected, whether any vulnerabilities are introduced, and whether the system behaves as expected.

These types of tests that measure correct functionality, security, and acceptance are described as follows.

Functional and UI testing

Functional testing is most concerned with code correctness. It exists primarily to see whether the coding works and meets the base requirements. Typically, functional tests go beyond the individual code functions, which would have been tested during unit testing. Specific types of functional testing are used in the following scenarios:

- **Sanity testing** is running a small set of functional tests to verify code features
- **Smoke testing** usually involves running short, high-level functional tests to gain confidence in a new build or a new deployment
- **Regression testing** is a more extensive execution of functional tests to verify that new code features work with the existing system functionality

Automated tools for functional testing depend upon the language the coding features are written in, the environment the code will be deployed in, and the technology platform (web versus mobile versus other). A cross-section of popular tools includes UFT by Micro Focus, Worksoft Certify, and Tricentis Tosca.

UI testing is functional testing for graphical UIs. This ensures that elements such as buttons and fields on a web page connect to the correct underlying code functions and ensures the correctness of those code functions.

Many popular UI testing tools are based on Selenium, a platform that captures actions performed on a web page in scripts that can be repeated by automation. Such tools include TestComplete and CrossBrowserTesting by SmartBear and Sauce Labs.

Load/performance testing

Performance testing, such as load testing, is not designed to measure correct functionality. Rather, the goal of performance testing is to verify any NFRs such as reliability and scalability. We want to see how the system, including any new code changes, can handle increased demand for its resources by flooding the system with a large number of system requests, such as logins and form evaluations.

Popular tools for performance testing include LoadRunner from Micro Focus and JMeter for traditional applications and Sauce Performance from Sauce Labs for web and mobile application performance testing.

Dynamic application security testing

Dynamic application security testing (**DAST**) continues the emphasis on security in DevSecOps. With DAST, automated tests continue security scanning by performing simulated attacks on the environment for web applications to find vulnerabilities.

A leading DAST scanner is OWASP Zed Attack Proxy, which is used by GitLab to provide DAST scanning functionality on its pipeline.

IaC scanning

Additional tests for DevSecOps continue with the ability to scan the IaC files to discover whether there are any misconfigurations or security vulnerabilities.

Leading tools such as Snyk and GitLab can scan for multiple IaC tools, including Ansible, Terraform, Dockerfiles, and configuration services for public clouds, such as CloudFormation, Google Deployment Manager, and **Azure Resource Manager** (**ARM**).

Container scanning

Containers are a technology where an application and any needed libraries are encapsulated as a virtual image. This virtual image can be an extension of base resources that represent the functions provided by an operating system.

Docker is the technology used to implement containers. Developers define the application and libraries in Docker images. The image can be placed in a repository where it can be pulled and executed in any environment by Docker Engine.

Container scanning allows the Docker image and dependent images to be scanned to look for security vulnerabilities. Tools that can implement container scanning include GitLab and Snyk.

Acceptance tests (behavior-driven development)

Acceptance tests are test scripts written in a business-readable language called Gherkin. Each test is composed of three main clauses, each starting with a keyword:

- **Given**: This clause describes the initial conditions
- **When**: This clause describes the input for the test
- **Then**: This clause describes the desired behavior

Cucumber is the tool that executes Gherkin tests. Cucumber is available in an open source version and paid versions are available in CucumberStudio and Cucumber for Jira. All versions are supported by SmartBear.

Monitoring the environment

We now leave tools that are part of the pipeline to tools that are run continuously. Ongoing evaluation of staging and production environments is done by tools independent of the pipeline. These tools perform the following functions:

Performance monitoring/reporting

Stability is the key goal for operations. To that end, they will monitor the health of the environment by collecting metrics that can indicate the health of key components. This may include the following metrics:

- CPU utilization
- Memory utilization
- Storage utilization

- The number of processes
- Network statistics
- Application state

Popular tools for monitoring include Prometheus for collecting metrics and Grafana for displaying the metrics on a dashboard. If the environment is on a public cloud, CloudWatch is available on AWS, and Azure Monitor is available on Azure. Cloud-based **monitoring-as-a-service** (**MaaS**) products can consolidate monitoring from multiple environments and sources. Such products include Datadog and New Relic.

Log collection

Another aspect of monitoring comes from collecting log messages created by the system and applications. The messages may provide context for issues when problems arise in the environment.

Logs from different systems, different system components, and different applications are collected into one source using log aggregation tools. These tools include a search capability to filter by an important keyword when necessary.

Log aggregation tools can be a software application that resides on-premises or in a private cloud, a feature available on public clouds, or a **SaaS** product. Popular log aggregation tools include the combination of Elasticsearch, Logstash, and Kibana (an ELK stack) for collection and analytics in on-premises/private cloud environments. Log collection is part of the AWS CloudWatch service. Splunk and Datadog are popular SaaS-based products that perform log aggregation.

Alerting

When problems arise, it is important to notify the key people in a timely fashion. Alert tools can provide multiple channels for notification, including emails, SMS messages, and IM chat messages. They may also provide a tolerance mechanism to prevent too many alert messages to operations personnel and *alert fatigue* from occurring. These tools can also create issues for incident management so that **IT service management** (**ITSM**) processes are followed.

Leading alerting tools include PagerDuty and Opsgenie by Atlassian.

At this point, we've talked about the technology involved in creating the automation that is part of DevOps. Let's focus our attention now on people, in terms of who can be responsible for installing and configuring such automation as the CI/CD pipeline.

DevOps topologies

With the growing list of tools and technologies available to Dev and Ops, it may be difficult to figure out where the responsibilities lie in moving toward a DevOps approach. Who is responsible for creating the CI/CD pipeline? What do we consider databases? How do we deploy into production?

In 2013, Matthew Skelton initially described three team *anti-types* to avoid and five possible team structures. Additional contributions have increased the number of anti-types to eight and the number of beneficial team structures to nine. The following list shows the anti-types and they are elaborated here at `https://web.devopstopologies.com`:

- Dev and Ops Silos
- Permanent DevOps Team Silo
- Dev Doesn't Need Ops
- DevOps as the Dev Tools Team
- Rebranded Sysadmins
- Ops Embedded in Dev Team
- Dev and DBA Silos
- Fake SRE

The 9 DevOps topologies from that site are as follows.

Dev and Ops collaboration

This structure is considered the ideal DevOps approach, where Dev and Ops are working together and have smooth collaboration. Implementing this structure likely requires a large organizational culture change toward a generative culture.

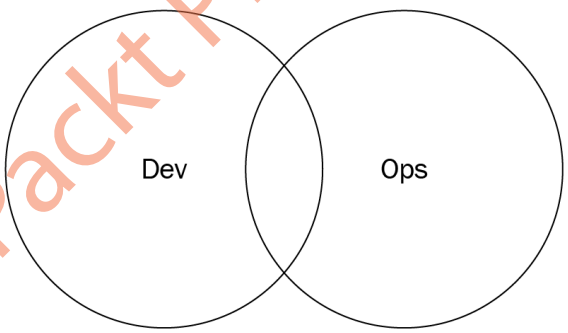


Figure 3.6 – Dev and Ops collaboration (diagram based on work at devopstopologies.com – licensed under CC BY-SA)

Fully shared Ops responsibilities

Some organizations with a single web-based product, such as Netflix or Facebook, may be able to take the Dev and Ops collaboration model shown previously and integrate Ops more fully. In this model, there is very little separation between Dev and Ops. Because of this, everyone is focused on the mission.

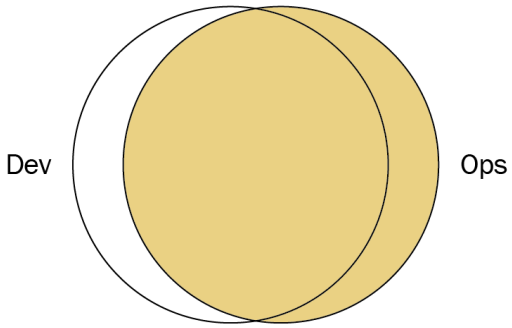


Figure 3.7 – Fully shared Ops responsibilities (Diagram based on work at devopstopologies.com – licensed under CC BY-SA)

Ops as infrastructure as a service

There may be some organizations that have a more traditional Ops department. Also, some organizations may deploy applications to public cloud environments such as AWS or Azure. In either case, a small subset of the Dev department may treat operations *as a service* and set up tooling for deployment, metrics, provisioning, and monitoring of those resources. In this model, there is no direct collaboration with Operations.

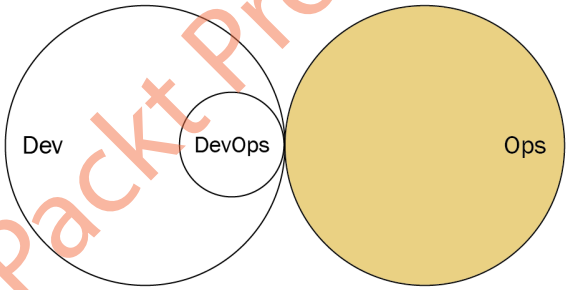


Figure 3.8 – Ops as infrastructure as a service (diagram based on work at devopstopologies.com – licensed under CC BY-SA)

DevOps as an external service

Some smaller teams and organizations may not have the manpower or experience to move toward a DevOps approach. In that case, they may contract an external vendor to create the test environments and automation and configure the monitoring. The DevOps vendors may also train Dev and Ops to move to a different model, such as Dev and Ops collaboration.

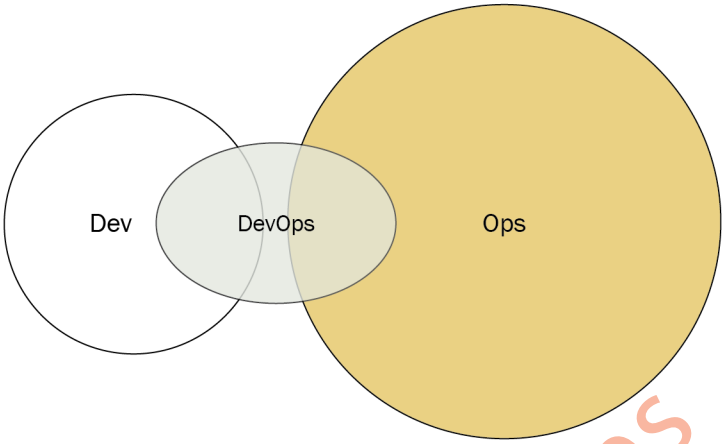


Figure 3.9 – DevOps as an external service (Diagram based on work at devopstopologies.com – licensed under CC BY-SA)

DevOps team (with expiration date)

There may be situations where having a dedicated DevOps team works. The idea is that the DevOps team can act as a *bridge* for both Dev and Ops teams. The DevOps team can instruct developers on working with infrastructure and can instruct operations personnel on Agile development. At some point, the DevOps team will disband, allowing Dev and Ops to collaborate in the Dev and Ops collaboration model.

The danger exists when the DevOps team does not disband, instead forming a separate silo. This is actually one of the identified anti-types (DevOps Team Silo) mentioned on the DevOps topologies website.

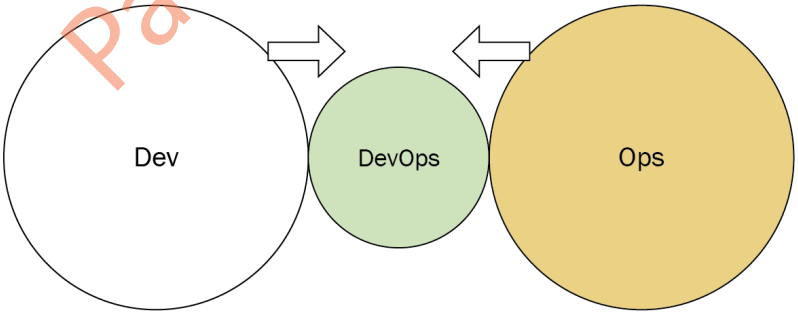


Figure 3.10 – DevOps team with expiration date (diagram based on work at devopstopologies.com – licensed under CC BY-SA)

DevOps advocacy team

A DevOps advocacy team acts as a facilitator between Dev and Ops if the two departments tend to drift apart. Unlike the DevOps team with an expiration date, this DevOps team is kept on an ongoing basis, ensuring both Dev and Ops follow current DevOps practices.

Like the DevOps team with an expiration date, a DevOps advocacy team runs the risk of turning into a DevOps Team Silo.

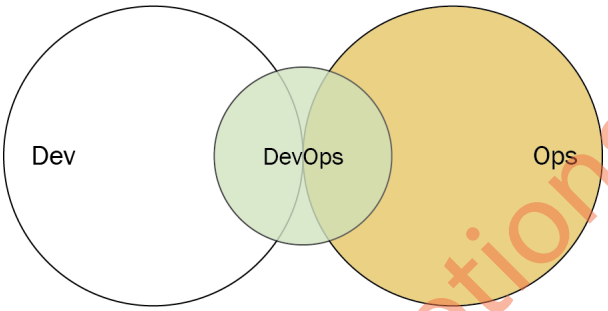


Figure 3.11 – DevOps advocacy team (Diagram based on work at devopstopologies.com – licensed under CC BY-SA)

SRE team

As far back as 2004, Google has used its software engineers as operations personnel. These **site reliability engineers** (**SREs**) handle the support of production environments, mostly by developing software to keep the resources and services running. SREs accept the application from Dev, but only if Dev provides enough evidence in the form of logs and metrics that it meets a quality threshold. If the code does not meet this standard, SREs can reject the deployment.

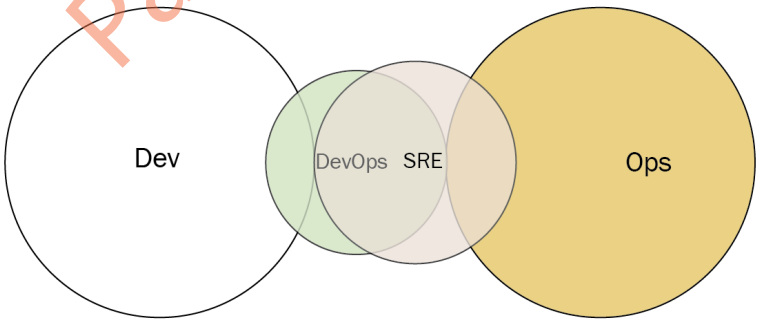


Figure 3.12– SRE team (diagram based on work at devopstopologies.com – licensed under CC BY-SA)

Container-driven collaboration

Because containers abstract many of the infrastructure details, most collaboration between Dev and Ops is not necessary. In this case, a container-based deployment may be accepted by Ops most of the time if there is a sound engineering culture. If not monitored closely, there is a risk of changing to an anti-type where Ops is expected to deploy anything from Dev without question.

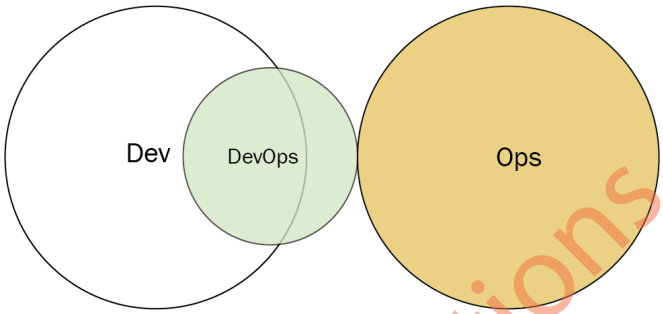


Figure 3.13 – Container-driven collaboration (diagram based on work at devopstopologies.com – licensed under CC BY-SA)

Dev and DBA collaboration

If the applications an organization develops rely on one or more central databases, the collaboration between developers and the **database administrators** (**DBAs**) may be crucial. To enable that collaboration, the database developers in Dev work closely with the DBAs in operations.

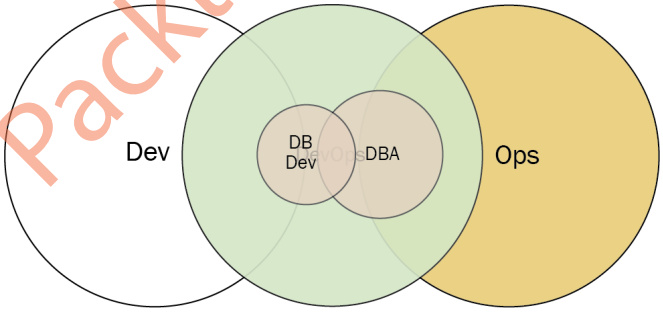


Figure 3.14 – Dev and DBA collaboration (Diagram based on work at devopstopologies.com – licensed under CC BY-SA)

Now that we have seen possible configurations for organizing the team responsible for the CI/CD pipeline, let's take a close look at such a team on the ART: the system team.

The system team

The system team is the team on the ART that is responsible for the tooling and automation of the Continuous Delivery Pipeline. They work with the other teams on the ART to help in delivering valuable solutions.

The system team may follow one of several DevOps topologies. The system team may be set up as a DevOps team with an expiration date. They will set up the Continuous Delivery Pipeline and instruct the Dev and Ops personnel on its use before disbanding. Another model for the system team may be being set up as a DevOps advocacy team.

As custodians of the automation and development process, they have deep responsibilities to the other teams on the ART. These responsibilities are described as follows.

Building infrastructure for solution development

The system team will often be responsible for setting up the pre-build, continuous integration, and continuous deployment portions of the CI/CD pipeline and integrating the technology so it's a seamless part of the Continuous Delivery Pipeline. They strive to apply automation as much as possible. This may also involve close collaboration with other teams, so they may visit other teams' events.

Spearheading solution integration

As part of maintaining the CI phase, the system team may be involved in determining the build process after a change has been committed to version control. They will maintain the proper build scripts and CI configuration files. If build automation is not yet available, they may be the team performing build and integration activities.

Setting up end-to-end testing

To support the other teams, the system team may help the testers with the creation and optimization of automated tests. They may also work with the other teams to aggregate separate tests into well-defined test suites for different types of testing, such as smoke testing.

Assisting with demos

The ART integrates the work from all its teams and demonstrates the working state of the solution at a given point in time. This integration and demonstration is called the *system demo* and happens at a regular cadence.

As maintainers of the Continuous Delivery Pipeline, the system team is there to ensure that technical environments work for all teams so that the system demo is seamless.

Facilitating the release

Because the system team has a holistic view of the process, they may be called upon to verify that deployments to production and final release are valid.

The system team can be considered the *DevOps* team for the ART. It may follow one of the DevOps topologies as a way of collaborating with the other Agile teams. Its responsibilities primarily involve configuring the automation, but it may assist the Agile teams in other ways as the entire ART endeavors to deliver value.

Summary

Automation plays a key role in DevOps. We looked at the important tools that make up a DevOps toolchain, especially those parts of the toolchain that are orchestrated from building and testing to deployment, creating the CI/CD pipeline or *the pipeline*.

CI typically includes activities that happen to code changes after they have been committed to version control. This may include preliminary testing, and upon passing, they may be built together and packaged into an artifact based on language and technology.

CD continues from where CI leaves off by taking the build artifacts and applying them to testing or production environments. Here, environments will be reconfigured, possibly with new resources. Additional testing will be performed to ensure security, correctness, and validation of anticipated value.

DevOps topologies outline possible models of collaboration between Dev and Ops teams with the possible inclusion of people specializing in DevOps. Some of the topologies are not long-lasting, lest they turn into *anti-types* that stifle collaboration.

In SAFe, the system team performs as the DevOps team on the ART. That team is responsible for constructing and maintaining the Continuous Delivery Pipeline for the other teams on the ART.

Automation does allow the ART or any DevOps team to deliver faster, but not if the development process is not optimized for Lean flow. In the next chapter, we will examine the practices from the Lean thinking movement that enable flow.

Questions

Test your knowledge of the concepts in this chapter by answering these questions.

1. What tests are examples of static analysis (pick two)?

   A. Unit tests

   B. Linting

   C. DAST

D. Dependency scanning

E. Acceptance tests

2. What allows code changes to be hidden in production until *turned on*?

A. Version control

B. Continuous integration

C. Feature flags

D. Continuous deployment

3. Monitoring includes activities such as performance monitoring, alerting, and what?

A. Load testing

B. Version control

C. Log collection

D. Unit testing

Further reading

- The original formulation of DevOps topologies, including three anti-types and five types: `https://blog.matthewskelton.net/2013/10/22/what-team-structure-is-right-for-devops-to-flourish/`
- The updated formulation of DevOps topologies: `https://web.devopstopologies.com`
- *Team Topologies: Organizing Business and Technology Teams for Fast Flow* by Matthew Skelton and Manuel Pais – the evolution of DevOps topologies to look at topologies for all kinds of teams.

4

Leveraging Lean Flow to Keep the Work Moving

In *Chapter 1*, *Introducing SAFe® and DevOps*, we saw the inclusion of Lean thinking approaches such as Lean software development and Kanban into the Agile movement. Jez Humble saw it as important enough to include it in the CAMS model, creating the CALMS model from which we base the SAFe® CALMR model. Scaled Agile talks about that Lean-Agile mindset with an emphasis on both Lean thinking and an Agile mindset derived from the Agile Manifesto. How does this focus on Lean thinking manifest itself in DevOps and SAFe?

In this chapter, we will see that keeping product development moving at a predictable pace requires the establishment of a Lean flow. Proper flow allows automation to succeed. To that end, we will look at the following practices to establish Lean flow:

- Making sure all work and work progress is visible
- Limiting our **Work in Progress/Process** (**WIP**)
- Keeping the size of each batch of work appropriately small
- Monitoring our work queues
- Employing systems thinking to change the traditional definitions of project and teams

Making the work visible

Work could be defined as the effort the team or **Agile Release Train** (**ART**) (as a team of teams) may put forth to develop a product or solution. But not all that work may be focused on customer value.

The Phoenix Project: A Novel about IT, DevOps, and Helping your Business Win by Gene Kim, Kevin Behr, and George Spafford identifies four kinds of work. These are summarized as follows:

- **Business projects**: Requests for new features that will bring value to the customer

- **Internal projects**: Work that helps organizations continue to develop products efficiently
- **Maintenance**: Work needed to maintain existing products
- **Unplanned work**: Bugs, defects, and emergencies that occur from time to time

SAFe takes a few of these work categories and places them in **enablers**. The idea here is that enablers help create future business value. The four kinds of enablers defined by SAFe are listed here:

- **Infrastructure**: This enabler exists to enhance how products can be developed and delivered. Examples include new automated tests to include in the **continuous delivery** (**CD**) pipeline.
- **Architectural**: This enabler exists to enhance the architecture that business features and user stories rely on. SAFe defines the sequence of architectural enablers as an architectural runway that drives future business value. Examples include creating a new database server **virtual machine** (**VM**) for the staging and production environments.
- **Compliance**: This enabler describes additional work that may be needed in certain regulated industries. Examples include **Verification and Validation** (**V&V**), approvals, and documentation.
- **Exploration**: Sometimes, additional research is required to understand the optimal approach, learn about new technologies, or refine customer desires. Exploration enablers are created to identify the work that research requires. An example of this is a Spike, used by Agile teams to research new technologies or evaluate a development option, such as determining the correct technology for web streaming.

Given the different categories of work that is done, it's important to have a uniform way of displaying all the work of a team or ART. Traditionally, enterprising people, often with a technical background, have resorted to spreadsheets whose arrangement is flexible to allow for additional information such as work category or status. In addition, sharing spreadsheets among team members becomes difficult as there may be changes that need to be synchronized.

Dominica DeGrandis, in her book *Making Work Visible, Exposing Time Theft to Optimize Work & Flow*, notes that the majority of people are visual-spatial learners, in that they are able to understand and respond to information presented visually. This is one of the reasons for setting up a Kanban board.

A Kanban board is a space divided into columns. Each column represents a state in the workflow of an item of work. Items of work are represented by note cards and color-coded to represent the appropriate work category.

A simple Kanban board is illustrated in *Figure 4.1*. Note the three columns, representing work to be done (**Backlog**), WIP (**Doing**), and work completed (**Done**):

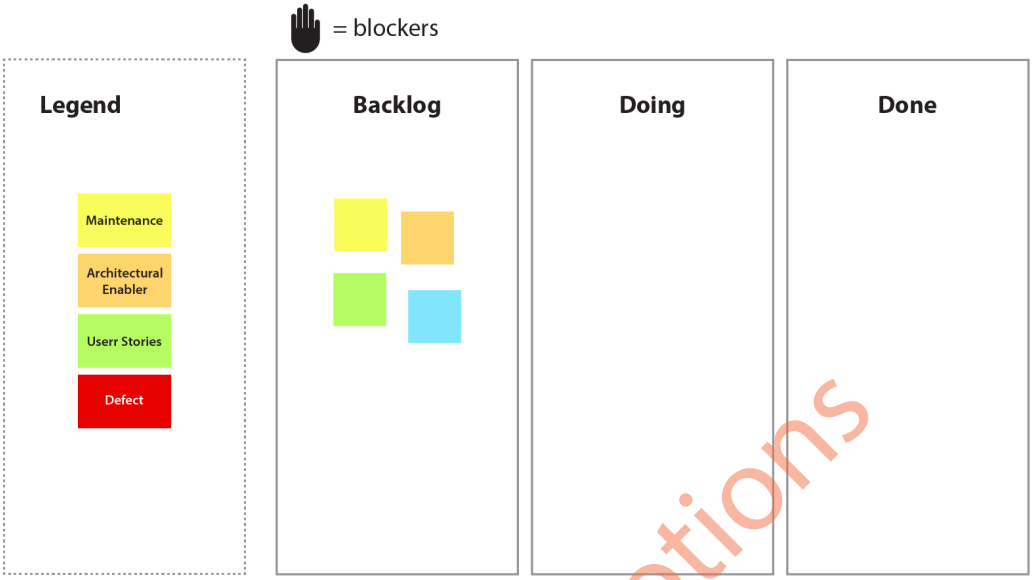


Figure 4.1 – Simple Kanban board

Additional features on a Kanban board can help teams manage the work they do. Let's take a look at those other features on a Kanban board.

Specifying workflow with additional columns

Often, having a single *in-progress* column doesn't provide visibility into everything that a team or ART is doing, especially if there are bottlenecks in the overall process. It makes sense to break up the *in-progress* column to highlight major steps in the development so that bottlenecks are easy to identify.

While having discrete process steps separated into separate WIP columns is generally done, it's important to remember that teams shouldn't be taking all issues and moving all of them from column to column as that devolves the process into Waterfall. The movement of the issues happens in a continuous fashion.

Figure 4.2 shows an illustration of separating our **Doing** column with stages for analysis, implementation, and review:

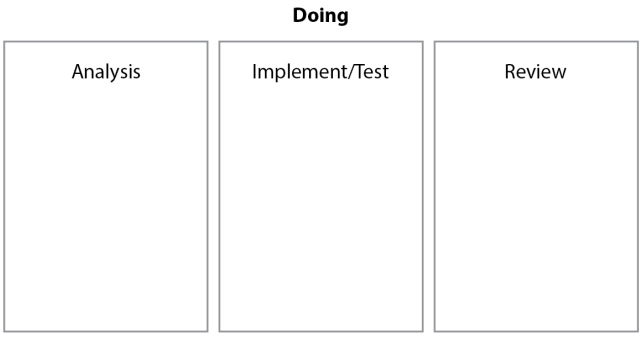


Figure 4.2 – Expansion of the Doing column into several stages

Flagging impediments and urgent issues

Sometimes a work issue may be blocked due to some external event or dependency. As these impediments or blockers come up, an icon can be attached to the work issue as an indicator to the teams to continue to pay attention to the issue until the blocker is removed.

In the same way, any urgent issue may need the attention of the entire team. The team may need to swarm on the urgent issue until it is resolved. Urgent issues may be identified by a special indicator put on the issue or by its position in an *expedite* swimlane that cuts horizontally across all columns of the Kanban board. In the following screenshot, an example of an urgent issue highlighted by a special indicator in the expedite lane is illustrated:

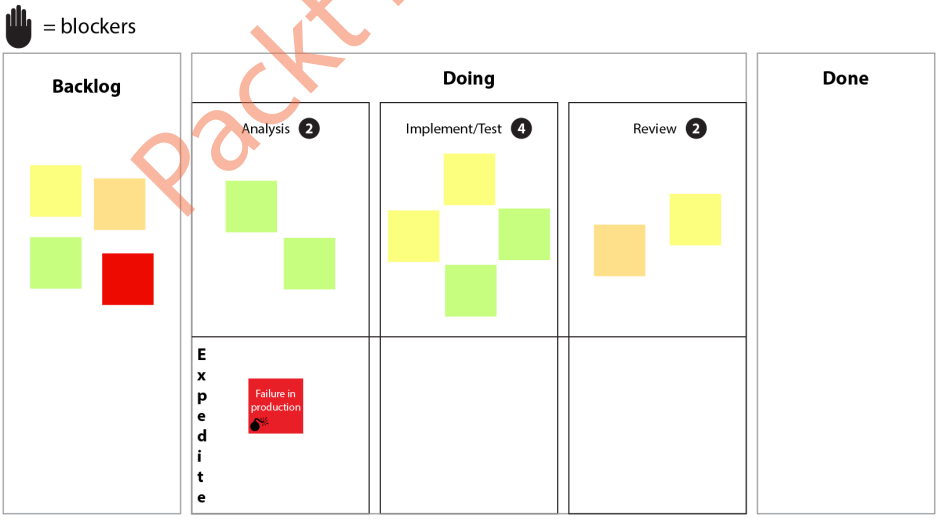


Figure 4.3 – Work issue with an urgent indicator in the expedite lane

Policies for specifying exit criteria

For every stage in the process, the team needs to have a clear agreement on the exit criteria. Having clear exit criteria avoids confusion as to whether an issue is truly finished at that stage. This agreement for each column is known as a policy.

Column policies have their place in a team charter or working agreement so that these are explicit and agreed upon. Examples of this include a **Definition of Ready** (**DoR**), where any questions are answered, and requirements are detailed enough that development can begin. Another example is a **Definition of Done** (**DoD**), which is an agreement by the team of the criteria that determine when a story is complete and development work on that story can stop. These policies avoid any confusion about whether a piece of work is truly complete as opposed to *done done*.

Other additions to the Kanban board will be identified as we look at other practices to ensure that Lean flow occurs. In the following section, we will examine the problem of having too much WIP and how a Kanban board can help by providing visibility and enforcing constraints on team behavior.

Limiting WIP

WIP is the work a team or ART has in process. It has been started but is not complete. If we were to view WIP on our Kanban board, it would resemble the following screenshot:

Figure 4.4 – Kanban board highlighting WIP

It is important to make sure that the work within these columns is monitored so that it doesn't overwhelm the teams or ART. According to Dominica DeGrandis, the effects of too much WIP may include the following:

- Too much multitasking, which will cause teams to spend too much time doing context switching and prevent them from finishing work
- New work items are started before existing work is finished
- Work takes a long time to finish (long lead times/cycle times)

A key way of ensuring that a team is not letting WIP go unchecked is by setting up WIP limits or constraints on each column between the **Backlog** column (where work has not been accepted yet) and the **Done** column (where completed work goes). An example of WIP limits on a Kanban board is shown in the following screenshot:

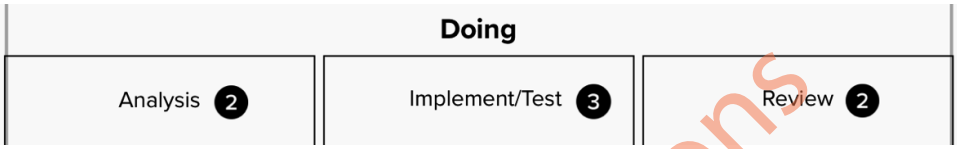


Figure 4.5 – WIP limits on columns

To understand how WIP limits can help a team achieve Lean flow, consider the following comparison of team behavior.

First, imagine a Kanban board with no WIP limits. A developer on the team has just finished the development of a user story. It meets the policy criteria for the column and could be pulled to the next column. If the developer moves that story to the next column and pulls a story to work, has that developer done anything to lower the amount of work that's in progress? This may be acceptable if there is flow already established, but if there is a bottleneck in the team's process, this action may exacerbate the bottleneck and prevent the entire team from actually delivering. This situation is captured on our Kanban board in the following screenshot:

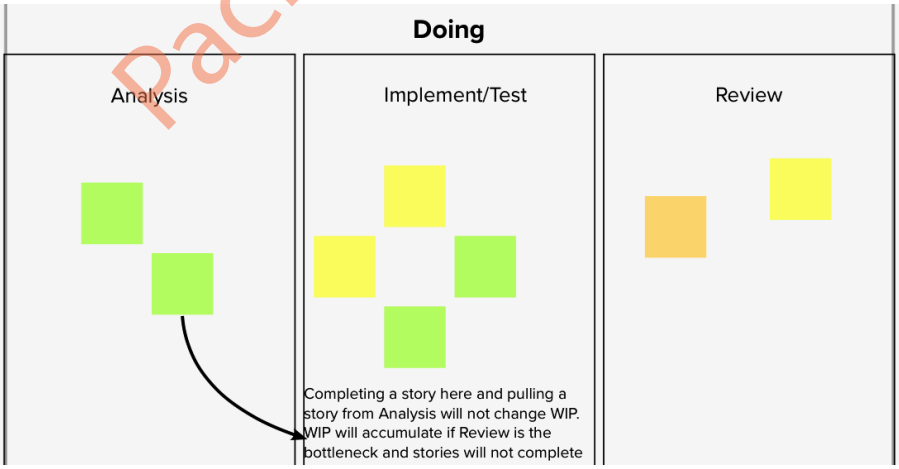


Figure 4.6 – Scenario with no WIP limits, resulting in no reduction in WIP

Now, let's imagine the same Kanban board with WIP limits in place. That developer who has finished the user story cannot move anything into the **Implement/Test** column. Also, no one can move anything from **Implement/Test** into **Review**. To help get the bottlenecks fixed, the developer acts to assist other team members to move a story from **Review** to **Done**. This has the effect of introducing throughput and establishing flow. We can view this scenario in the following screenshot:

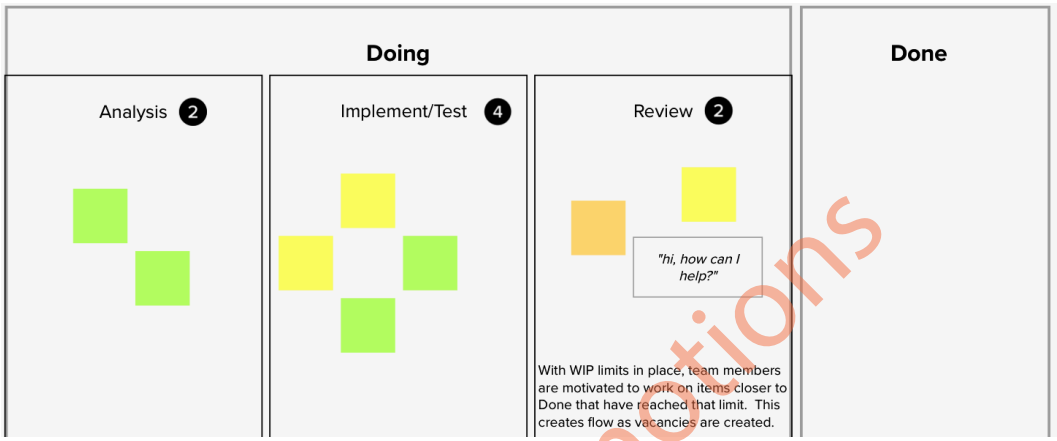


Figure 4.7 – Scenario with WIP limits, resulting in flow

Teams can select initial WIP limits as they start work. After a period of time, if bottlenecks are still seen, they can lower the WIP limits until the bottlenecks disappear, and flow at that point is achieved. Even with that bottleneck gone, other bottlenecks will appear. The Theory of Constraints, as written by Eliyahu M. Goldratt in his book *The Goal*, specifies that you move to the subsequent bottlenecks, removing them one by one, to optimize the flow for the entire process.

WIP limits are a necessary mechanism according to David J. Anderson in his book *Kanban: Successful Evolutionary Change for Your Technology Business*. WIP limits create tension for a team, encouraging the team to act in order to work and enable flow, as seen in our previous scenario. The tension also may highlight constraints and systemic impediments in the organization. Open discussion of these constraints and impediments to finding solutions leads to continuous improvement.

We've seen how limiting WIP by imposing column constraints or WIP limits may help us address bottlenecks in our development process. One other source of bottlenecks in our process is the size of our items of work. Let's take a look at keeping those at an appropriate size to attain and maintain flow.

Keeping batch sizes small

Batch size commonly refers to the size of a standard unit of work. One of the accomplishments of the Agile movement was the success of focusing delivery on smaller increments. That forced a look at reducing the batch size that could be accomplished for delivery. Reducing batch sizes in addition to

limiting WIP are important parts of accomplishing Lean flow. Donald Reinertsen noted the effects of batch size in his book *The Principles of Product Development Flow: Second Generation Lean Product Development*. Let's examine what role small batch sizes play.

Small batch sizes decrease cycle time

A key takeaway from Agile development was the delivery of value in short cycles. This had the effect of shortening the cycle time a team had to deliver an increment of value, allowing the team to look at delivering only what it could deliver by the end of the cycle. This allows us to say that batch size is directly related to the cycle time.

Keeping to large batches of work has several adverse effects. The work may not be delivered by the end of the cycle. What may be delivered may have defects that need to be fixed or reworked, increasing cycle time. The appearance of these defects also increases batch sizes, creating a *snowball effect*. The growth of the batch size and cycle time leads to cost overruns and schedule slippages.

Small batch sizes decrease risk

As we saw in the previous subsection, a large batch of work may carry with it defects that need to be fixed, impacting both the overall batch size and cycle time. Working in small batches will not eliminate the possibility of defects but will keep the number of possible defects small so that they can be managed easily.

A side effect of small batch sizes is that they result in short cycle times. This short cycle time provides opportunities for customer feedback. The opportunity for feedback negates any risk that the team is moving in the wrong direction in terms of delivering value.

Small batch sizes limit WIP

Batch sizes and WIP are directly related to each other. Directly changing one changes the other in a correlated fashion. Let's look at a few examples of this correlation at work.

A large batch size will mean there is a large number of items of WIP. As mentioned before, large numbers of WIP increase multitasking, preventing work from finishing because of increased context switching. This effect is another factor that increases cycle time.

The large batch size also creates a bottleneck in the system. This variability in flow impedes work from effectively moving, increasing cycle time.

Reinertsen also advocates in his book *The Principles of Product Development Flow: Second Generation Lean Product Development* that when optimizing between batch sizes and limiting WIP by attacking bottlenecks, start with reducing batch sizes first. This is often an easier step to implement. Reducing bottlenecks to allow for adequate flow may involve deeper changes in both process and technology.

Small batch sizes improve performance

As counter-intuitive as it may seem, large batch sizes do not create efficiencies with overhead. Time processing overhead activities are actually greater with large batches as opposed to smaller batches. The overhead compounds on large batches. Overhead costs on small batches can be reduced easily because of the shorter cycle times.

Small batch sizes improve efficiency. This may seem to also be counter-intuitive, but because small batch sizes receive fast feedback, subsequent batches of work are able to take advantage of that feedback, reducing rework. A large batch of work exposes all those problems at once, creating more work and removing any implied efficiency.

We've seen the advantages of working with small batch sizes and what happens when you work with a large batch size. The question then becomes, "How do I make sure I'm not working with a large batch size?" Let's take a look at finding out what your optimal batch size is.

Finding the ideal batch size

We have identified the benefits of working with small batch sizes. The question then becomes, "What would be an adequate enough batch size?"

Reinertsen proposes looking at batch size in terms of economics. When approaching the cost of development and determining what size of work is ideal, you need to consider two costs:

- **Holding cost**: The cost of keeping (and not releasing) what you develop
- **Transaction cost**: The cost to develop your work

An example of this comes from releasing work to a production environment. The following diagram shows the relationship between the transaction and holding cost curves if we were to look at the cost of releasing a change versus the cost of waiting to collect changes at an optimal time:

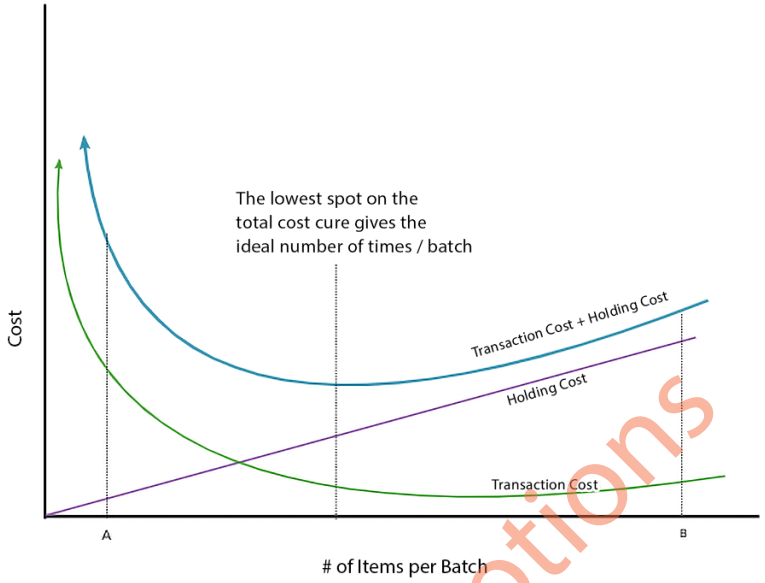


Figure 4.8 – Transaction and holding cost curves

In our preceding diagram, an example of a small change—such as a single line of software code to release—is represented by point **A**. At point **A**, we incur a very low holding cost to immediately release, but the high cost at point **A** comes from the transaction cost. We are spending a lot of time performing testing and deployment for a simple line of code.

Does this mean we should always look to consolidate changes to incur less cost? Let's look at point **B** in our preceding diagram, which collects our changes over a larger time period—say, a month. Now, the costs are reversed. The transaction cost is low as we are performing testing and releasing a large number of changes, but now our holding cost is high. Perhaps in delaying the release of our change, we missed an important market window and lost sales because our competitors released an equivalent change first.

If we wanted to figure out the *break-even* point of when to actually release a collection of changes, we would perform a *u-curve optimization*. We would take the sum of the holding cost and the transaction cost and plot it on the same graph as shown in the preceding diagram. On the graph of the total cost curve (sum of holding cost and transaction cost), find the lowest point on the curve. That's the ideal number of items to have in your batch.

How can we improve on this ideal number of items per batch? If we introduce new ways of doing things or new technology, that can affect the transaction cost, giving us a new total cost curve with which to perform a u-curve optimization. Extending our example of releasing changes into production, the use of automated testing and deployment tools that enable faster, more reliable deployments with more

frequent testing in a CD pipeline reduces transaction costs, giving us the confidence to release stories more frequently within a sprint. The new cost curves would probably resemble the following diagram:

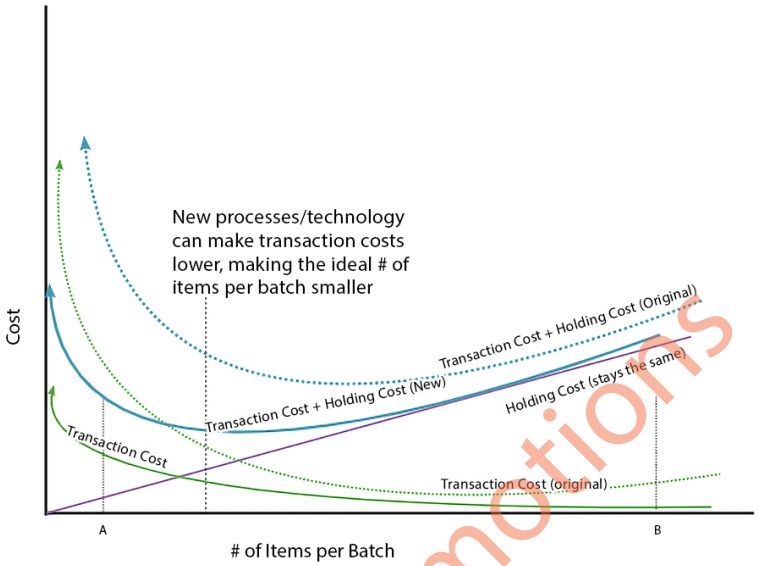


Figure 4.9 – New cost curves after optimization

We can see from the preceding diagram at point **A**, the holding cost remains the same but the transaction cost has decreased. Also, we can see that the low point of the total cost curve has shifted to the left, moving our ideal point "left," to smaller batch sizes.

So, what can our teams and ARTs do that will yield lower transaction cost curves? Adopting practices and technology such as automated testing and automated deployment is one example that certainly fits the bill. They allow for smaller batch sizes to go through and encourage flow.

In this section and the previous section, we looked at WIP and batch size as factors for achieving Lean flow. In the next section, we will see how those factors work with other factors to determine whether your system has Lean flow.

Monitoring queues

To enable Lean flow, you need to closely examine queueing theory, the mathematical field behind the behavior of queues and waiting in line (think Starbucks or the checkout area of your local supermarket!). A number of mathematical formulas will be useful in helping us make sure we understand what we have to do to ensure Lean flow. These are:

- Little's Law
- Kingman's Formula

Let's see how we can use these elements of queueing theory to enable Lean flow.

Where are our queues?

A key difference between product development queues and product manufacturing queues is that the artifacts of product development (particularly software development) are not physical and it is difficult to grasp the progress until completion, whereas in manufacturing, you can visually ascertain the completeness of a product while on the factory floor. This *invisibility factor* makes it easy to ignore these queues, often at your own peril.

Long queues unrestrained by WIP limits or small batch sizes create an array of problems, as summarized by Reinertsen. These problems include the following:

- Longer cycle times
- More risk
- More overhead
- More variability
- Lower quality
- Lower motivation

We've talked about problems with overhead, risk, and quality when talking about large batch sizes. If we consider the size of our work as the queue for our system, then the mathematical formulas that we will examine later are applicable and can model the state of our Lean flow.

How can long queues lower workplace motivation? Imagine a scenario where you were preparing work and the next part of the process was ready whenever you finished it. There would be a sense of urgency to finish your part of the work. That sense of urgency would not be there if, after you delivered your work, it sat behind a queue of other items and wouldn't be seen for weeks.

Now that we've established our batch sizes and WIP as queues, it's time to see how mathematical formulas illustrate the relationship between our queues and cycle time and variability.

Little's Law and cycle time

We have seen how both WIP and batch size correlate to cycle time, or the time it takes to deliver work once we've accepted it. Mathematically, cycle time, WIP, and batch size are tied using Little's Law.

Little's Law is an expression tying cycle time, WIP or batch size, and throughput. This expression for Little's Law is presented here:

$$L = \lambda W$$

L stands for the length of the queue. This can be thought of as either WIP or batch size. L is the throughput of work processed by the teams or ART. W gives you the cycle time or the wait time for a customer.

At this point, it's simple mathematics if we're concerned about finding out our cycle time. Written in the following manner, you can see that cycle time (W) is directly related to the size of our queue (L):

$$W = L/\lambda$$

A simple illustration of this is a backlog for a Scrum team. If their backlog has 9 user stories, each estimated at 5 story points, and their velocity (the measure of how many story points' worth of work they have been able to deliver per sprint) is 15 story points per sprint, we can take Little's Law to predict how many sprints it will take to complete that set of stories in the backlog, as illustrated in the following diagram:

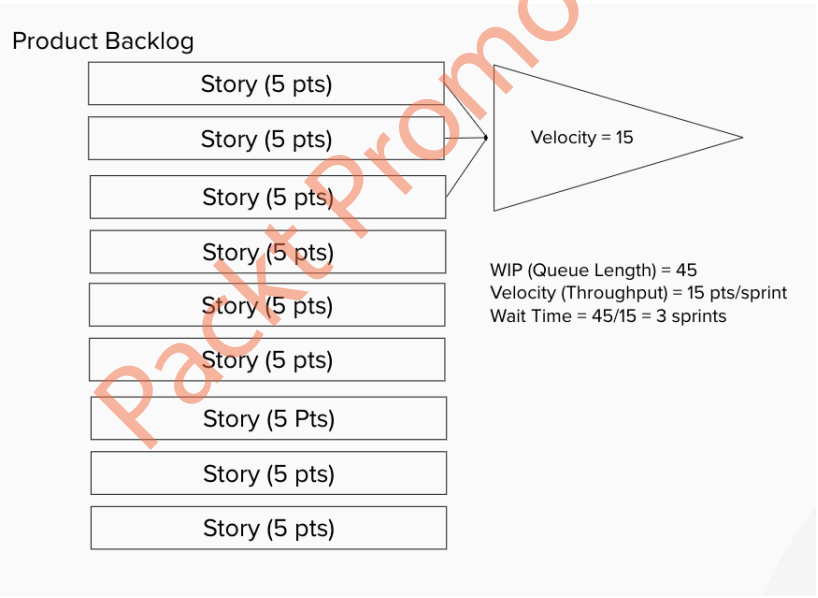


Figure 4.10 – Illustration of Little's Law

Kingman's Formula

Kingman's Formula is a mathematical model to describe how wait time equates to cycle time, variability, and utilization. The formula is illustrated here:

$$E(W) \approx (\frac{\rho}{1-\rho})(\frac{c_a^2 + c_s^2}{2})\tau$$

Each term in the formula represents an individual quantity. The first term ($\frac{\rho}{1-\rho}$) represents the utilization of the people doing the work. The second term ($\frac{c_a^2 + c_s^2}{2}$) represents the variability of the system. The third term (τ) represents the service time or the cycle time.

We want to examine the relationship between wait time, utilization, variability, and cycle time. If we substitute a letter for each term, we get the following (simpler) equation, commonly referred to as the *VUT equation*:

$$E(W) \approx U \times V \times T$$

So, this equation shows us that the total wait time for a customer is directly proportional to the utilization, variability, and cycle time.

We have previously looked at how queue size, batch sizes, and limiting WIP have an effect on cycle time. Let's take a look at the other variables of Kingman's Formula to see how utilization and variability can affect the wait time.

Utilization and its effects

Let's start with utilization. When we refer to utilization, we look at the percentage of the overall capacity of the system. A high utilization may be desired by management—after all, we don't want our workers to goof off. But after looking at Kingman's Formula, we see that wait time increases as utilization increases. We can see this effect by plotting out utilization in comparison to the queue size. The resulting curve is a nonlinear graph that starts to ascend around 60% utilization and approaches infinity at 100% utilization. This is seen in the following diagram:

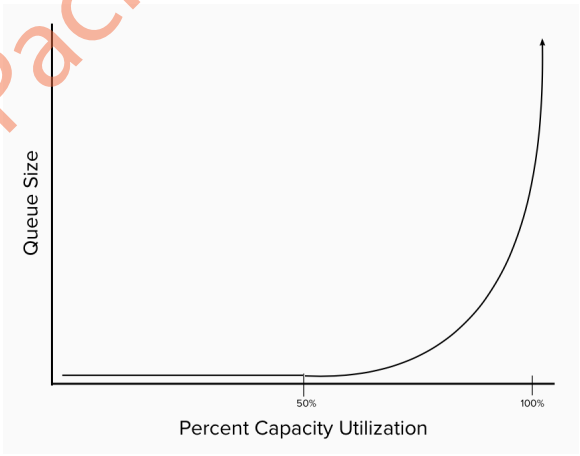


Figure 4.11 – Utilization compared to queue size

Another way of looking at this percentage utilization is seeing it as a load/capacity. If load looks at the rate at which new work comes in, capacity looks at the rate at which work leaves and is delivered to the customer. This ensures adequate utilization so that we do not take in more work than we can process.

The idea that for Lean flow to occur, scheduling slack or idle time must be introduced to keep utilization at a reasonable level (for example, not at 100%) has its roots in the Toyota Production System. *Muri* or overburdening was seen as one of the three *wastes* to eliminate in Lean. We will look at another waste: *Mura*, or unevenness. We also call this variability.

Variability, its effects, and actions

Let's explain what variability is. So far, we've been making the assumption that, as with manufacturing products on a factory floor, all work involves the same effort. But this is often not the case. Each piece of work may involve different levels of effort or even different efforts. Some work may have unknowns that require a more detailed investigation. Some work may have defects. Variability is the quality that describes the *individuality* of each piece of work.

From Kingman's Formula, we see that the effects of variability have a compounding effect on wait time. The curve in the following diagram shows the effects of high variability versus low variability when looking at utilization:

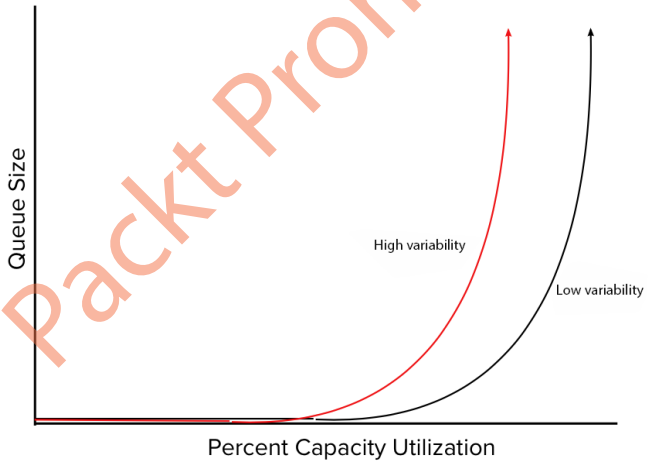


Figure 4.12 – Effects of variability on utilization and queue size

We can see from the preceding diagram that this effect is linear, but the combination produces an undesired result. With high variability, working at somewhat lower utilization will not stem the growth of your queue. You'd have to work at much lower utilization to keep your queue in control. What can you do?

It's important to understand that not all variability is bad. Some may be necessary to learn new things. So, the key is really to manage variability so that it doesn't affect the queue size and, consequently, the wait time. Ways to do that include the following:

- Limiting WIP
- Working with smaller batch sizes
- Setting up buffers in your system
- Establishing standard processes

We've spoken before about the benefits of limiting WIP and working with smaller batch sizes, so let's examine some of the other ways we can manage variability.

Setting up process buffers

Lean manufacturing looks to limit variability by setting up buffers. These buffers are used to limit the following factors:

- Inventory
- Capacity
- Time

In product development, WIP limits and small batch sizes act as inventory and capacity buffers accordingly. To set up a time buffer, you can establish buffer states on those WIP columns on your Kanban board where variability exists in your process. An example on our Kanban board is shown in the following screenshot:

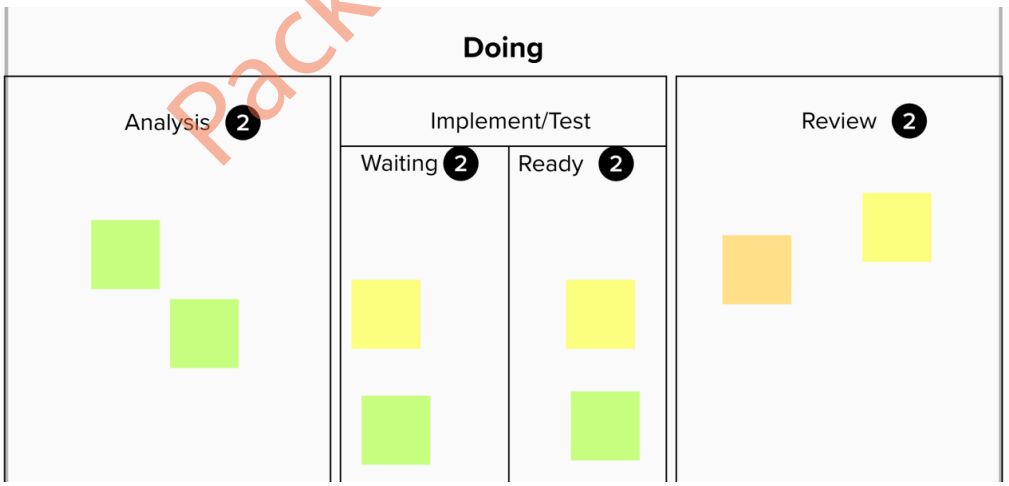


Figure 4.13 – Kanban board with buffer states in the Implement/Test column

Note that WIP limits are established in each buffer state to ensure throughput is maintained.

Establishing standard processes

In the previous section, we discussed using several types of buffers to ensure that variability is managed at several stages of the process. The variability managed by buffers is often the type that is tolerated, even encouraged by the nature of the work.

Some variability, however, may exist due to inefficiencies in the development process. An example of this is encouraging one type of test to be automated—for example, unit tests for code—without striving to automate **behavior-driven development** (**BDD**) tests to ensure correctness. This has the effect of keeping cycle times long.

Establishing standard processes ensures the reduction of needless variability of cycle time. Standardizing processes includes establishing a standard, detecting any possible problems, and discovering the root cause of these problems continuously.

We've seen the practices you can use to model your development process to ensure that the work is progressing to completion. These practices need to be anchored by people that look at the entire process from start to finish. In the next section, we'll take a closer look at establishing that viewpoint and the systemic changes needed.

Moving from project-based to product-based work

When adopting value stream thinking, it's important to change the view and mindset of development from project-based management to product-based management.

In *Project to Product: How to Survive and Thrive in the Age of Digital Disruption with the Flow Framework*, Mik Kersten contrasts the success of integrating IT and software in assembling BMW models in Leipzig, Germany, with the failure of Nokia to continue its dominance in the smartphone industry as the iPhone and Android phones were introduced. He notes that although Nokia had successfully adopted Agile practices, it did not appear to foster change to the entire product development process or affect the entire organization. He identifies that for this *Age of Software*, product-oriented development using value streams allows for the creation and maintenance of successful products.

Mik Kersten highlights seven key categories where the differences between project-based management and product-based management are apparent. These differences are illustrated as follows:

- Budgeting
- Timeframe
- Success
- Risk
- Team assignments

- Prioritization
- Visibility

Let's individually look at these differences now.

Project budgets versus value stream funding

In project management, the *iron triangle* is a well-known construct. In management, you look at three factors or *sides* to see whether you can fix one or more of them to bring a project or product to completion. These three factors are as follows:

- Resources (people, equipment, facilities, and so on)
- Scope
- Time

A project budget is often a bet. Can the project do everything required (scope) in the projected timeframe (time) and with the projected resources? Because this budget is necessary for approvals, it is often a guess of the largest number of resources for the other parts of the triangle. Sometimes these guesses fall short, resulting in cost overruns and possibly another project (with another budget).

Funding for value streams is easier. Resources and time are kept constant over the period of the budget. At the end of the timeframe for the budget (often less than an annual basis; typically, quarterly) the results of the development and the customer feedback determine whether continued effort and capacity are needed to match the demand for delivered features. If so, a new budget with an increased allocation is created for the new time period.

Defined endpoint versus product life cycle

A defining characteristic of the project is its life cycle There is a beginning, with a flurry of activity to organize teams and get started. This then proceeds along with development, continuing until the project reaches its conclusion: the delivery of a product. At this point, the project is over. With the project and its funding at an end, the teams that developed the product are dispersed. The product goes to a dedicated maintenance team that may not have had a role in its development, consequently preventing learning from reaching the entire organization.

Value streams approach the timeline of the entire product *from cradle to grave*. The same teams and ARTs that develop and release the initial features of a product become responsible for its maintenance and ongoing health. Part of maintenance includes identifying and removing technical debt, resulting in keeping the product viable. This occurs until the product reaches the end of its life.

Cost centers versus business outcomes

Measurement of progress in project-based development is often aligned with how the cost centers that comprise teams are performing. This disjointed view of progress focuses on the performance of individual silos rather than the entire system.

Another consequence of this cost-center approach is that because the budgets tend to be large, the project stakeholders are drawn to create large projects as opposed to viewing the value of efforts once delivered.

Value streams use a different metric for success. They look at the outcomes produced by efforts delivered. By allowing for incremental delivery and learning from customer feedback, the value stream can adjust to deliver better business outcomes.

Upfront risk identification versus spreading risks

In project-based development, risks to delivery are identified as early as possible to create contingencies. But frequently, there are risks not identified due to the unknowns yet to be discovered when development is underway.

With product-based development, risks are identified as more learning occurs. Incremental delivery allows for pivots to occur at regular checkpoints. Although there may be overhead involved in these pivots, this overhead is spread out over the product life cycle and becomes a small fraction as it is distributed over that long period of time.

Moving people to the work versus moving work to the people

Project-based development creates teams from cost-center-based resource pools. This method of creating teams to work on projects assumes that individuals from the resource pools have identical talents and skills. This assumption is never true.

Another consequence of this reallocation of these individuals interferes with the team's well-being and productivity. In 1965, psychologist Bruce Tuckman created the **Forming, Storming, Norming, Performing, and Adjourning** (**FSNPA**) model of team creation, illustrating the stages teams go through as they become high-performing. Constant reassignments and team shakeups impede the ability of the team to achieve a high-performing status where the team works as one unit.

Product-based development emphasizes long lifespans for teams. This allows teams to focus on acquiring product knowledge and grow together as one team. This improves the ability of the team to deliver and the team's morale.

Performing to plan versus learning

In project-based development, adherence to the project plan is paramount. Adjustments to the plan result in cost overruns and reallocation of resources due to the overhead cost of performing any changes.

Product-based development welcomes changes by setting up piecemeal delivery of features and creates a space for learning and the validation of hypotheses. After the delivery of each increment of value, feedback and outcomes are collected and necessary adjustments are made.

Misalignment versus transparent business objectives

In project-based development, there is often a disconnect between the business stakeholders and the IT departments that develop products. This disconnect stems from IT's focus on the product and the business side's focus on completing the project through a progression of steps with no runway for readjustment.

With product-based development, the goals for both the business and IT development sides are the same: the fulfillment of business objectives. This transparency of the goal allows for alignment to occur and for easier sharing of progress and feedback.

Summary

In this chapter, we looked at Lean flow as part of the CALMR model. We know that achieving a Lean flow of work where the progression of work is steady and teams are neither overburdened nor underburdened allows for the success of the other parts of the model. To that end, we took a close look at Lean practices that allow teams to achieve Lean flow.

The first practice we investigated was to make sure that all the work a team commits to and its progress are visible. To ensure this visibility, we looked at the types of work a team could do. We then mapped that work to a Kanban board, highlighting the features of the board that allow us to see the progress for any one piece of work and where urgent work could go. We saw how we can visualize WIP on the Kanban board and how to keep WIP in check using WIP limits.

From there, we took a look at the size of the work or batch size. We strove to understand the importance of making sure the batch size was as small as possible. We looked at the relationship batch size had with cycle time, WIP, and performance. With this in mind, we looked at the economics of batch size and how you could determine the ideal batch size through u-curve optimization.

We then looked at other factors at play by looking closely at queueing theory. We saw how Little's Law describes the relationship between cycle time and WIP or batch size. We saw the other factors and how they related to cycle time by examining Kingman's Formula. Approaching one of these factors, utilization, we saw how cycle times increased with high utilization. We also learned more about the other factor, variability, and how to manage it.

Finally, to allow value streams to deliver work through Lean flow, we looked at the differences between project-based development and product-based development. These differences create stronger teams and stronger products in shorter cycle times.

To ensure that work is progressing in Lean flow, we need to take regular measurements. We also want to make sure that whatever is delivered does not adversely affect the staging and production

environments. To do this, we will examine measurement, the next element of our CALMR model, in the next chapter.

Questions

Test your knowledge of the concepts in this chapter by answering these questions.

1. Which of these are types of enablers in SAFe (pick two)?

   A. Exploration
   B. Measurement
   C. Visualization
   D. Compliance
   E. Actual

2. Which feature of a Kanban board can be used to visualize work that is urgent?

   A. WIP limits
   B. An expedite lane
   C. Column policy
   D. Expanded workflow columns

3. What is a consequence of too much WIP?

   A. Too much multitasking
   B. Reduced cycle times
   C. Newer work gets started after older work is completed
   D. Short queues

4. What results from large batch sizes?

   A. Low WIP
   B. Decreased risk
   C. High cycle times
   D. High performance

5. According to Reinertsen, long queues lead to which problems (pick two)?

   A. Shorter cycle times
   B. Higher overhead

C. Less risk

D. Higher quality

E. More variability

6. Which of these is a method for managing variability?

A. Larger batch sizes

B. Establishing buffers

C. "One-off" processes

D. Increasing WIP

7. According to Mik Kersten, which of these is present in product-based development?

A. Moving the people to the work

B. Identifying all risks upfront

C. Focus on business outcomes

D. Project planning

Further reading

Here are some resources for you to explore this topic further:

- *Making Work Visible: Exposing Time Theft to Optimize Work and Flow* by *Dominica DeGrandis*: A look at five time thieves and how Lean practices can remove them. Too much WIP is identified as one of these time thieves.
- *The Goal* by *Eliyahu M. Goldratt*: A look at the Theory of Constraints and how to eliminate bottlenecks in your process.
- *Kanban: Successful Evolutionary Change for Your Technology Business* by *David J. Anderson*: The authoritative source on Kanban. Further exploration of the Kanban board and limiting WIP can be found here.
- *The Principles of Product Development Flow: Second Generation Lean Product Development* by *Donald Reinertsen*: An exhaustive look at the economics behind Lean practices in this chapter, whether limiting WIP, identifying the ideal batch size, or the effects of utilization and variability the effects of utilization and variability.
- *Project to Product: How to Survive and Thrive in the Age of Digital Disruption with the Flow Framework* by *Mik Kersten*: A look at value streams and measuring their performance.
- A look at teamwork models including the FSNPA model from Bruce Tuckman: `https://www.atlassian.com/blog/teamwork/what-strong-teamwork-looks-like`

5

Measuring the Process and Solution

We measure to see how far our efforts have progressed. This measurement has multiple dimensions. We examine whether we are achieving Lean flow and delivering work properly. Following that, we deploy the work to see if the changes work in the environment and whether the environment is safe. Finally, we gauge whether the changes created add value and are worthy of continued development.

In this chapter, we will look at the measurements used to answer the questions posed above. We will look at metrics to determine the following:

- Whether we are on track to deliver the solution
- The health of our environments before and after our solution is delivered
- Whether our solution is delivering on hypothesized value

Measuring solution delivery

In *Chapter 4*, *Leveraging Lean Flow to Keep the Work Moving*, we looked at practices to establish and maintain Lean flow. We need metrics to evaluate whether we have achieved that Lean flow and whether we can be predictable in our commitments.

Common metrics to evaluate Lean flow are outlined in the following list:

- Cycle time
- WIP
- Throughput
- Blockers or bottlenecks in the process

A useful tool to obtain these metrics is a cumulative flow diagram. We will take a close look at a cumulative flow diagram to find these metrics.

Cycle time

For a value stream, cycle time is the time it takes for a piece of work to be delivered after it has been accepted and goes through the entire development process.

We saw in *Chapter 4*, *Leveraging Lean Flow to Keep the Work Moving*, the many factors that can affect cycle time. These factors included too much WIP, large batch sizes, high utilization, and variability.

Another factor that can affect cycle time is if there is waste in the development process. This waste may come in the form of delays or *wait times* as work gets handed off from one phase to the next.

Regular measurement of cycle time allows us to see how long it takes for a value stream to deliver work. Increases in cycle time can allow us to determine if the root cause is one of the factors mentioned. Once determined, corrective steps can be taken to put cycle time back to its previous length of time.

Lead time

Lead time and cycle time are frequently confused with each other. The big difference between them is the perspective: while lead time is from the customer's perspective, cycle time is from the perspective of the value stream and is an internal metric.

Lead time is the length of time a customer waits for delivery of an item of work after a request for that work. As shown in the following diagram, lead time is made up of cycle time as well as any *wait time* for the value stream to accept and start the work.

Figure 5.1 – Illustration of cycle time and lead time

From the preceding illustration, we can improve two times to improve the lead time. We could improve the wait time or we could improve the cycle time. Most organizations strive to improve cycle times with the added benefit of improving the lead time.

WIP

By now, we should be familiar with **WIP** or **Work in Progress/Process** as the work that a value stream has started but not finished. We have seen the ill effects and consequences of too much WIP.

WIP is visible on the Kanban board shown in the following illustration.

Figure 5.2 – Kanban board with WIP highlighted

In the preceding example, we can count the items and determine our WIP is six.

Throughput

Up to this point, we have looked at metrics that are units, either of time or quantity. Throughput is the only metric we look at that is basically a rate. In other words, how much work is accomplished in a given unit of time.

Throughput is basically the number of units of work completed in a set period of time. Scrum practitioners know of this metric as velocity: the sum total of story points for stories completed in a sprint (a fixed period of time, often two weeks). Other Agile methodologies look at other units of work (stories, features, etc.) delivered per standard unit of time (week, month).

We can see that cycle time has an inverse relationship with throughput for a value stream. The greater the throughput, the shorter the cycle time.

Blockers and bottlenecks

To ensure Lean flow, we need to find and resolve those impediments that are preventing that flow from occurring. These may be temporary or systemic blocks preventing one or more pieces of work from progressing through the process.

Vigilance on these blockers may require daily checks to see if the blockage is still occurring and escalation if resolving the block has not occurred for several days.

Measuring with cumulative flow diagrams

A cumulative flow diagram is an easy way to examine cycle time, lead time, throughput, and WIP. You can also inspect and find bottlenecks in your process.

The cumulative flow diagram is a graph of the history of work that the value stream works through over time. The *x* axis of the cumulative flow diagram refers to time. The *y* axis refers to the count of work. Each band in the cumulative flow diagram refers to a step in the workflow that is derived from columns on a Kanban board. An example cumulative flow diagram follows.

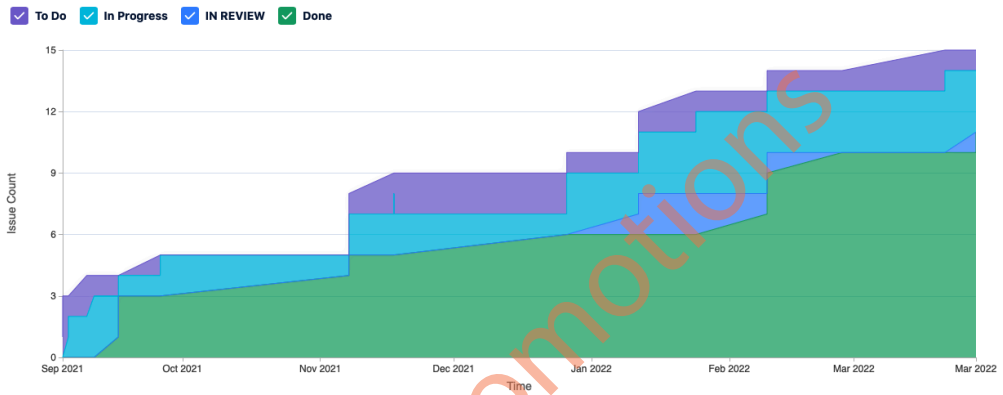


Figure 5.3 – Cumulative flow diagram

In general, we should see the bands move upward and grow from left to right. The entrance and exit lines for a band that represents a process step in the workflow should be roughly parallel in a situation where flow occurs. If the entrance and exit lines grow apart, creating a large growing area, that's an indicator that a bottleneck exists in that part of the process. We can see such a bottleneck in the following diagram.

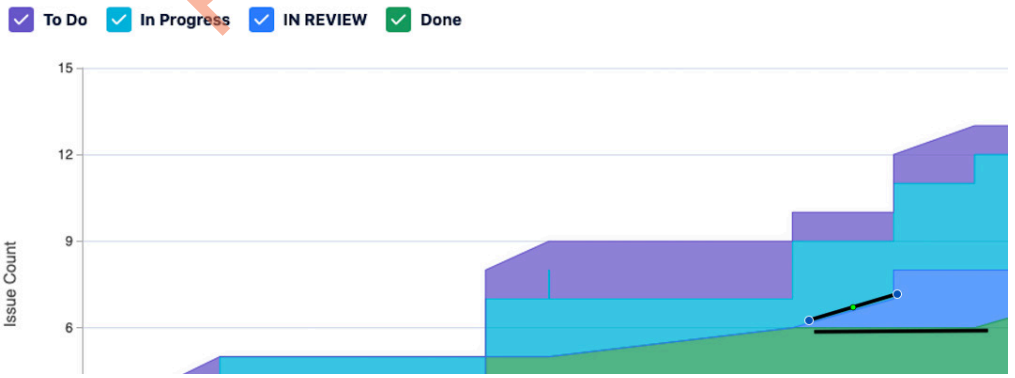


Figure 5.4 – Annotation of bottleneck in cumulative flow diagram

Let's take a look at how we can use the cumulative flow diagram to measure cycle time, WIP, and throughput.

Measuring WIP with the cumulative flow diagram

To measure the WIP that a value stream has at a given point in time, do the following:

1. Start at the point where the boundary between the band corresponding to the Backlog or **To Do** column and the band corresponding to the first **In Progress** column resides. In our example, this is the boundary between the **To Do** band and the **In Progress** band. We want to look at only the work that has started and, thus, beyond the **To Do** state.
2. Make a note of how many units of work that is.
3. Draw a vertical line down through the other bands until you reach the boundary between the band representing the last **In Progress** column and the band representing the **Done** column. In our example, we draw the line through the **In Progress** band and the **IN REVIEW** band. Make a note of how many units of work that is.
4. Subtract the second number from the first number.

The following diagram illustrates the method described.

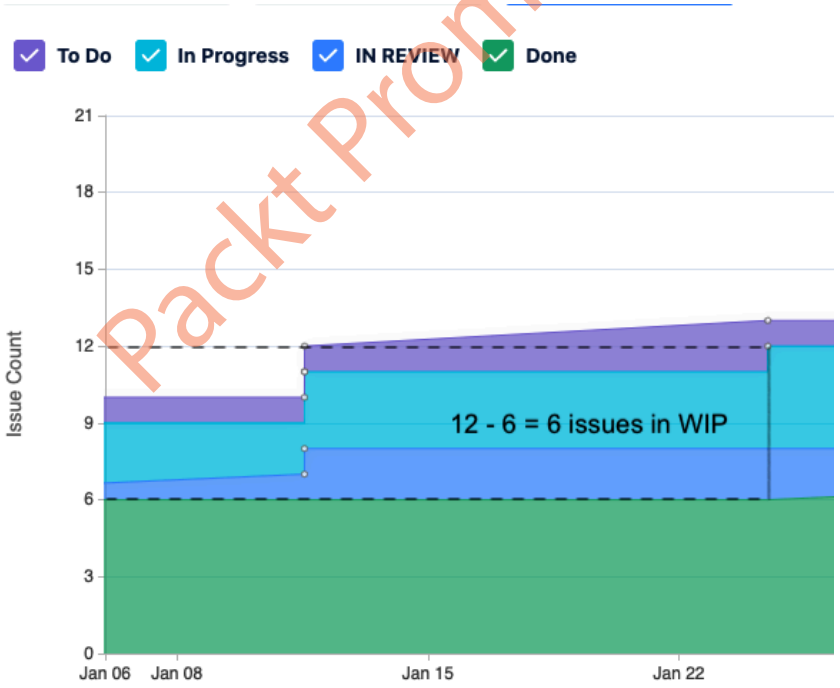


Figure 5.5 – Calculation of WIP

Note that you can use the same method on a smaller scale to find the number of **In Progress** issues in a single column at a point in time by looking at the two lines that form the band for the column of interest.

Measuring cycle time with the cumulative flow diagram

To measure cycle time, do the following:

1. Find and mark a spot when an issue has moved from the band representing the **To Do** or Backlog column and note its date.
2. Draw a horizontal line until you reach the spot where it touches the band representing the **Done** column. Mark the second date.
3. Subtract the first date from the second, and you will get a measure of the cycle time for one issue.

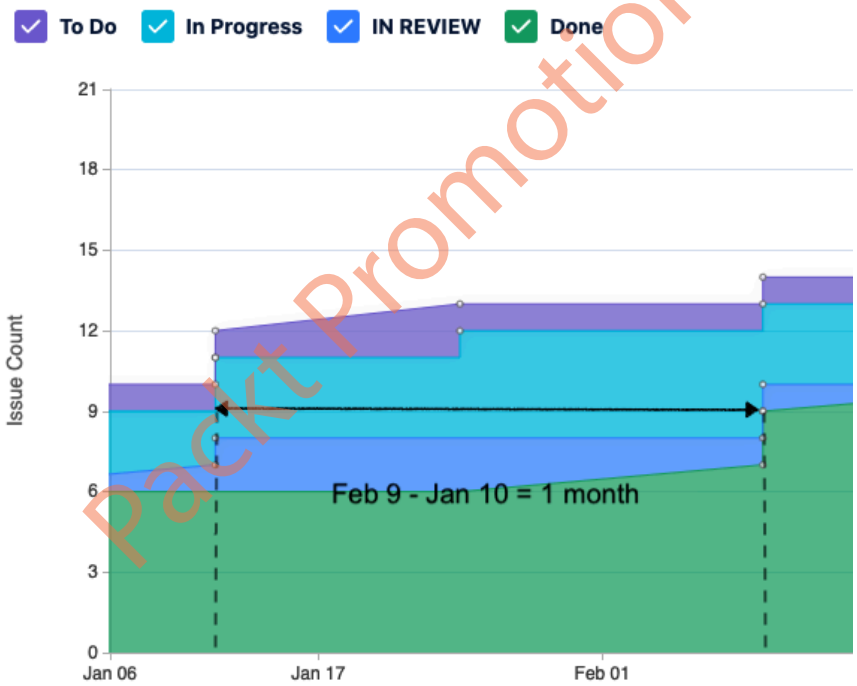


Figure 5.6 – Cycle time calculation

The steps are illustrated in the preceding diagram.

Measuring throughput with the cumulative flow diagram

Throughput on a cumulative flow diagram is done by finding a line and determining its slope:

1. Mark one point of the line and make a note of the date and number of issues.
2. On a higher point on the line, make a note of the date and number of issues.
3. The numerator will be the number of issues on the second point minus the number of issues on the first point.
4. The denominator will be the period of time between the two points.

An example of this calculation is shown in the following example.

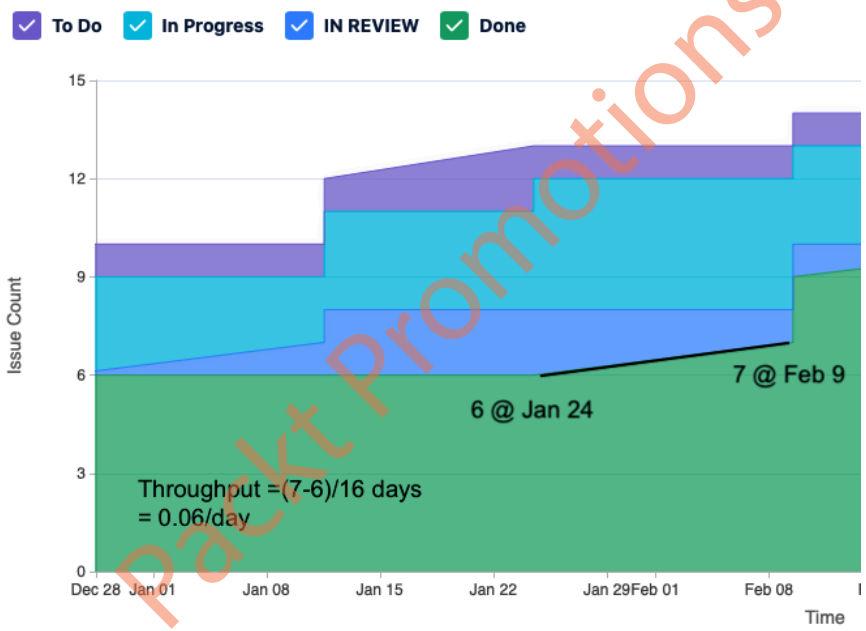


Figure 5.7 – Throughput calculation

Note that areas of no throughput will appear as horizontal lines.

So, for our examples, we have found the following measurements:

- On January 22, our WIP was six issues
- Our cycle time for one issue was one month
- Our throughput for issues moving from **IN REVIEW** to **Done** was 0.06/day

Having evaluated the measurements taken to ensure Lean flow and timely delivery of solutions, we need to take a closer look at ensuring that quality is present during deployment and before release. To do this, let's look at the important measurements to capture from the staging and production environments.

Looking at full stack telemetry

We now turn our attention from the process of developing the product to the product itself. Throughout the development process, we ran testing in our CI/CD pipeline. The passing of those tests is an indication that the product is working as intended. Now, as the product is deployed in multiple environments, we want to ensure proper operation. To do this, we monitor performance in those environments.

Monitoring is the act of measuring our environment. We measure or capture the following three key types of data:

- **Logs**: Logs are an indication that a notable event has occurred. These events may be classified to determine their severity.
- **Traces**: Traces show the path inside an application and the messages sent by the application for a given business transaction. Timing information may also be included. The information from traces helps determine the correct function and performance when troubleshooting.
- **Metrics**: Metrics are indicators of the current state of the system and its components. Periodic measurements are taken to determine good or bad trends over time.

We collect these types of data artifacts through monitoring to answer the following questions about the environment that contains the system and its components:

- **Security**: Are security vulnerabilities present? Has the system been hacked?
- **Performance**: Is the system performing properly? Are transactions following the expected paths?
- **Reliability**: Is the system up and performing well?

To answer these questions, monitoring takes measurements from multiple perspectives. A few of the key perspectives follow:

- Application performance monitoring
- Infrastructure monitoring
- Log management
- Network monitoring
- Observability

Let's take a closer look at these elements of full stack telemetry.

Application performance monitoring

When looking to see if an application is performing adequately in an environment, there are usually two perspectives that are taken into account:

- How do users perceive the performance of the application? This may be seen as user load or response times.
- How much of our resources (e.g., allocated memory) is the application using? This is typically measured as a change in capacity from a previous state or configuration.

In 2010, Gartner looked at expanding these sets of measures into five dimensions called the APM Conceptual Framework. These dimensions are the following:

- End user experience (e.g., response times)
- Runtime application architecture (e.g., garbage collection events)
- Business transactions
- Middleware components (e.g., database reads and writes)
- Application analytics

Many of the applications that have their performance measured by application performance monitoring are web-based applications with a web browser or mobile app user interface. The reason for this is the easy collection of metrics that span the five dimensions.

Many monitoring tools, especially ones detailed in *Chapter 3*, *Automation for Efficiency and Quality*, have features that touch all five of these dimensions.

With the growing popularity of containers and microservices, application performance monitoring has had to look at creating accurate measurements of performance within a container or microservice. Because of this, there is some overlap between application performance monitoring and infrastructure monitoring.

Infrastructure monitoring

Infrastructure monitoring is used to measure the resources of the entire system or environment. This is usually identified in terms of the following utilizations:

- CPU utilization
- Memory (RAM) utilization
- Storage availability

Historically, infrastructure monitoring was important for on-premises equipment, such as *bare-metal* servers. With the growth of cloud environments, infrastructure monitoring tools need to measure dynamically allocated resources that will be instantaneously created and destroyed depending on need.

Network monitoring

Network monitoring measures the performance of the organization's network to discover slow performance or outage situations. Generally, network monitoring systems keep track of the following areas:

- Availability of network resources (e.g., connection uptime, connection speed)
- Networking hardware status
- Network interface state

A growing area for network monitoring is ensuring that security in terms of integrity, accessibility, and privacy is maintained. Some network defenses to ensure security include the following:

- **Network Access Control** (**NAC**)
- Firewalls
- Antivirus/anti-malware software
- **Virtual Private Networks** (**VPNs**)
- Email security
- Application security
- Cloud security

Log management

All the tools that perform monitoring will create logs of all activities and measurements, from critical alerts to informational notices. Log management collects these events as well as performing the following activities:

- Aggregation into a central location
- Storage
- Log rotation and disposal
- Analysis
- Search and reporting

Properly managed logs are needed both during the development process and afterward for several purposes. These include the following:

- Determining whether tests pass
- Troubleshooting application and environment failures
- Evaluating customer feedback

While collecting the logs, traces, and metrics is important, the ability to sift through all that data when something goes wrong is also important. For that, we turn to observability. Let's discuss what to do with that monitoring information.

Observability

The mountain of data created by collecting logs, traces, and metrics through monitoring now introduces the following new questions:

- When something wrong occurs, can I quickly find the data in the logs, traces, and metrics to understand the root cause and find a solution?
- Can I use the data I've collected in logs, traces, and metrics to predict when problems may occur and try to prevent them from occurring?

Observability looks to answer these questions by shifting from collecting the data to understanding the data itself to identify its current state. Doing so moves beyond just monitoring to include the following activities:

- Analysis of logs, traces, and metrics
- Looking for correlation between the logs, traces, metrics, and specific events, such as outages
- Visualization and exploration of data through dashboards
- Alerts and notifications when problems occur

Observability helps enforce a higher level of systems thinking by allowing teams to understand the cause and effect the changes have on the environment. By understanding what causes changes beneficial to the environment, teams can incorporate the correct changes for better outcomes.

Having ensured that the solution is of sufficient quality, we come upon the greater test: does the solution developed provide sufficient value to our customers to the point that they will love it? To evaluate this, we need to take measurements of outcome-based metrics. This is the topic of the next section.

Measuring the value proposition

It's difficult to apply an objective measurement to a subjective quality such as value. What we can do is measure actions taken by our customer or feedback that our customer has given in terms of survey responses or thoughts during specific events such as reviews.

When looking at the measurements for value, some metrics may be collected too late to allow a value stream to know when to pivot. Examples of this include **Profit & Loss** (**P&L**) or **Return on Investment** (**RoI**). So, other metrics that can act as leading indicators are needed. The practice of collecting leading indicators to allow pivots or continued development is called *innovation accounting* by Eric Ries in his book, *The Lean Startup*.

The Innovation Accounting framework

Ries describes the method of hypothesizing and learning as the **Innovation Accounting** framework. In this framework, teams work to establish hypotheses of what the customer wants, develop those hypotheses, and then measure and determine further actions. This is done in a three-phase cycle in which the following is done:

1. Look at the present situation and establish a baseline hypothesis through a **Minimum Viable Product** (**MVP**) or development of *just enough* of a product or its features to bring to a customer and obtain feedback as validation of the hypothesis.
2. Based on metrics, make small adjustments to the MVP.
3. If the metrics indicate success, continue the refinement of the MVP. If metrics tell you otherwise, pivot to something new.

In the Innovation Accounting framework, the metrics gathered must be carefully chosen, so they don't end up being **vanity metrics**. Vanity metrics tell a good story but provide no meaningful insights. To avoid a metric from falling into the vanity metric category, Ries recommends they have the following three qualities:

- **Actionable**: Does the metric show clear cause and effect? In other words, could you replicate the results by performing the same actions?
- **Accessible**: Does everyone on the value stream have access to the same data, and is that data understood by all?
- **Auditable**: Is there credibility to the report?

Metrics that work with Innovation Accounting will depend on the end product developed by the value stream and customer interaction with the product. We will look at a few of the leading metric collections used to measure value.

Pirate metrics

For many products available and sold to end user customers, such as **Software as a Service** (**SaaS**) or web-based products, *pirate* metrics offer the best guidance on whether development and marketing efforts are bearing fruit.

Pirate metrics were devised by Dave McClure in 2007. They get their name from the acronym formed when we look at the first letter of the five phases (**AARRR**), which sounds like the noise a pirate makes. These phases are the following:

- Acquisition
- Activation
- Retention

- Referral
- Revenue

Let's take a close look at each phase.

Acquisition

In this phase, metrics look to determine the number of new visitors to the product. The metrics here are used to capture the effectiveness of the following methods to attract new users.

- **Search Engine Optimization** (**SEO**)
- Social media
- Advertising
- Marketing campaigns

Activation

In the activation phase, we look to see how many new users want to become more engaged after first encountering the product or website. Metrics here look at the following user actions as indications that new users are beyond the acquisition phase and into this phase:

- Visiting additional pages after the landing page
- Looking at or playing with additional product features
- Spending more time on the website
- Signing up for the newsletter/email list
- Signing up for a free trial

Results of tests recording user behavior, such as A/B tests, are used to create these types of metrics.

Retention

In this phase, we look to see how long we can keep users after their first encounter. We want to measure whether the following things occur:

- Return visits to the website
- Opening email newsletters
- Repeated use of the product over a given period of time (often the first month)

Often, these metrics measure the effectiveness of marketing emails and campaigns.

Referral

This phase looks to see if the established customer spreads the word to friends or coworkers. The metrics to determine entry into this phase include the following:

- Number of referrals who enter the acquisition phase
- Number of referrals who enter the activation phase

Revenue

Finally, this phase looks at those customers that have reached a higher level and purchased enhanced product features or website content. We look to see which customers do the following:

- Pay the minimum revenue
- Pay enough revenue for marketing efforts to *break even* throughout all phases

The Google HEART framework

The HEART framework was devised by Kerry Rodden, Hilary Hutchinson, and Xin Fu at Google. It was developed to gauge the effectiveness of **User Experience** (**UX**) design for the large-scale web applications offered by Google.

The framework has team members evaluate the following dimensions:

- **Happiness**: These are typically subjective measures such as user satisfaction, perceived ease of use, and likelihood to recommend.
- **Engagement**: This tracks how involved the user is with the product and its features.
- **Adoption**: This tracks the number of new users in a given time period.
- **Retention**: This tracks how many users in a given time period are still engaged at a later time period.
- **Task success**: This measures the overall usability of the product or individual feature in question. Can a user accomplish the desired outcome using the product or feature?

For each dimension, the team looks at setting three qualities. These qualities are outlined in the following list:

- **Goals or objectives**: What is the goal for the product or feature?
- **Signals**: How do we know that we achieved the goal or objective?
- **Metrics**: What metrics can we collect to get the signals we want?

Teams will often take the dimensions and qualities and arrange them in a table. An example of this is in the following diagram.

	Goals	**Signals**	**Metrics**
Happiness			
Engagement			
Adoption			
Retention			
Task success			

Table 5.1 – Google HEART framework

The next metrics framework looks for alignment between customer desires and organization efforts. Let's look at this framework now.

Fit for Purpose metrics

In *Fit for Purpose: How Modern Businesses Find, Satisfy, & Keep Customers*, authors David J. Anderson and Alexei Zheglov describe the **Fit for Purpose** (**F4P**) framework, a way of aligning customer needs (*purpose*) with the products and solutions an organization may offer (*fit*).

Anderson and Zheglov identify four classifications of metrics that organizations typically collect and show where they fit in the framework. The classifications are as follows:

- Fitness criteria
- General health indicators
- Improvement drivers
- Vanity metrics

Let's look at each category now.

Fitness criteria

Fitness criteria describe the metrics your customer will use to determine whether your product or solution fits their needs or purpose. These fitness criteria will be based on the following dimensions:

- Design
- Implementation
- Service delivery

Customers will evaluate fitness criteria using thresholds. The first is a minimum acceptance level, below which they will decide the product or service does not meet their need. The second threshold is for exceptional service or for situations where the product in all three dimensions exceeds expectations.

Many organizations start with these metrics as fitness criteria until they find out more about the customer or market segment:

- Lead time and its predictability.
- Quality and its predictability.
- Safety, including compliance with necessary laws and standards if in a regulated industry.
- Price. Note that this is often not an independent variable. In other words, the price may be sacrificed for higher expectations of the other fitness criteria.

Customer input into their fitness criteria is often acquired through F4P surveys, which ask customers what three purposes they were seeking, how well the product or service met those purposes, and any other notes.

General health indicators

General health indicators are metrics internal to the organization. They can be used to determine suitability to improve one or more of the dimensions (design, implementation, service journey) that customers use to align an organization's fit to their purpose. Although these are important metrics and should be collected and analyzed, they are secondary to fitness criteria in determining whether value has been created to the point that customers will select the product.

Examples of general health indicators include cycle time for a value stream, velocity for a Scrum team, and mean time to restore/recover for an operations team.

Improvement drivers

Improvement drivers are metrics used by organizations for the purpose of improvement. They are the measurement that an organization is adopting a specific behavior. There is usually a target associated with them, and when that target is achieved, that metric should be changed to a general health indicator.

An example of an improvement driver could be the number of tests automated. This metric would be there to encourage an emphasis on test automation.

Vanity metrics

We've discussed vanity metrics before when talking about Innovation Accounting. Many organizations do collect vanity metrics, and they do provide an emotional boost. But like sugary snacks or junk food with their *empty calories*, these metrics offer no real insight into whether an organization's actions are bringing the true value that a customer wants.

An example of this is the total number of website visits that an organization's website generates. Without diving deeper into details, this is a number that will always go up and doesn't relate to any efforts the organization is making in marketing or SEO.

Net promoter score

Net promoter score is a common metric used in customer support. This usually takes the form of a single survey question: how likely are you to recommend the product or service to a friend or colleague? This question is accompanied by a range of responses from 1-10. The users then divide the respondents into the following categories based on their response:

- Promoters (9-10)
- Passives (7-8)
- Detractors (1-6)

The score is then calculated by looking at the percentage of promoters and detractors and subtracting the percentage of detractors from the percentage of promoters. This percentage is then expressed as an integer.

We have now completed our exploration of popular metrics frameworks used to measure the value we deliver to our customers. Most of these involve direct contact with the customer and it is beneficial if this contact is made on a frequent basis.

Summary

In this chapter, we looked at the measurements we make during both the development process and afterward. We found that we had to make measurements in three areas to determine whether our value stream was optimized and working toward value.

The first area examined using measurement was the teams and value stream to see if they were working toward optimizing flow. Measurements such as cycle time, lead time, WIP, and throughput were defined and determined using a cumulative flow diagram.

The second area measured the environments where the applications reside. Tools are available to ensure the proper functioning of the application and the environmental resources, such as the infrastructure, including storage and networks.

The third area measured the value provided by the solution. Metric frameworks, such as pirate metrics (AARRR), the Google HEART framework, and F4P metrics, look at the customer actions and thoughts to determine whether the solution developed works with customer expectations to provide value.

In our next chapter, we'll finish our examination of CALMR by taking a look at *Recovery*. We will look at the methods value streams will use to mitigate the risk of a failure in production and the methods they use to quickly resolve issues in production if they happen.

Questions

Test your knowledge of the concepts in this chapter by answering these questions.

1. Lead time measures wait time and _____?

   A. throughput
   B. blockers
   C. cycle time
   D. WIP

2. When measuring throughput and cycle time, if cycle time decreases, what happens to throughput?

   A. Throughput goes up
   B. Throughput goes down
   C. Throughput stays the same
   D. Throughput goes up, then down

3. How does WIP present itself on a cumulative flow diagram?

   A. Horizontal line
   B. Vertical line
   C. Upward slope
   D. Downward slope

4. What measurements are taken for infrastructure monitoring (pick 2)?

   A. Garbage collection rate
   B. CPU utilization
   C. Response rate
   D. Storage availability
   E. Network availability

5. Which type of metric is valued by customers in the Fit for Purpose framework?

   A. Fitness criteria
   B. General health indicators
   C. Improvement drivers
   D. Vanity metrics

Further reading

- `http://www.gartner.com/DisplayDocument?id=1436734&ref=g_sitelink` – A Gartner report that describes the APM Conceptual Framework.
- `https://500hats.typepad.com/500blogs/2007/09/startup-metrics.html` – Blog page of Dave McClure where he describes pirate metrics (AARRR).
- `https://research.google/pubs/pub36299/` – Paper submitted by Kerry Rodden, Hilary Hutchinson, and Xin Fu outlining Google's HEART framework.
- *The Lean Startup* by Eric Ries – An examination of the Lean Startup Cycle. Part of the cycle includes a discussion of Innovation Accounting and what metrics are truly beneficial and not vanity metrics.
- *Fit for Purpose: How Modern Businesses Find, Satisfy, & Keep Customers* by David J. Anderson and Alexei Zheglov – An examination of the Fit for Purpose Metrics.

6

Recovering from Production Failures

We live in an imperfect world. We first see bugs escape into our production environment. Then, we may find as we start moving to DevOps practices, there are gaps in our understanding that affect how we deliver in our production environment. As we get those fixed, we may encounter other problems that are outside our control. What can we possibly do?

In this chapter, we will examine mitigating and dealing with failures that happen in production environments. We will look at the following topics:

- The costs of errors in production environments
- Preventing as many errors as we can
- Practicing for failures using chaos engineering
- Resolving incidents in production with an incident management process
- Looking at fixing production failures by rolling back or fixing forward

Learning from failure

Production failures can happen at any time in the product development process, from the first deployment to supporting a mature product. When these production failures happen, depending on the impact, they may adversely affect the value the customer sees and potentially ruin a business's reputation.

Often, we don't see the lessons offered by these production failures until the failures happen or afterward when reading about such failures happening to another organization (or even a competitor!).

We will examine a sample of such famous production failures, hoping to glean lessons through the benefit of hindsight. The following examples include the following:

- The rollout of `healthcare.gov` in 2013
- The Atlassian cloud outage in 2022

Other lessons will come from other sections in this chapter.

healthcare.gov (2013)

In 2010, the *Patient Protection and Affordable Care Act* became law in the USA. A key part of this law, colloquially known as *Obamacare*, was the use of a website portal called `healthcare.gov` that allowed individuals to find and enroll in an affordable health insurance plan through multiple marketplaces. The portal was required to go online on October 1, 2013.

`healthcare.gov` was released on that date and immediately encountered problems. The initial demand upon launch, 250,000, was five times what was expected and caused the website to go down in the first 2 hours. By the end of the first day, a total of six users had successfully submitted applications to enroll in a health insurance plan.

A massive troubleshooting effort ensued, eventually allowing the website to handle 35,000 concurrent users, and registering 1.2 million users to a health insurance plan before the enrollment period closed in December 2013.

One of a number of reports looking at the `healthcare.gov` debacle was written by Dr. Gwanhoo Lee and Justin Brumer. In the report (seen at `https://www.businessofgovernment.org/sites/default/files/Viewpoints%20Dr%20Gwanhoo%20Lee.pdf`), they specified the challenges of such a massive undertaking, which included the following:

- A complex IT system in a limited period of time
- Policy problems that created uncertainty in their implementation
- High-risk contracting with limited timeframes
- A lack of leadership

Lee and Brumer also identified a series of missteps from the early stages of the design and development of the portal that would serve to doom the project. These included the following:

- A lack of alignment between the government policy and the technical implementation of the portal
- Inadequate requirements analysis
- Failure to identify and mitigate risks
- Lack of leadership
- Inattention to bad news
- Rigid organizational culture
- Inattention to project management fundamentals

Fixing healthcare.gov

One of the efforts that came about after the disastrous initial launch of `healthcare.gov` was the Tech Surge, a takeover by software developers from Silicon Valley, who refactored major parts of the `healthcare.gov` website. The teams in the Tech Surge operated as small teams with a start-up mentality and were accustomed to the use of Agile practices that brought close collaboration, DevOps tools such as New Relic, and cloud infrastructures.

One of the offshoots from the Tech Surge was a small group of coders led by Loren Yu and Kalvin Wange, known as **Marketplace Lite** (**MPL**). They started as part of the Tech Surge and worked with existing teams at the **Centers for Medicare and Medicaid Services** (**CMS**), showing them new practices, such as collaborating over chat instead of emails, as they rewrote the parts of the website to log in and register for a new plan.

MPL continued to work on `healthcare.gov` as many of the contracts ran out for other developers of the Tech Surge. It continued to work alongside CMS to improve systems testing and deliver fixes incrementally, as demonstrated in a **Government Accountability Office** (**GAO**) report at the time. The efforts were starting to bear serious fruit. One of the rewritten parts that MPL worked on, *App 2.0*, the tool to register for new healthcare insurance, was *soft-launched* for only call centers but became so successful that it was the main tool for registering new applications for those who had a simple medical history.

The work of MPL and the Tech Surge and the success of subsequent rollouts of `healthcare.gov` in further enrollment periods provided a proving ground for Agile and DevOps mindset and practices. Agencies such as 18F and the United States Digital Service took up the baton and began the job of coaching other federal agencies to apply Agile and DevOps to technology projects.

Atlassian cloud outage (2022)

On April 5, 2022, 775 of Atlassian's more than 200,000 total customer organizations lost access to their Atlassian cloud sites, which served applications such as Jira Service Management, Confluence, Statuspage, and Opsgenie. Many of these customers remained without access for up to 14 days until service to the remaining sites was restored on April 18.

The root cause of the site outage was traced to a script used by Atlassian to delete old instances of Insight, a popular standalone add-on to Jira that was acquired by Atlassian in 2021. Insight eventually became bundled into Jira Service Management, but traces of the legacy app remained and needed to be removed.

A miscommunication occurred where the team responsible for running the script was given a list of site IDs as input for the script instead of a list of Insight instance IDs. What followed was the immediate deletion of sites.

Atlassian's cloud architecture is composed of multi-tenant services that handle applications for more than one customer. A blanket restoration of services would have affected those customers whose sites

weren't deleted. Atlassian had the knowledge of how to restore a single site but had never anticipated needing to restore the number of sites they currently faced. Atlassian began restoring customer sites. Restoration of a bunch of sites would take 48 hours. A manual restoration effort would have taken weeks for all the missing sites; clearly, Atlassian needed to automate.

The automation effort was designed as Atlassian figured out a method to restore multiple sites at once. The automation was run starting on April 9 and accomplished the restoration of a site in 12 hours. Roughly 47% of the total sites were restored using automation by the time the last site was restored on April 18.

The bigger issue was communicating with the affected customers. Atlassian was first made aware of the incident through a customer support ticket, but it was not immediately aware of the total number of affected customers. This is because deleting the sites also deleted metadata containing customer information that would be used by the customer to create support tickets. Recovering the lost customer metadata was important for customer notification.

The inability of Atlassian to directly contact affected customers made a major communication problem even greater. Those customers not contacted by Atlassian began reaching out on social media sites such as Twitter and Reddit to get news of what had happened. A general tweet from Atlassian made its way on April 7. A blog article from Atlassian's CTO, Sri Viswanath, with more detailed explanations came on April 12. After the incident was solved, a post-incident review report was made generally available on April 29.

Lessons from the Atlassian outage

The Atlassian outage provided challenges both from a technical and a customer service perspective. The post-incident review outlined four major learning point that Atlassian must improve upon to prevent similar outages from occurring. These learnings included the following:

- Changing the process of deleting production data to *soft deletes*, where it is easier to recover and will be removed only after a certain period of time has elapsed.
- Looking at specific processes for multiple-site, multiple-product data for a larger set of affected customers.
- Considering incident management for large-scale events. Atlassian had processes in place for one customer's site. It now needed to consider large-scale incidents, affecting a large number of customers.
- Improve customer communication during an incident. Atlassian started communication when it had a grasp of the cause and the efforts needed to correct the incident. This delay in communication allowed the incident to play out on social media.

But there are broader lessons for us as well. Failures in production can happen from the first deployment to any point in the life cycle of a product. Failures can happen to companies starting out with Agile and DevOps or companies such as Atlassian that have succeeded in using Agile and DevOps. With all

of these companies, the key to handling production failures includes ensuring that the process roots out as many failures as possible, practicing for failures to determine the best process, and setting up a proper process when failure does occur.

To aid us in this, we turn to a growing discipline called **Site Reliability Engineering** (**SRE**). This discipline was created by Ben Treynor Sloss at Google to initially apply software development methods to system administration. Originally seen as a hybrid approach utilizing methods used in development groups for traditional system administration operations, SRE has grown to its own branch within DevOps to ensure continued reliable systems operations after automated deployment has occurred.

The first step is planning and prevention. Let's start looking at the safeguards used by SRE for preventing production failures.

Prevention – pulling the Andon Cord

The **Andon Cord** holds a special place in Lean thinking. As part of the Toyota Production System, if you suspected a problem with a car on the assembly line, you would pull the cord that ran around on the assembly line, and it would stop the line. People would come to the spot where the Andon Cord was pulled to see the defect and determine, first, how to fix the defect, and second, what steps would be needed to prevent the defect from occurring in the future.

Taiichi Ohno, the creator of the Toyota Production System, uses the Andon Cord to practice *jidoka*, empowering anyone to stop work to examine and implement continuous improvement.

For site reliability engineers, the following ideas and principles are used as a way of implementing the Andon Cord and ensuring continuous improvement:

- Planning for risk tolerance by looking at **service-level indicators** (**SLIs**), **service-level objectives** (**SLOs**), and error budgets
- Enforcing release standards through release engineering
- Collaborating on product launches with launch coordination engineering

Let's examine these ideas in closer detail.

SLIs, SLOs, and error budgets

Many people are familiar with the concept of **service-level agreements** (**SLAs**), where if the service does not meet a threshold for service availability or responsiveness, the vendor is then liable to pay for that agreed-upon level of performance, typically in the form of credits.

If we take a look at the goal or threshold that an SLA is expected to achieve or maintain, that is called an SLO. Generally, there are three parts to an SLO:

1. The quality/component to measure

2. The measurement periods
3. The required threshold the quality must meet, typically written as a desired value or range of values

That quality or component to measure is known as an SLI. Common SLIs that are typically used include the following:

- Latency
- Throughput
- Availability
- Error rate

For every SLO, the time inside the measurement period where the threshold is not met is known as the error budget. Closely monitoring the error budget allows SREs to gauge whether the risk is acceptable to roll out a new release. If the error budget is almost exhausted, the SRE may decide that the focus should change from feature development to more technical work, such as enablers, which would enhance resiliency and reliability.

Teams generally want to understand an error budget in terms of allowable time. The following table may provide guidance on the maximum allowable error on a monthly and an annual basis:

SLO Percentage	Monthly Allowed Error Budget	Annual Allowed Error Budget
99% (1% margin of error)	7 hours, 18 minutes	87 hours, 39 minutes
99.5 (0.5% margin of error)	3 hours, 39 minutes	43 hours, 49 minutes, 45 seconds
99.9% (0.1% margin of error)	43 minutes, 50 seconds	8 hours, 45 minutes, 57 seconds
99.95% (0.05% margin of error	21 minutes, 54 seconds	4 hours, 22 minutes, 48 seconds
99.99% (0.01% margin of error)	4 minutes, 23 seconds	52 minutes, 35 seconds

Table 6.1 – Error budgets in terms of allowable monthly and annual time

The journey to implementing SLOs often begins with an evaluation of the product or service. From there, look at the components or microservices that make up the product: which parts of these, if not available, would contribute to unhappiness for the customer?

After the discovery of components critical to customer happiness, choose those measurements (SLIs) to capture and set up your goals (SLOs), making sure that the measurements give true indicators of potential problems and that the goals are realistic and attainable (100% of any measurement is not attainable). Start with a small set of SLOs. Communicate these SLOs to your customer so that they understand the role SLOs will play in making a better product and the expectations.

SLIs, SLOs, and error budgets should be documented as policy, but the policy is meant to change and adjust. After some time, reevaluate the SLIs, SLOs, and error budget to see whether these measurements are effective, and revise the SLIs and SLOs as needed.

Release engineering

To ensure that SLOs are maintained, site reliability engineers need to ensure that anything that is released to a customer is reliable and may not contribute to an outage. To that end, they work with software engineers to make sure releases are low-risk.

Google details this collaboration as release engineering. This aspect of SRE is guided by the following principles:

- Self-service
- High velocity
- Hermetic builds
- Policy/procedure enforcement

Let's look at these four parts of the release engineering philosophy now.

Allowing release autonomy through a self-service model

For agility to prosper, the teams working must be independent and self-managing. Release engineering processes allow the teams to decide their own release cadence and when to actually release. This ability for teams to release when and how often they need to is aided by automation.

Aiming for high velocity

If teams choose to release more often, they are often doing so with smaller batches of changes of highly tested code. More frequent releases of small changes reduce the risk of outages. This is especially helpful if you have a large error budget.

Ensuring hermetic builds

We want consistency and repeatability in our build-and-release process. The build output should be identical no matter who creates it. This means that versions of dependent artifacts and tools such as libraries and compilers are standardized from test to production.

Of course, if problems occur out in production, a useful tactic for troubleshooting is known as *cherry-picking*, where the team starts with the last-known *good* production version, retrieved from version control, and inserts each change one by one until the problem is discovered. Strong version control procedures ensure that builds are hermetic and allow for cherry-picking.

Having strongly enforced policies and procedures

Automated release processes that produce hermetic builds require standards of access control to ensure that builds are created on the correct build machines using the correct sources. The key is to avoid adding local edits or dependencies and only use verified code kept in version control.

These four principles we have discussed are really applied when looking at the automation that handles the following parts of the release process:

- **Continuous integration/continuous deployment** (**CI/CD**)
- Configuration management

We first saw these parts as automated implementations of the CI/CD pipeline in *Chapter 3*, *Automation for Efficiency and Quality*. Now, let's see how we tie the process into the automation.

CI/CD

The release process begins with a commit made to version control. This starts the build process with different tests automatically executed depending on the branch. Release branches run the unit tests as well as applicable system and functional tests.

When the tests pass, the build is labeled so that there is an audit trail of the build date, dependencies, target environment, and revision number.

Configuration management

The files used by the configuration management tools are kept in version control. Versions of the configuration files are recorded with release versions as part of the audit trail so that we know which version of the configuration files is associated with which versions of the release.

Launch coordination engineering

Launching a new product or feature to customers may have greater expectations than iterative releases of existing products. To facilitate the release of new services, Google created a special consulting function within SRE called **launch coordination engineering** (**LCE**).

The engineers in LCE perform a number of functions, all intended to ensure a smooth launch process. These functions include the following:

- Auditing the product or service to ensure reliability
- Coordinating between multiple teams involved in the launch
- Ensuring completion of tasks related to technical aspects of the launch
- Signing off that a launch is *safe*
- Training developers on integration with the new service

To aid launch coordination engineers in ensuring a smooth launch, a launch checklist is created. Depending on the product, engineers tailor the checklist, adding or removing the following checklist items:

- Shared architecture and dependencies
- Integration
- Capacity planning
- Possible failure modes
- Client behavior
- Processes/automation
- Development process
- External dependencies
- Rollout planning

We have seen techniques and processes SREs use to ensure that the product launch or code release is ready. We've seen the tolerance for failure through SLIs, SLOs, and error budgets. But do we know whether the SREs are ready if an outage occurs?

One way of determining is by simulating a failure and seeing the reaction. This is another tool that SREs use, called chaos engineering. Let's take a look at what's involved.

Preparation – chaos engineering

On September 20, 2015, **Amazon Web Services** (**AWS**) experienced an outage with more than 20 services out of its data centers in the US-EAST-1 region. These services affected applications from major companies such as Tinder, Airbnb, and IMDb, as well as Amazon's own services such as Alexa.

One of AWS's customers that was able to avoid problems during the outage and remain fully operational was Netflix, the streaming service. It was able to do so because it created a series of tools that it called the *Simian Army*, discussed in this blog article at `https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116`, which simulated potential problems with AWS so that Netflix engineers could design ways to make their system more resilient.

Over several AWS outages, the Simian Army proved its worth, allowing Netflix to continue providing service. Soon, other companies such as Google started wanting to apply the same techniques. This groundswell of support led to the creation of the discipline of chaos engineering.

Let's take a closer look at the following aspects of chaos engineering:

- Principles
- Experiments

Chaos engineering principles

The key to chaos engineering is experimentation in production environments. The idea of performing reliability experiments in production does seem to be laden with risk. This risk, though, is tempered by your confidence in the resiliency of the system.

To guide confidence, chaos engineering starts with the following principles:

- Build your hypothesis around the steady-state behavior of your production environment
- Create variables that simulate real-world events
- Run the experiment in your production environment
- Automate the experiment
- Minimize the experiment's fallout

Let's discuss these principles in detail.

Basing experiments around steady-state behavior

In devising our experiments, we really want to focus on the system outputs rather than the individual components of the system. These outputs form the basis of how our environment behaves in a steady state. The focus in chaos engineering is on the verification of the behavior and not on the validation of individual components.

Mature organizations that look at chaos engineering as a key part of SRE know that this steady-state behavior typically forms the basis for SLOs.

Creating variables that simulate real-world events

Given the known steady-state behavior, we consider *what-if* scenarios that happen in the real world. Each event you consider then becomes a variable.

One of the famous tools in Netflix's Simian Army, *Chaos Monkey*, was based on the event that a virtual server node in AWS would become unavailable. So, it tested for that condition only.

Running the experiment in production

Running the experiment in a staging or *production-like* environment is beneficial, but at some point, you need to run the experiment with its variable in the production environment to see the effects on real-world processing of real traffic.

At Netflix, Chaos Monkey was run every day in production. It would look at every cluster in operation and randomly deactivate one of the nodes.

Automating the experiment

The benefits of learning from chaos engineering experiments are only apparent when experiments are run consistently and frequently. To achieve this, automating the experiment is necessary.

Chaos Monkey was not initially popular with Netflix engineers when initially rolled out. The idea that every day, this program would intentionally cause errors in production did not sit well with them, but it did consistently raise the problem that instances could vanish. With this problem, engineers had a mandate to find solutions and make the system more resilient.

Minimizing the fallout

Because you are running your experiment in the production environment, your customers who are also using that environment may be affected. Your job is to make sure the fallout from running the experiment is minimized.

Chaos Monkey was run once per day, but only during business hours. This would be to ensure that if any ill effects to production were discovered, it would be while most of the engineers were present and not off-hours such as at 3 A.M. when there would only be a skeleton crew.

With these principles in place, let's apply them and look at creating experiments.

Running experiments in chaos engineering

Experimenting in production involves planning and developing a process. The goal of the experiment is to find those weak areas that could be more resilient to ensure SLOs are kept. The goal is not to *break the system*.

In *Chaos Engineering: System Resiliency in Practice*, Richard Crowley writes a chapter dealing with creating the *Disasterpiece Theater* process for Slack. He outlines the following steps for the process:

1. Make sure a *safety net* is in place.
2. Prepare for the exercise.
3. Run the exercise.
4. Debrief the results of the exercise.

Let's examine the details of each step now.

Ensuring the environment is ready for chaos

The goal of chaos engineering is to find weaknesses in resiliency, not to disable the environment. If the existing environment has no fault tolerance, there's no point in running experiments.

Make sure there is spare capacity for services. That spare capacity should be easy to bring online.

Once the spare capacity and resources have been identified and allocated, have a plan to allow for the removal of malfunctioning resources and automatic replacement with the spare capacity.

Preparing for the exercise

For Crowley, an exercise starts with a worry: Which critical component or service will fail, impacting resiliency? This becomes the basis for the exercise.

Crowley then takes this basis and works on expanding this to an exercise to run in development, staging, and production environments. He sets up a checklist, making sure each of the following items is fulfilled for the exercise:

- Describe the server or service that is to fail, including the failure mode, and how to simulate the failure.
- Identify the server or service in development and production environments, and confirm the method to simulate is possible in the development environment.
- Identify how the failure should be detected. Will an alert be produced that will show up on dashboards? Will logs be produced? If you cannot imagine how it will be detected, it may still be worth running the exercise to determine a way to detect the failure.
- Identify redundancies and mitigations that should eliminate or reduce the impact of the failure. Also, identify any runbooks that are run if the failure should occur.
- Identify the relevant people that should be invited to contribute their knowledge to the exercise. These people may also be the first responders when the exercise happens.

Preparation culminates in a meeting with the relevant people to work out the necessary logistics of the exercise. When all the preparations are set, it's time to run the exercise.

Running the exercise

The exercise should be well publicized to all involved people before it is executed. After all, they will be participating in the exercise, with the goal of creating a more resilient environment.

Crowley executes the exercise with the following checklist:

1. Make sure everyone is aware the exercise is being recorded. Make a recording if everyone allows it.
2. Review the plan created in the preparation step. Make adjustments as necessary.
3. Announce the beginning of the exercise in the development environment.
4. Create a failure in the development environment. Note the timestamp.
5. See whether alerts and logs are created for the failure. Note the timestamp.
6. If there are automated recovery steps, give them time to execute.

7. If runbooks are being used, follow them to resolve the failure in the development environment. Note the timestamp and whether any deviations from the runbooks occurred.
8. Have a go/no-go decision to duplicate this failure in the production environment.
9. Announce the beginning of the exercise in the production environment.
10. Create a failure in the production environment. Note the timestamp.
11. See whether alerts and logs are created for the failure. Note the timestamp.
12. If there are automated recovery steps, give them time to execute.
13. If runbooks are being used, follow them to resolve the failure in the production environment. Note the timestamp and whether any deviations from the runbooks occurred.
14. Announce the all-clear and conclusion of the exercise.
15. Perform a debrief.
16. Distribute the recording if one was made.

With the exercise complete, a key part of the exercise begins with the debrief, after the all-clear is announced. Let's examine how to create a learning debrief.

Debriefing for learning

Crowley recommends performing a debrief while the experience of the exercise is still fresh in everyone's minds. During the debrief, only the facts are presented, with a summary of how well the system did (or didn't) perform.

Crowley has the following starter questions to help display what was learned. These are offered as a template:

- What was the time until detection? What was the time until recovery?
- Did the end users notice when we ran the exercise in production? How do we know? Are there solutions to make that answer *no*?
- Which recovery steps could be automated?
- Where were our blind spots?
- What changes to our dashboards and runbooks have to be made?
- Where do we need more practice?
- What would our on-call engineers do if this happened unexpectedly?

The outcome of the exercise and the answers in the debrief can form recommendations for the next steps to add resiliency to the system. The exercise can be repeated to ensure that the system correctly identifies and resolves the failure.

Disasterpiece Theater can be an effective framework for performing your chaos engineering exercises. The flexibility of the exercise is dependent upon how resilient your system is already.

Even with regular chaos engineering exercises, bad things can still happen in your production environments that will affect your customers. Let's look at ways to solve these production issues with incident management.

Problem-solving – enabling recovery

For SREs, a solid incident management process is important when things go wrong in production. A good incident management process allows you to follow these necessary goals, commonly referred to as *the three Cs*:

- **Coordinate** the response
- **Communicate** between the incident participants, others in the organization, and interested parties in the outside world
- Maintain **control** over the incident response

Google identified necessary elements to their incident command system in the *Managing Incidents* chapter written by Andrew Stribblehill in *Site Reliability Engineering: How Google Runs Production Systems*. These elements include the following:

- Clearly defined incident management roles
- A (virtual or physical) command post
- A living incident state document
- Clear handoffs to others

Let's look at these elements in detail.

Incident management roles

Upon recognition that what you are facing is truly an incident, a team should be assembled to work on the problem and share information until the incident is resolved. The team will have roles so that coordination is properly maintained. Let's look at these roles in detail.

The incident commander

The incident commander may start as the person who originally reports the incident. The incident commander embodies the *three Cs* by delegating the necessary roles to others. Any role not delegated is assumed to belong to the incident commander.

The incident commander will work with the other responders to resolve the incident. If there are any roadblocks, the incident commander will facilitate their removal.

Operations lead

The operations lead will work together with the incident commander. They will run any needed tools to resolve the incident. The operations lead is the only person allowed to make changes to the system.

Communications lead

The communications lead is the public face of the incident and its response. They are responsible for communication with outside groups and stakeholders. They may also ensure that the incident document is kept up to date.

Incident planning/logistics

Planning and logistics will work with the operations people by working on longer-term issues of the incident such as arranging for handoffs between roles, ordering meals, and entering tickets in the bug tracking system. They will also track how the system has diverged from the norm so that it can be returned to normal when the incident is resolved.

The incident command post

A *war room* is needed for all members of the incident response team to convene and collaborate on the solution. This place should be where outside parties can meet with the incident commander and other members of the incident response team.

Because of distributed development, these command posts are typically virtual as opposed to a physical room. **Internet Relay Chat** (**IRC**) chat rooms or Slack channels can serve as the medium for gathering in one *spot*.

The incident state document

The incident commander's main responsibility is to record all activity and information related to the incident in the incident state document. This is a living document, meant to be frequently updated. A wiki may suffice, but that typically allows only one person to edit it at a time.

Suitable alternatives may be a Confluence page or a document shared in a public Google Drive or Microsoft SharePoint folder.

Google maintains a template for an incident state document that can be used as a starting point.

Setting up clear handoffs

As we saw with Atlassian's incident earlier in this chapter, incidents can stretch over several days or even weeks. So, the handoff of roles is essential, particularly for the incident commander. Communication must be clear to everyone that a handoff has taken place to minimize any confusion.

While in an incident, some actions that may help move toward a solution include rolling back or *rolling forward*. These may work if the root cause is diagnosed as a new change recently made. We'll look at these alternatives in our next section.

Perseverance – rolling back or fixing forward

If the reason for the production failure is a new change, a quick resolution may involve reverting to the state of the system before the change, or if a fix is found, immediately running it through the CI/CD pipeline to immediately deploy.

Some of the methods for rolling back or rolling forward a fix include the following ones. Let's examine them in detail.

Rolling back with blue/green deployment

A blue/green deployment makes use of two production environments: one live and the other on standby. The live environment is the one that customers use, while the standby environment is there as a backup. The change is made on the standby environment and then the standby environment is made live. You can see an illustration of this type of deployment here:

Figure 6.1 – Blue/green deployment: environment switch

As the preceding diagram indicates, both environments are still present, but only one has access to customer traffic. The arrangement remains this way until changes are deployed into the standby environment or a rollback becomes necessary, as illustrated in the following diagram:

Figure 6.2 – Blue/green deployment: rollback

A blue/green deployment works well when the environment is stateless—that is, there is no need to consider the state of data. Complications arise when the data's state has to be considered in artifacts such as databases or volatile storage.

Rolling back with feature flags

We saw in *Chapter 3*, *Automation for Efficiency and Quality*, that feature flags allowed the propagation of changes to deployment without the change visible until the flag was *toggled on*. In this same way, if a new feature is the root cause of a production failure, the flag can be *toggled off* until the new feature can be fixed, as illustrated in the following diagram:

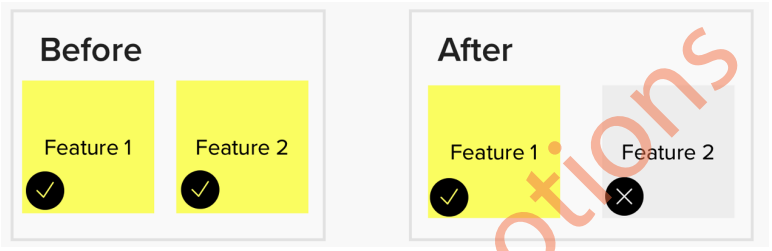


Figure 6.3 – Rollback with a feature flag

Rolling back by using feature flags allows for a quick change to previous behavior without extensive changes to the source code or configuration.

Rolling forward using the CI/CD pipeline

Rolling forward or *fixing forward* is the method of resolving an incident by developing a fix for the error, allowing it to go through the CI/CD pipeline so that it can be deployed into production. It can be an effective way to resolve the incident, especially if the change is small.

Fixing forward should be viewed as a *last resort*. If fixing forward is the only viable option, your product's architecture may be too tightly coupled with dependent components. For example, if the new release depended on a change to the database schema and customer data was already stored in the new tables before the production failure was discovered, there may be no rollback without losing the customer data.

Changes that are intended to be released in a roll-forward solution should undergo the same process as normal releases. *Quick fixes* that may not follow the entire process, which may not have the same scrutiny and test coverage, may increase the technical debt by introducing errors in other parts of the system.

Summary

We examined in this chapter what happens when things go wrong in production. We began our chapter by looking at two incidents: the initial release of `healthcare.gov` in 2013 and the Atlassian cloud outage in 2022. We learned from both incidents the importance of prevention and planning for future incidents.

We then explored methods of preparation by looking at important parts of the discipline of SRE. SRE begins this process by setting the SLIs and SLOs so that we have an idea of the tolerance of risk through the error budget. SRE also looks at the process of releasing new changes and launching new products.

We looked at practicing for disaster through the discipline of chaos engineering. We understood the principles behind the discipline and how to create experiments through the Disasterpiece Theater process.

Ultimately, even with adequate preparation, production failures will still happen. We looked at the key parts of Google's incident management process for incident management and techniques for resolving incidents, such as rolling back or fixing forward.

With this, we have completed *Part 1: Approach – A Look at SAFe® and DevOps through CALMR*. We will now look at a key activity of DevOps, value stream management, in *Part 2: Implement – Moving Toward Value Streams*.

Questions

Test your knowledge of the concepts in this chapter by answering these questions.

1. What is an example of an SLI?

   A. Velocity
   B. Availability
   C. Cycle time
   D. Scalability

2. If your organization sets up an SLO of 99% availability on a monthly basis, what is your error budget if the acceptable downtime is 7.2 hours/month?

   A. 0.072 hours/month
   B. 0.72 hours/month
   C. 7.2 hours/month
   D. 72 hours/month

3. Which is *NOT* a principle of release engineering?

   A. Self-service model

B. High velocity
C. Dependent builds
D. Enforcement of policy/procedures

4. Which company created Chaos Monkey and the Simian Army?

A. Netflix
B. Google
C. Amazon
D. Apple

5. Which of these are chaos engineering principles (choose two)?

A. Decentralized decision making
B. Organize around value
C. Minimize the experiment's fallout
D. Run the experiment in production
E. Apply systems thinking

6. Which role in Google's incident command system is the primary author of the incident state document?

A. Operations lead
B. Incident commander
C. Communications lead
D. Planning/logistics

Further reading

Here are some resources for you to explore this topic further:

- `https://www.businessofgovernment.org/sites/default/files/Viewpoints%20Dr%20Gwanhoo%20Lee.pdf`—*Managing Mission-Critical Government Software Projects: Lessons Learned from the Healthcare.gov Project* by *Dr. Gwanhoo Lee* and *Justin Brumer*. An interesting look at the root causes of the healthcare.gov debacle.
- `https://www.theatlantic.com/technology/archive/2015/07/the-secret-startup-saved-healthcare-gov-the-worst-website-in-america/397784/`—A writeup of the efforts of the Tech Surge and MPL in particular.

- https://www.gao.gov/assets/gao-15-238.pdf—A report from the GAO describing the initial problems with the healthcare.gov launch and progress toward needed fixes.
- https://oig.hhs.gov/oei/reports/oei-06-14-00350.pdf—A case study of the initial launch of healthcare.gov and the changes brought about by Tech Surge that allowed success in subsequent launches. Created by the **Office of the Inspector General** (**OIG**) of **Health and Human Services** (**HHS**).
- https://www.atlassian.com/engineering/post-incident-review-april-2022-outage—Post-incident review of the Atlassian cloud outage from the current CTO, Sri Viswanath.
- *Site Reliability Engineering: How Google Runs Production Systems* edited by *Betsy Beyer*, *Chris Jones*, *Jennifer Petoff*, and *Niall Richard Murphy*—Reference book for SRE detailing principles and essays about SRE topics.
- https://medium.com/kudos-engineering/managing-reliability-with-slos-and-error-budgets-37346665abf6—A writeup that describes the relationship between SLIs, SLOs, and error budgets.
- https://www.techrepublic.com/article/aws-outage-how-netflix-weathered-the-storm-by-preparing-for-the-worst/—Article that describes how Netflix avoided the effects of an AWS outage in September 2015.
- https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116—Blog article from Netflix explaining the Simian Army.
- https://principlesofchaos.org—Repository for chaos engineering principles.
- *Chaos Engineering: System Resiliency in Practice* by *Casey Rosenthal* and *Nora Jones*—Reference for chaos engineering, describing principles and essays on creating experiments.
- *The Site Reliability Workbook: Practical Ways to Implement SRE* edited by *Betsy Beyer*, *Niall Richard Murphy*, *David K. Rensin*, *Kent Kawahara*, and *Stephen Thorne*—Reference book for implementing SRE practices.
- https://www.linkedin.com/pulse/service-recovery-rolling-back-vs-forward-fixing-mohamed-el-geish/—Blog article by Mohamed El-Geish that talks about recovery strategies, rolling back, and fixing forward.

Part 2: Implement – Moving Toward Value Streams

In the book *The Phoenix Project: A Novel about IT, DevOps, and Helping your Business Win* by Gene Kim, Kevin Behr, and George Spafford, we are introduced to the Three Ways. These ways outline how we can shift toward DevOps and CALMR.

The First Way emphasizes working toward a flow. To do this, we will organize and structure along value streams to visualize the steps and people that perform those steps. Working as a value stream allows us to optimize the flow. We will see how to establish our value streams in *Chapter 7*.

After the First Way, we come to the Second Way, which emphasizes the amplification of feedback loops. To find our feedback, we look at the metrics that can be used to evaluate our value streams in *Chapter 8*.

Finally, we come to the Third Way. The Third Way emphasizes moving to continuous experimentation and learning. *Chapter 9* will discuss what makes an organization that is continuously learning and methods for continuous learning to improve your value streams.

This part strives to set up an easy way to achieve Value Stream Management, a key practice in DevOps.

This part of the book comprises the following chapters:

- *Chapter 7*, *Mapping Your Value Streams*
- *Chapter 8*, *Measuring Value Stream Performance*
- *Chapter 9*, *Moving to the Future with Continuous Learning*

7

Mapping Your Value Streams

Value streams are an important part of the SAFe®. We saw evidence of this in *Chapter 2*, *Culture of Shared Responsibility*, when we looked at the SAFe Lean-Agile Principles and came to *Principle #10: Organize Around Value*. In that same way, value streams have a key role to play in DevOps. In *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*, the authors Gene Kim, George Spafford, and Kevin Behr introduced the concept of the Three Ways. The First Way is to use a value stream to establish flow and this is precisely what we will look at in this chapter.

We will examine how to discover your organization's value stream and find future optimizations. In all, we will look at the following activities:

- Aligning the organization's mindset to value streams
- Setting the context for development value streams
- Mapping the development value stream
- Finding areas for improvement

Let's take a look at preparing the organization to think of operations in terms of value streams.

Aligning the organization's mindset

A focus on value streams will require major changes to an organization, which may culminate in a change of culture. These efforts at transformation are often the beginning of a never-ending journey of continuous improvement.

We'll first evaluate one example of such a transformation: the adoption of SAFe through the implementation roadmap. In examining the roadmap, we'll see how identifying value streams plays a key role.

Another example of a guide to transformation is the eight steps that form a **Value Stream management** (**VSM**) initiative, as identified in *Driving DevOps with Value Stream Management: Improve IT Value Stream Delivery with a Proven VSM Methodology to Compete in the Digital Economy* by Cecil "Gary" Rupp. The VSM initiative, as described in this book, is the reference method for the VSM Consortium, a group of individuals and organizations looking to advance methods to improve VSM.

We will see that there is a significant overlap between the two transformation guides. Because of this, it is possible to have a VSM initiative inside of a SAFe transformation, and the two methods complement each other. With that in mind, let's begin our examination of value streams with a look at SAFe's implementation roadmap.

Moving to SAFe

The implementation roadmap is the main pattern that an organization can use to adopt SAFe. The roadmap starts with setting up a coalition of change agents, training them on the new ways of working, and then setting this up through value streams. The roadmap can be customized so that it fits the organization.

The steps of the implementation roadmap are as follows:

1. Reach the tipping point to adopt SAFe.
2. Train the champions and change agents.
3. Train the executives and leaders on the new ways of working.
4. Create a Lean-Agile center of excellence for the change agents.
5. Identify the value streams.
6. Create an implementation plan.
7. Prepare to launch the first **Agile release train** (**ART**).
8. Train the ART and launch.
9. Coach the ART on execution.
10. Launch additional value streams and ARTs.
11. Extend ART guidance to a portfolio.
12. Accelerate and continue through inspection and adaptation.

We can see that this roadmap aligns with the Kotter Change Model, previously discussed in *Chapter 2*, *Culture of Shared Responsibility*, specifically as identified in the following table.

Kotter Change Model step	**Applicable implementation roadmap step(s)**
Create a sense of urgency	Reach the tipping point to adopt SAFe
Guide a powerful coalition	Train the champions and change agents, train the executives and leaders on the new ways of working, create a Lean-Agile center of excellence
Create a vision	Identify the value streams, create an implementation plan
Communicate that vision	Create an implementation plan, prepare to launch the first ART

Empower others to enact the vision	Prepare to launch the first ART, train the ART and launch
Celebrate the short-term wins	Coach the ART on execution
Consolidate wins to create more change	Launch additional value streams and ARTs, extend ART guidance to a portfolio
Anchor new changes in the culture	Launch additional value streams and ARTs, extend ART guidance to a portfolio, accelerate and continue through inspection and adaptation

Table 7.1 — Mapping of the Kotter Change Model to the SAFe implementation roadmap

We can see that identifying the value stream(s) is an important part of launching the first ART and subsequent ARTs. With this in mind, we can look at the steps of launching a VSM initiative to assist in identifying value streams.

Moving to value streams

The VSM methodology is presented in *Driving DevOps with Value Stream Management: Improving IT Value Stream Delivery with a Proven VSM Methodology to Compete in the Digital Economy* by Cecil "Gary" Rupp. In this book, the author adapts the Improvement Kata model from Lean thinking to create and maintain a value stream or multiple value streams.

The VSM methodology is outlined in the following steps:

1. Committing to Lean
2. Choosing the value stream
3. Learning about Lean
4. Mapping the current state of the value stream
5. Mapping the ideal future state of the value stream
6. Setting up an improvement (Kaizen) plan to get to the future state
7. Implementing the improvement plan
8. Repeating the process

Now that we understand the overall strategy to map value streams, let's take a closer look at the process of identifying value streams in our next section.

Setting the context for development value streams

The work done by a development value stream is applied to a solution that has a role in the greater context of the journey that the customer takes with an organization to obtain value. This greater context is captured in the operational value stream that the organization may have.

We took an initial look at the differences between *Operational* value streams, which detail the customer's needs and how a company's solutions can fit those needs, and *Development* value streams, which show the steps needed to develop and maintain a solution, in *Chapter 2*, *Culture of Shared Responsibility*. In this section, we will demonstrate how to create an operational value stream with the help of useful tools, such as Gemba walks. The last part of this is finding the solutions that the Operational value stream uses. These solutions become the basis for Development value streams.

Preparing for value stream identification

Mapping both the operational and development value streams can be an extensive effort, one that can easily be frustrating when looking at the complexities of the organization. Before engaging in this endeavor, some preparation is necessary.

Mapping value streams is often done through workshops held with different members of the organization in attendance. Before facilitating such a workshop, you may want to make sure the following are determined by the organization:

- The scope of the value stream
- The members of the team that will meet to identify the value streams
- An understanding of the customer's point of view through a Gemba walk

Let's examine each of these in closer detail.

Determining the scope

Organizations may be small or immense, based on the history or number of products or solutions that an organization designs, produces, and maintains. Before discovering the organizational and development value streams, the organization should set the proper scope for discovery.

Karen Martin and Mike Osterling lay out the scope as a value stream charter in their book *Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation*. In the book, they collect the following items for the value stream charter:

- value streams.
- **Specific conditions**: It may be useful to only allow a limited number of scenarios. If this is the case, you may want to set up only a small number of conditions that bring about the desired scenarios.
- **Trigger**: The event that starts the activity in the value stream.
- **Demand rate**: This is how often the trigger occurs.
- The first step and last step in a stream.
- The boundaries/limitations.

- **Improvement time frame**: This may help determine what possible future state improvements should be made first.
- Current state problems and needs.
- Measurable target conditions.
- The value to the customer and value to the business.
- The accountable team members.

Let's take a look at the members and roles needed for the mapping in the next section.

Creating the team

A value stream mapping exercise may take several days. During that time, a number of people will be called to provide their expertise and fill out the items required by the value stream charter for the operational and development value streams.

The following roles are laid out in *Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation*:

- **Executive sponsor**: This may be a C-level leader who is ultimately accountable for the results. They will oversee the overall process and keep track of the progress of the value stream toward the future state.
- **Value Stream champion**: This is someone accountable for the performance of the value stream. They will be playing a key role in value stream mapping.
- **Facilitator**: This is someone who ensures that value stream mapping is handled smoothly.
- **Logistics coordinator**: This person makes sure that all the necessary arrangements are made for the value stream mapping. This may include reserving the room for a face-to-face mapping meeting or getting the necessary web collaboration meeting configuration ready for a virtual meeting.
- **Meeting attendees**: The members of the mapping team will make up the rest of the team. This will be a diverse group that will have knowledge of the strategic elements of value streams, as well as direct experience of the tactical steps that development value streams will take.

These people should be shown the charter to allow them to be prepared for the mapping workshop. Questions may arise about possible situations. They should be brought to the workshop to be communicated to everyone.

Understanding the customer's perspective with Gemba walks

Because operational value streams directly involve the customer, it is important to understand the customer's perspective so that solutions are more valuable. One way to do that is through a **Gemba**, also referred to as *go see*, walk.

With a Gemba walk, members of the VSM team will go to a customer's site to view how the value stream works from the point of view of the customer and whether there are any necessary improvements.

When performing a Gemba walk, it is often beneficial to look at the final step of the value stream and work backward. This helps keep the focus on the customer perspective and values the delivery to the customer.

Gemba walks should happen often to understand not only the customer's mindset and the effects of the Value Stream from the customer's perspective but also whether the improvements to the value stream to reach the ideal value stream future state are having the desired effect.

Driving DevOps with Value Stream Management: Improve IT Value Stream Delivery with a Proven VSM Methodology to Compete in the Digital Economy outlines the following steps as a Gemba walk strategy:

1. Identify the goals and objectives.
2. Notify the customer of the Gemba walk visit and the reason for it in advance.
3. VSM team members should visit as a group.
4. Trace the flows from the value stream.
5. Discover any issues with the process and its related activities. Do not put the blame on people.
6. Document what you find.
7. Identify the root causes. The *Five whys* technique, where you repeatedly ask *why?* five times, is an effective way to drill to the root cause.
8. Listen.
9. Follow up with recommendations.
10. Repeat the walk to confirm that the recommended changes are being implemented.
11. Repeat the walk for other Kaizen (continuous improvement) opportunities.

The rest of the value stream identification will happen in dedicated workshops with the team members in attendance. Let's take a close look at how an operational value stream is identified during a workshop.

Creating the operational value stream

During the initial activities of a value stream workshop, you will be called to identify the operational Value Stream and the solutions that are the end products of development value streams.

Operational value streams typically fall into one of the following four categories, based on how they deliver value to the customer:

- **Digitally enabled product or service**: This operating value stream processes an online customer request for a product or service. E-commerce sales are a good example.

- **Manufacturing**: This operating value stream is responsible for the creation of physical products purchased by the customer.
- **Software products**: This operating value stream is responsible for software products delivered to the customer.
- **Support**: This operating value stream looks at the workflow for activities that support the customer, typically an internal customer.

Examples of these four categories of operational value streams appear in the following diagram:

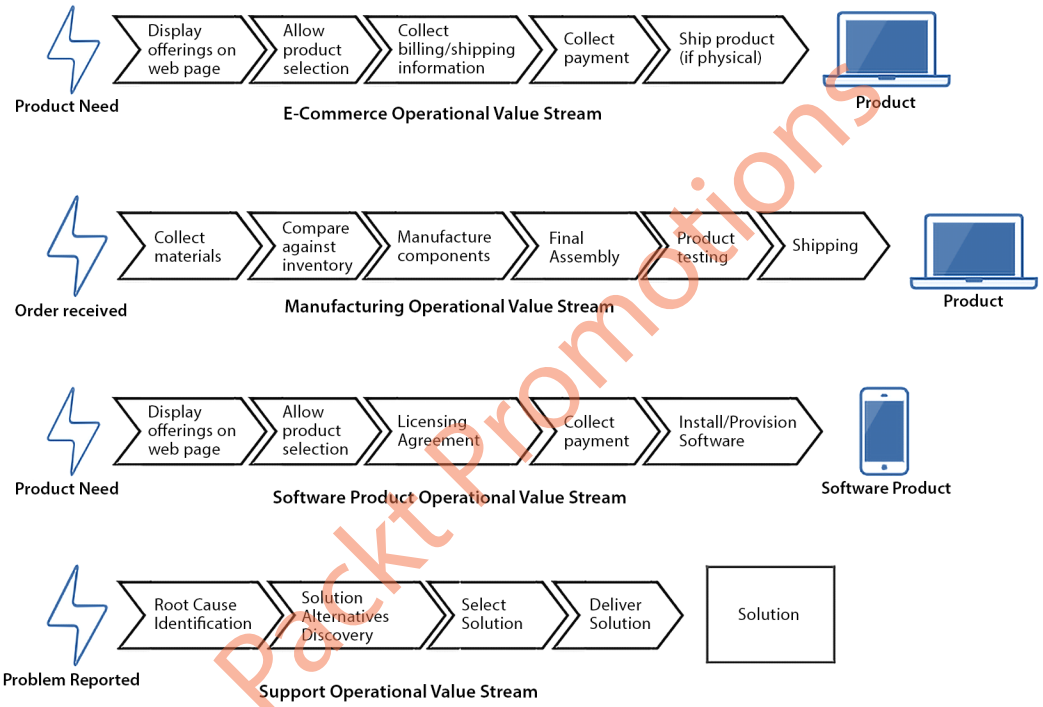


Figure 7.1 – Example operational value streams

Identifying the category of the operational value stream allows the mapping team to look at the journey the customer takes, and identify the possible solutions needed. They can then outline the steps that the customer takes to find value and create a note for each step.

The team then develops a chain of steps to form their organizational value stream, as shown in the following diagram, of the operational value stream which we first presented in *Chapter 2*, *Culture of Shared Responsibility*.

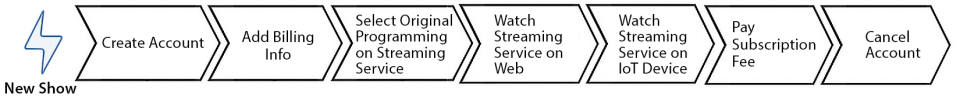


Figure 7.2 – Example of an operational value stream

The determination of the steps of the value stream may take some back and forth among the mapping team, particularly if there is a multitude of conditions. It may make sense to evaluate the most common conditions. If there is no consensus on the number of candidates for the operational value stream, it's a good idea to bring all the candidates forward and see how it affects the development value streams.

We have a good definition of an operational value stream when we can identify the following characteristics:

- The type of operational value stream
- The name
- A description of the value stream
- The customer(s)
- The triggers
- The value the customer receives
- The value the organization receives

When the steps of the operational value stream have been determined, it's time to find the solutions the operational value stream relies on.

Finding solutions

Once the steps of the operational value stream are identified, the next task is to look at each step and determine whether the step is enabled through a solution. SAFe defines a solution as a product, system, or service. Solutions could be something that the customer engages with or could be internal only.

The mapping of the solution to operational value stream steps is not one to one. Some steps may require the same solution, and some steps may not need a solution.

It may also be possible that a solution may support multiple operational value streams. Such a solution and the corresponding development value stream may be centralized in the organization.

We have identified the following solutions for our example operational value stream:

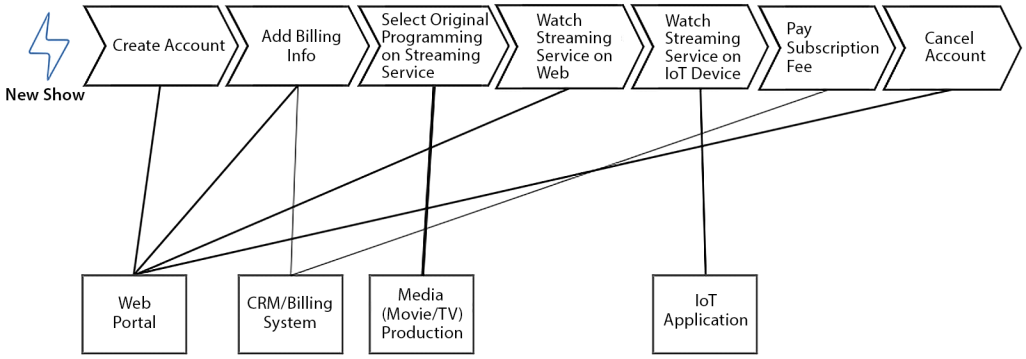


Figure 7.3 – Solutions and example of an operational value stream

Once we find the solutions, we are closer to finding the development value streams that create and maintain these solutions. We'll investigate how to determine those development value streams in the next section.

Mapping the development value stream

In SAFe, discovering the development value streams gets us closer to modeling those value streams as ARTs or even Solution Trains for larger solutions.

In mapping the way solutions are developed as development value streams, we are going to concentrate on the following aspects:

- The process and workflow
- The people involved in the process, such as suppliers and customers

Let's start our mapping journey by considering the process used to develop the solutions.

Finding the process and workflow

When looking at the development value stream, we can start by thinking of the development value stream initially as a *black box* with only its inputs and outputs visible. From there, we can piece together the intermediate steps.

The initial black box of the development Value Stream is formed by identifying the following characteristics:

- The trigger for the development value stream
- The first step of the development value stream
- The last step of the development value stream
- The demand rate for solutions

The mapping of the black box can continue from the determination of the solution as shown in the following diagram:

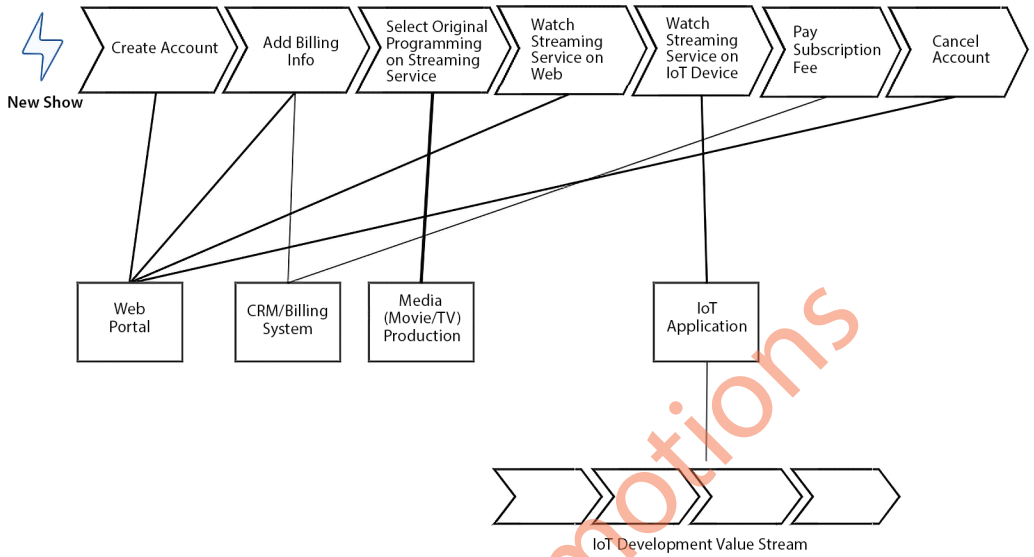


Figure 7.4 – Initial mapping of the development stream black box

When the members of the value stream mapping team are in agreement about the trigger, first step, last step, and demand rate, it's time to fill in all the steps that happen in between the first step and last step.

Connecting the dots

The mapping team now sets about filling in the steps between the first step and the last step of the development value stream. The authors of *Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation* recommend between 5 and 15 steps or **process blocks**.

It's important to remember that when mapping the process, we want to look at the current state of these development value streams, making sure we do not omit the bottlenecks and weaknesses in the processes. It's difficult to imagine an *ideal future state* for our development value stream without knowing how the current development value stream works.

The following diagram elaborates on the steps of our example development value stream:

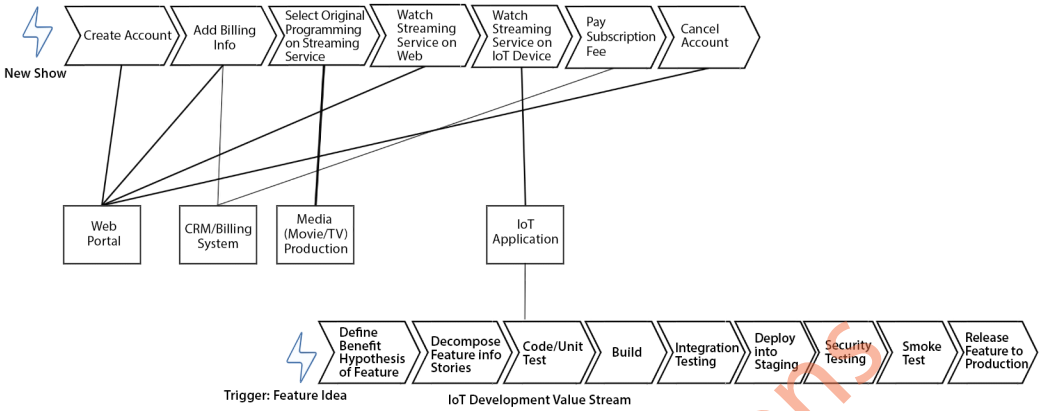


Figure 7.5 – Development value stream with steps

With the steps identified, the team now has the job of detailing each step. Let's look at what's involved in that part of the process.

Elaborating on each step

Now that the steps have been identified, we should see what goes on inside each step. The following characteristics are important to understand what occurs in an individual step:

- Batch sizes, if they are defined
- Defects
- WIP in the form of existing queues
- Necessary approvals
- Step metrics such as lead time, cycle time, and wait times
- Resources

Note that the preceding list represents barriers to the free flow of information and materials that should happen between steps. Recording the current state of this list is important for determining the future state. We will look at these in the *Finding areas for improvement* section.

After thoroughly evaluating each process step, the next step is determining who is responsible for executing each process step, and who receives the finished product. Let's look at that now.

Finding the people behind the process

Identifying the people involved is beneficial for looking at the potential future-state development value streams to see whether the correct teams or people are involved as well as whether the workload can be met by the existing teams.

We will focus on the following two roles within each step of the value stream:

- The people who perform the work identified in the process step.
- The people who are the recipients of the work done in the process step. They may be internal *customers* in that they receive information and material from the end of the process step to continue work in the next process step, or they may be external customers.

From what we have collected, we have a good idea of the present state of our development value stream. We will now look at what opportunities there may be for improvement.

Finding areas for improvement

Once the current state of the development value streams has been identified, the mapping team can look for areas for improvement and create a future-state development value stream. One other thing they'll create is an improvement plan to reach that future state.

To reach that future state, the mapping team will need to apply metrics to the current-state development value stream. These metrics will be taken at each step and then integrated into the metrics that evaluate the entire development Value Stream.

The complete picture of the current development value stream and the future-state development value stream is kept together in a DevOps transformation canvas. This canvas puts the work of the entire workshop in one place, allowing us to see where we are and where we want to go.

Let's start looking at our future state by examining the current state metrics for our development value stream steps.

Process step metrics

During the workshop, we identify metrics at each step to determine whether that step is a bottleneck that inhibits flow. By taking these measurements, we look to find bottlenecks and get optimal results in our future-state development Value Stream.

We measure the following metrics for each step in our current-state development value stream:

- The lead time
- The process time
- The **percent complete and accurate** (**%C&A**)

Let's take a close look at these metrics now.

The process time and lead time

When looking at an individual process step, we want to know how long it takes from when it leaves the previous process step or trigger to the point it leaves the process step. This total time is known as **lead time**.

Within the lead time, there may be idle times when the team executing the process step is waiting, perhaps for approval or review. Or, there may be times when value-adding work is being performed. These periods of activity within the process step are known as **process time**.

We previously talked about cycle time in *Chapter 4*, *Leveraging Lean Flow to Keep the Work Flowing*, where we saw that cycle time was related to the batch size and the size of the queue the team had worked on. If the team were working on a batch of items, the cycle time would be related to the process time by the following equation:

$$Cycle\ Time\ = \frac{Process\ Time}{Number\ of\ Items\ in\ Batch}$$

With this equation, if the team were working on one item at a time, its cycle time and process time would be equal.

Let's look at an example from our development value stream where we focus on two steps in the value stream, build and integration testing. If building and testing are done manually with no automation, there may be delays after the build step is completed when someone takes the completed build and starts the integration testing. The integration testing takes 3 hours to complete. A typical delay for the handoff from build to integration testing is 5 hours. This makes our process time for integration testing equal to 3 hours. Hence, our lead time for integration testing is our process time (3 hours) plus our handoff delay time (5 hours), or equal to 8 hours overall.

It's important to note that there may be different time scales involved between lead time and process time. Process time involves a calendar of business days (5 days per week), and each business day is made up of 8 hours. The lead time typically uses the standard calendar of 7 days a week, with each day composed of 24 hours. There may be exceptions in process time, particularly where geographically distributed development allows for efforts to *follow the sun*. Teams should also be clear on the distinction between business weeks versus calendar weeks.

The %C&A

Another process step metric is determined not by the people executing the process step, but by the recipients of the information or materials of the process step so they can execute the next process step.

These customers or recipients will be asked what percentage of the deliverables they receive are free of defects and can be used as is. This percentage is known as the %C&A.

Knowing the %C&A is important for identifying improvements. Process steps with low %C&A scores will have higher lead times, due to the need to perform rework.

Let's look at the two steps in our development value stream, build and integration testing. If a build leaving the build step fails 20% of the time, then when it reaches the integration testing step the %C&A will be *100% - 20% = 80%*.

Once the individual process step metrics are determined for all the process steps, you can calculate the composite value stream metrics for the entire development value stream.

Value stream metrics

With individual process step metrics discovered, the mapping team members can calculate metrics for the entire current-state development value stream. These metrics are composite metrics; that is, they are calculated by adding or multiplying together the individual process step metrics.

The following set of metrics is calculated to determine the overall performance of the development value stream:

- The total process time
- The total lead time
- The activity ratio
- Rolled %C&A

Let's see how to calculate these metrics.

The total process time and total lead time

We want to evaluate for the development value stream how long it takes from the initial trigger to the release of a feature. That can be calculated by taking the sum of each process step's lead time. This sum is known as the **total lead time**:

$$Total\ Lead\ Time = Lead\ Time_{1st\ step} + Lead\ Time_{2nd\ step} + \ldots + Lead\ Time_{last\ step}$$

Another composite metric to look at is how much time in the development value stream is active development time. To do that, we can add together all of the process times from each process step. That sum is called the **total process time**:

$$Total\ Process\ Time = Process\ Time_{1st\ step} + Process\ Time_{2nd\ step} + \ldots + Process\ Time_{last\ step}$$

The total process time and total lead time are used to calculate the metric we will see next.

The activity ratio

Once we have the total process time and total lead time, we may want to look at the productivity of the development value stream. To do this, we will look at the ratio of total process time to total lead time. This ratio is called the **activity ratio**:

$$Activity\ Ratio\ = \frac{Total\ Process\ Time}{Total\ Lead\ Time}$$

We now have a measure of how productive the development value stream is. The next metric we'll look at shows whether quality is embedded into the value stream.

Rolled %C&A

Remember that the %C&A is the measurement that people responsible for the *next* process step give to each process step, based on whether the deliverable is usable for continued processing. To determine the %C&A for the entire value stream, we look to the rolled %C&A.

To determine the rolled %C&A, we multiply together the %C&A of all of the process steps. We then multiply that value by 100 to get a percentage:

$$Rolled\ \%C\&A\ =\ \%C\&A_{1st\ step}\ \times\ \%C\&A_{2nd\ step}\ \times\ \ldots\ \times \%C\&A_{last\ step}\ \times\ 100$$

Metrics such as the process time, lead time, and %C&A provide important data to determine where the bottlenecks to flow are occurring. To improve our development value stream, we need to see what the *ideal* future development value stream looks like. Let's explore how to do that now.

Identifying the future development value stream

While identifying the development value stream, a lot of discussions may be generated by what the steps of the Value Stream should be versus what they actually are. Such differing views are beneficial to understanding the "ideal" development Value Stream and may provide inputs as to the shape of the evolution of the development Value Stream.

Mapping the future-state Value Stream may occur after the current Value Stream has been mapped and is running to collect data. This data is valuable for highlighting bottlenecks to flow and other weaknesses.

Karen Martin and Mike Osterling, in *Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation*, recommend looking at the following perspectives when approaching the future Value Stream:

- Determining the correct steps in the Value Stream
- Looking for flow
- Looking for continuous improvement

Let's look at each of these perspectives in detail.

Fixing the process and steps

A key focus of Lean thinking is the elimination of waste. We can apply that focus to our future-state Value Stream mapping by looking for processes and steps that do not add value and removing them. We start by categorizing the steps as value-added, necessary non-value-added, and unnecessary non-value-added work. We attempt to focus on removing unnecessary non-value-added work first. Next, we reevaluate the necessary non-value-added work to see whether it can be reduced, or whether it can be reclassified as unnecessary non-value-added work and can be eliminated.

When removing these steps, it's important to consider the reason for these steps in the first place and ensure that this reason can still stand even with the removal of the step. An example of this is an inspection step near the end of our Value Stream. The reason for this step may be to ensure quality. If the Value Stream contains automated testing steps at various levels, the inspection step may be removed without harming quality.

Another way of fixing our Value Stream may be to add additional processes and steps. It may seem counterintuitive to add steps if the focus is to remove waste, but the addition of steps, even on a temporary basis, allows teams to establish new ways of doing things on the Value Stream so unnecessary non-value-added work can be removed at a later time. If we look at improving quality, we may want to add testing steps to our Value Stream while keeping the inspection steps so that the inspections can be removed once we have confidence in the testing.

Looking for flow

It may not be enough that we have the correct steps in our Value Stream. We want to ensure that work moves from step to step in our Value Stream, without problems or delays. For this reason, we look to ensure that flow is occurring in our Value Stream.

The metrics that we have gathered for each step in our Value Stream (process time, lead time, and %C&A) are important for fixing flow. They help determine which steps are the bottlenecks in the Value Stream. We look for those steps where the lead times include long wait times. We also look at steps that have low %C&A percentages to fix, because those steps indicate that rework is being performed.

Once we've found the steps, we apply the *theory of constraints* that Eliyahu M. Goldratt identified in his book *The Goal*. The theory of constraints details the following steps for dealing with bottlenecks and achieving flow:

1. Identify the constraint.
2. Decide how to overcome the constraint.
3. Focus on overcoming the constraint, subordinating all other concerns.
4. Overcome the constraint.

5. Once the constraint has been overcome, other constraints (bottlenecks) may appear. Go back to *step 1* to deal with them.

Removing bottlenecks at a specific step may introduce other bottlenecks at other steps, but the overall metrics (total process time, total lead time, and rolled %C&A) will show improvement.

Looking for continuous improvement

This combination of optimizing steps and removing bottlenecks may give us a better Value Stream, but now we want to make sure we have ways of ensuring that the performance is improving. For this reason, we will establish two to five **key performance indicators** (**KPIs**) and regularly track their measurements. The KPIs may deal with measurements for cost, quality, safety, and morale.

After collecting and calculating the information about the current- and future-state development value streams, it may be beneficial to place this information in a single location for reference as we seek to improve our Value Stream. We'll look at one such location, the DevOps transformation canvas.

The DevOps transformation canvas

The DevOps transformation canvas can serve as the backdrop to the mapping of your development Value Stream, as well as identifying the future state and implementation steps to take us there.

The DevOps transformation canvas has areas for the current-state Value Stream and the following characteristics:

- The trigger
- The first step
- The last step
- The demand rate
- Value Stream metrics (total process time, total lead time, activity ratio, and rolled %C&A)

For identifying the ideal future state, the DevOps transformation canvas includes the following areas to fill out during the workshop:

- Target Value Stream metrics (total process time, total lead time, activity ratio, and rolled %C&A)
- The boundaries/limitations
- Improvement items

An example DevOps transformation canvas is as follows.

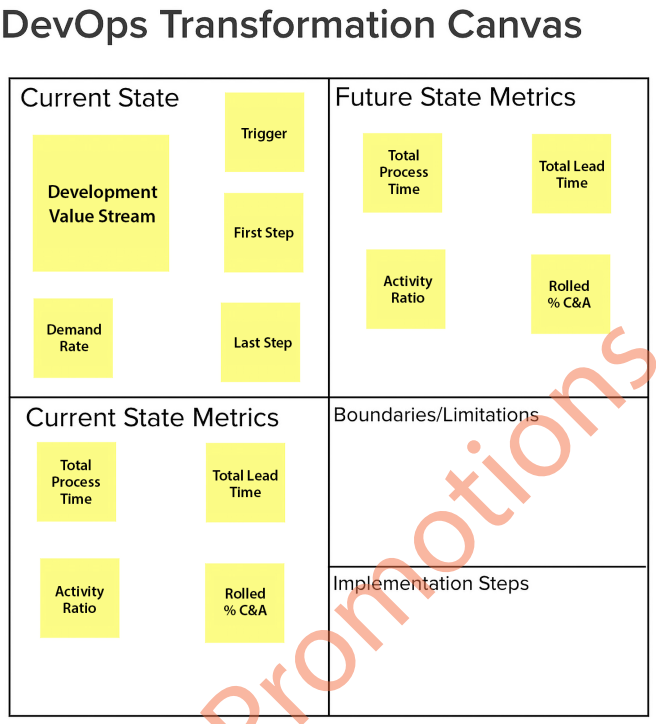


Figure 7.6 – DevOps transformation canvas

With the completion of the workshop, the work begins on improving our development value streams and measuring the effectiveness of the improvements. This is the topic we will delve into in our next chapter.

Summary

In this chapter, we took an extensive look at value streams. We saw that aligning our activities in terms of value streams is a key approach used in DevOps and is a major part of transforming SAFe through its implementation roadmap.

We started out by looking at the current state of our organization. We prepared the team to look at mapping our value streams and looked at how we receive work through Gemba walks. From this examination, we saw what our process is and any deficiencies in it.

We then looked at discovering and mapping our value streams. We started by mapping our operational value streams, which connect us to our customers. From there, we started looking at the solutions that our operational value streams use to bring value to our customers. With the solutions in hand, we

looked at the development value streams that create and maintain those solutions. Finally, we looked at metrics to view the current state of the development Value Stream and measure our improvement as we strive toward the future-state development Value Stream.

Now that we have identified our development value streams, we will look for feedback to see whether our improvements are having the desired effect. The next chapter shows us how to collect that feedback.

Questions

1. Which of these things is not identified when doing Value Stream mapping?

   A. Trigger
   B. Demand rate
   C. Location
   D. First step

2. Which of the following identifies the time taken to complete a process step from handing off from the previous process step?

   A. Demand rate
   B. Lead time
   C. Process time
   D. Cycle time

3. Who determines the %C&A of a process step?

   A. The customer
   B. The people that work on the process step
   C. The people that work on the next process step
   D. The people that work on the previous process step

For the remaining questions, use the following illustrated Value Stream:

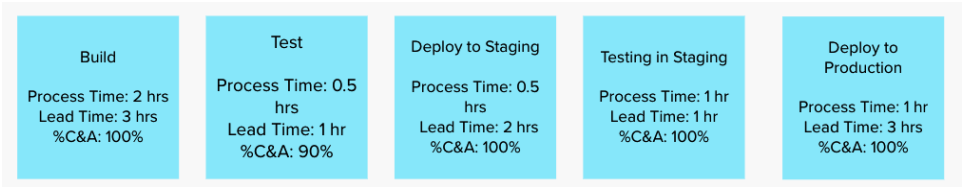


Figure 7.7 – Illustration of a Value Stream

4. What is the total process time?

   A. 0.5 hours
   B. 2 hours
   C. 1 hour
   D. 5 hours

5. What is the total lead time?

   A. 5 hours
   B. 3 hours
   C. 1 hour
   D. 10 hours

6. What is the activity ratio?

   A. 1.0
   B. 0.66
   C. 0.25
   D. 0.5

7. What is the rolled %C&A?

   A. 100%
   B. 490%
   C. 90%
   D. 400%

Further reading

- *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win* by Gene Kim, George Spafford, and Kevin Behr – A "must-read" introduction to DevOps. This book introduces the Three Ways that we use to examine VSM.
- `https://www.scaledagileframework.com/implementation-roadmap/` – The reference from SAFe that talks about the implementation roadmap.
- `https://www.scaledagileframework.com/development-value-streams/` – An article from SAFe that talks about development value streams and their relationship to operational value streams.

- *Driving DevOps with Value Stream Management: Improve IT Value Stream Delivery with a Proven VSM Methodology to Compete in the Digital Economy* by Cecil "Gary" Rupp – The reference for VSM that includes guidance on Value Stream mapping. This is a resource used by the VSM Consortium.
- *Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation* by Karen Martin and Mike Osterling – The reference for how to perform Value Stream mapping.
- *The Goal* by Eliyahu M. Goldratt – This reference introduces the theory of constraints.

8

Measuring Value Stream Performance

In our last chapter, we started our exploration of Value Streams by referring to the Three Ways found in *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*. The First Way was about achieving flow through the establishment of Value Streams.

This chapter moves beyond the establishment of Value Streams to verify their performance. For this chapter, we look at the Second Way: *amplify feedback loops*. To follow the Second Way, we need to look for and pay attention to the feedback from our Value Streams.

We can use metrics as a feedback mechanism. There are a few metrics frameworks that have emerged, such as **DevOps Research and Assessment** (**DORA**) metrics and the Flow Framework®, that can function as feedback loops.

We will cover the following topics in this chapter:

- Creating good measurements
- Looking at the DORA metrics
- Looking at the Flow Framework® and Flow Metrics
- Understanding measurements in SAFe®

Let's begin by understanding how to get effective feedback.

Creating good measurements

We originally saw the different types of measurements we can apply in *Chapter 5*, *Measuring the Process and Solution*, where we looked at measurements in three of the following areas:

- The development process

- The environment
- The customer value delivered

The set of metrics that can adequately cover these three areas of measurement are called **Key Performance Indicators** (**KPIs**).

KPIs are metrics chosen to determine whether an individual, a team, a Value Stream or an **Agile Release Train** (**ART**) in our case is meeting its goals. KPIs are evaluated both at specific points in time as well as over a given time period to highlight trends and movements away from or toward a goal.

KPIs should be objective measurements and not subject to opinion or interpretation.

While looking at these measurements, we cautioned against using vanity metrics, which are measurements that yield good information but don't really supply any meaningful data. Examples of this include the number of hits on a website or the cumulative number of subscribers to a service.

To set up KPIs, let's look at the steps in the following diagram, as advised by `kpi.org`.

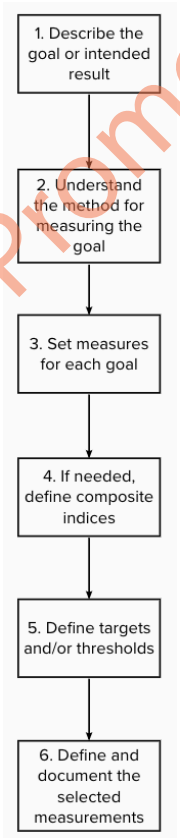


Figure 8.1 — KPI establishment process

Let's look at each step in detail.

Describing the goal or intended result

When looking at the intended goal, it's important to understand whether it is a strategic goal or a result that is more tactical. The desired goal should be a concise, concrete result rather than a broad objective.

In this way, KPIs are different from **Objectives and Key Results** (**OKRs**). OKRs generally describe broad strategic goals with desired results as the measurement of whether that strategic goal was achieved. An example of an OKR is shown in the following table:

Objective	**Key Results**
We delight customers at every opportunity through excellent usability and service	Our usability score goes up from 6 to 9 by Q3
	Our customer satisfaction score goes up from 7 to 10 by Q3
	The number of monthly active users goes up from 56% to 65%

Table 8.1 — Example of an OKR

A KPI points to whether a concrete target was achieved. An example of a KPI would be the measurement of **Net Promoter Score** (**NPS**) survey results from favorite customers.

If you have performed a value stream mapping as shown in the last chapter, *Chapter 7*, *Mapping Your Value Streams*, then the desired future state of your value stream is an effective tactical goal that your ART or value stream can strive toward.

Understanding the method for measuring the goal

For your intended goal, you need to understand how the measurements will work. Are there direct measurements that will correspond to the intended goal? If so, those should be the measures you use.

However, what happens if you cannot directly measure the intended goal? In that case, you may need to look at creating hypotheses and measuring the results against the hypotheses as experiments.

Selecting measures for each goal

With the goals in place and possible measurements defined for each goal, it's time to winnow down your choices to the important measurements to take. In most cases, a total of five to seven KPIs is enough to get an adequate picture of how the value stream is doing. It's better to focus on a few key measurements as opposed to being awash in a sea of data.

We want the following characteristics for our KPIs:

- Answer questions about our performance in line with our objectives

- Provide clear information needed for strategy
- Are valid and can be verified
- Encourage desirable employee behavior
- Are not difficult to acquire

Now that we have our measurements, we need to see what the ideal values are for our value stream.

Defining composite indices if needed

There may be some measures that cannot individually provide all of the information to get the intended result. This is especially pertinent when the intended result is intangible, such as customer satisfaction.

When that is the case, you may need to bundle individual measurements into a composite index so that analysis becomes easier.

Defining targets or thresholds

We want to see how we are performing against the KPI. To judge our performance, we need to set a target value for each KPI. This target value should reside in a threshold for optimum performance. Thresholds should also be defined for poor performance.

Defining and documenting the selected measurements

We have now defined our KPIs, as well as target values and thresholds for good, satisfactory, and poor performance. It's time to expand and detail other information about our KPIs.

The following additional information may help when collecting and analyzing our KPIs:

- Its intended goal
- The KPI
- A description of the KPI
- The type of measurement
- The formula to calculate the measurement
- Units of measure
- Where the measurement is stored
- Who makes the measurement and is responsible for it
- The data source
- The frequency at which it is collected and reported

- The individuals responsible for validation
- The individuals responsible for verification
- The method of presenting the KPI

For a development value stream, it may be acceptable to start with a standard set of KPIs. One standard of this kind was devised by DORA and examined in annual surveys. Let's look at these metrics and see their applicability for a new development value stream.

DORA metrics

Since 2014, an annual report detailing the state of DevOps adoption has been published by Nicole Forsgren, Gene Kim, and Jez Humble of DORA and Alanna Brown of Puppet. Each year, they detail the general state of DevOps adoption and the maturity of respondents in adopting DevOps practices.

In 2016, they outlined certain metrics meant to measure the throughput and stability of DevOps practices across different organizations. These metrics have come to be known as **DORA metrics**.

The annual report has served as a barometer of the extent to which DevOps practices are being incorporated and how effectively this is being done. Each year, the report identifies the following aspects of the DevOps movement:

- Key KPIs
- Performance levels of organizations based on the KPIs
- Upcoming trends

Let's look at each of these aspects now.

The DORA KPI metrics

The Accelerate State of DevOps report looks at the following four metrics to determine performance levels for participating organizations:

- Lead time
- Deployment frequency
- Change failure rate
- Mean time to repair

The first two metrics measure an organization's velocity, or how quickly it can deliver changes to production. The third and fourth metrics determine an organization's stability, or how well an organization can keep its production environment running.

Let's look at these metrics in more detail.

Lead time

We have discussed lead time before in our previous chapter, *Chapter 7*, *Mapping Your Value Streams*. In it, we saw that each process step has a lead time. The total lead time was the sum of the lead times of all the process steps.

The DORA metrics look for the lead time for changes – that is, the length of time from a commit to the version control repository to the deployment of that code into a production environment. This is a subset of the total lead time. The authors of the book *Accelerate: Building and Scaling High Performing Technology Organizations*, Nicole Forsgren, Jez Humble, and Gene Kim (all key contributors to the definition of DORA metrics), specify the delivery lead time and disregard any lead time related to design. This is primarily due to the uncertainty of when to *start the clock* for the design lead time. The delivery lead time is easier to measure and is resistant to any variability.

Let's look at the following illustration of a sample value stream:

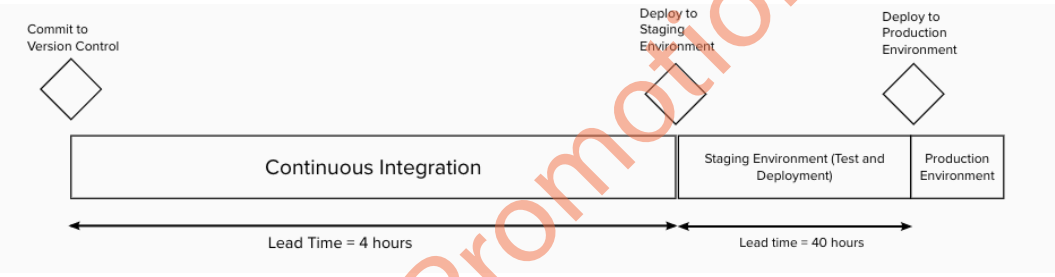


Figure 8.1 — Sample value stream for DORA lead time

In the preceding diagram, we see that with Continuous Integration, automation performs tests in a non-production environment. If no errors are found, this process takes 4 hours, and the changes are deployed to a staging environment.

In the staging environment, perhaps automated testing is not performed. This may explain why the time to move from a staging environment to a production environment takes 40 hours.

So, for this value stream we total the times for each stage (4 hours + 40 hours), resulting in a total lead time of 44 hours.

Deployment frequency

The DORA metrics look at how frequently organizations can successfully deploy code to production with this KPI.

If we continue with our value stream example that has a lead time of 44 hours, or 1.83 days (given that lead times are measured against a 24/7 calendar), we can see that they deploy roughly 16 times per month.

Change failure rate

This metric is the examination of quality in the deployment process. It measures how often a release to production results in a degraded or failed service that requires a fix, either by rolling back or patching and implementing a hotfix.

Determining the change failure rate comes from examining the record of deployments and seeing whether any directly led to any production failures.

Suppose that in our value stream, we looked at the past 12 deployments. Of those 12 deployments, the production environment experienced three problems.

Our change failure rate would come from the following calculation:

$$change\ failure\ rate = \frac{failures}{deployments} = 3/12 = 25\%$$

Recovery time

This metric looks at instances where service incidents occurred in production environments. For those incidents, how long did it take to restore service?

Often, when there is more than one incident, the metric for recovery time is expressed as the **Mean Time to Recovery** (**MTTR**), or the average time to recover.

In our value stream, for those three failures that were experienced in the past 12 deployments, if the times to repair for each failure were 3 hours, 2 hours, and 7 hours, the mean time to repair would be that shown in the following calculation:

$$MTTR = \frac{(3+2+7)}{3} = 4 \text{ hours}$$

For the preceding four metrics, DORA has done a cluster analysis of the responses and established four performance levels. Let's look at these levels.

DORA metric performance levels

Every year in the Accelerate State of DevOps report, DORA analyzes the responses to view the levels of performance among the respondents. From the four DORA metric KPIs, they have created the following four performance levels:

- Elite
- High
- Medium
- Low

The criteria for each level change every year that the survey is produced. This is due to an overall improvement in practices not only at each organization but also within the industry as a whole. The 2021 report saw an increase in the number of respondents that were at the Elite or High performer levels compared with previous years, signifying the role continuous improvement plays in adopting DevOps practices.

The 2022 State of DevOps report saw a major change in these performance levels. For the first time, the Elite tier was removed. The survey results indicated that the highest-performing group was not performing at the same level as the Elite tier in the previous year. More data is required to look for possible reasons.

Emerging trends from the State of DevOps report

To adjust to changing times, the survey also includes ancillary questions about the organization's environment in order to view emerging aspects. The two most recent reports (2021 and 2022) included the following additional items:

- Since 2018, DORA has added another metric: reliability. This measure looks at performance beyond software delivery into how well an organization maintains its environments or operational performance.
- DORA has investigated the adoption of cloud infrastructures since 2019, noting that the adoption of cloud technologies is an enabling technology that improves all four DORA metric KPIs.
- The 2021 report started investigating the adoption of SRE practices, as a way of finding a correlation with reliability.
- In addition to inquiring about technical DevOps practices, DORA has expanded the scope of its inquiry to include documentation and security practices integrated into the development process.
- Because of the disruption to established work patterns due to the COVID-19 pandemic, questions were included to gauge the resiliency of organizations to continue delivering while avoiding burnout.
- As part of the inquiry of security practices, the 2022 State of DevOps report inquired about whether companies had adopted measures to ensure the security of their software supply chain. These measures fall into one of two frameworks for standardization: **Supply Chain Levels for Software Artifacts** (**SLSA**) and the **Secure Software Development Framework** (**SSDF**).

The DORA metrics offer a good elementary look at KPIs to measure, but often, not all the work done by value streams is directly related to providing customer value. To measure those KPIs, adopting the Flow Framework® and measuring the Flow Metrics® may be a good option. Let's now take a look at the Flow Framework® model.

Flow Framework® and Flow Metrics®

The Flow Framework® model is described in detail in Mik Kersten's book *Project to Product: How to Survive and Thrive in the Age of Digital Disruption with the Flow Framework*. In this book, Kersten describes the need to move from project-based design to product development using long-lived value streams.

To measure the performance of the value streams, Kersten proposes the Flow Framework®, a structure of Flow artifacts and the measurement of those artifacts using Flow Metrics®.

Kersten initially formulated the Flow Framework® in order to measure the flow of software delivery for his company, Tasktop. While looking at the value streams at Tasktop, Kersten identified the following four outcomes that he wanted to monitor:

- Value
- Cost
- Quality
- Happiness

He related these items to four Flow Items, which were the types of work done by Tasktop's value streams. To track the progress of these Flow Items, Kersten found four Flow Metrics® that these Flow Items exhibited.

Let's start our look at the Flow Framework® by examining the four Flow Items.

Flow Items

Kersten identified the following four types of work done by a value stream:

- Features to deliver
- Defect fixes
- Risk avoidance
- Technical debt reduction

Each item has unique differences in terms of the stakeholders that desire them and the value of these items.

Features

Features are items that bring about direct business value. Customers pull features from the value stream to provide the solutions they want or need. This work is considered the primary type of work a value stream will deliver.

In SAFe®, Flow Framework® features can be mapped to features, the items of work that are decomposed into user stories and enabler stories, which have a timebox of one **Program Increment** (**PI**).

Defect fixes

A value stream may also work on fixing any defects discovered. These fixes may be for defects uncovered during the development process or defects that have made their way to production. Regardless of when the defect was discovered, the fix is pulled by the customer as part of the solution.

SAFe does not directly identify separate work for fixing defects, but that work is frequently tracked as part of the work tracking system that value streams use. An example of this is when value streams using Jira identify a distinct issue type (bug) to track the effort to fix defects.

Risk avoidance

Value streams may work in organizations where compliance, security, governance, and privacy are important **Non-Functional Requirements** (**NFRs**). These NFRs may be related to the industry they're in, which may have important contractual requirements or regulations to comply with. Items meant to reduce risks are delivered by the value stream to different stakeholders. These stakeholders are frequently internal to the organization, such as security or governance groups.

In SAFe, items that are meant to fulfill NFRs and mitigate or eliminate risks are compliance enablers. Compliance enablers that are timeboxed to a PI are identified as features meant for an ART to accomplish while story-sized compliance enablers are worked on by an individual team inside the ART.

Technical debt reduction

Paring down technical debt is an important kind of work that value streams carry out. If technical debt is not managed to a controllable level, the delivery of the other Flow Framework® items (features, defect fixes, and risk avoidance) will be impacted due to deficiencies in the architecture.

SAFe categorizes Flow Framework® debt items as enablers. We have seen compliance enablers when talking about Flow Framework® risks. The other types of enablers help to maintain the architecture of a product or solution.

Infrastructure enablers are used by ARTs and teams to enhance the development process. Work of this kind includes the incorporation and maintenance of automation for testing and deployment.

Architectural enablers directly improve the architecture of the ART's product or solution. A series of architectural enablers created so that these enhancements can be utilized by future features is known as the Architectural Runway.

Exploration enablers allow team members of the ART to research unknown technologies or determine the best functional approach. Spikes are a prevalent example of exploration enablers.

Value streams that use the Flow Framework® for measurement divide their work into all four of these Flow Items. We will now take a look at the measurements applied to these items.

Flow Metrics

In the Flow Framework®, we want to see how well our value stream is performing against all the kinds of work there are. To do that, we will apply the following measurements to each Flow Item:

- Flow Velocity®
- Flow Time®
- Flow Load®
- Flow Efficiency®

In addition, we have one more metric, Flow Distribution®, so we can see which Flow Items the value stream works on most.

Let's look further into each metric now.

Flow Velocity®

Flow Velocity® looks at the number of Flow Items, regardless of type, completed in a standard unit of time. The following diagram illustrates Flow Velocity® for a Value Stream:

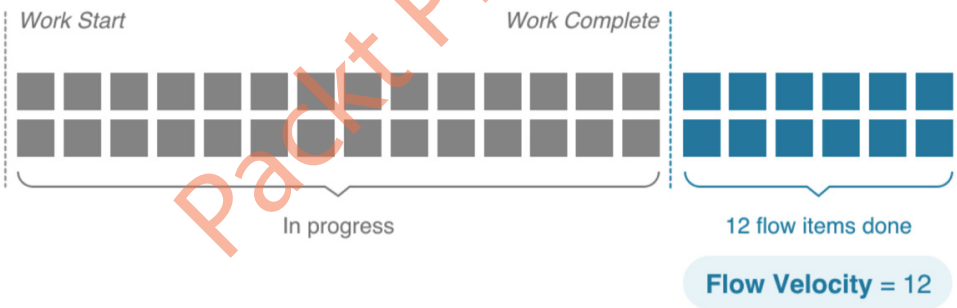


Figure 8.3 — Illustration of Flow Velocity® (Copyright © 2018 Tasktop Technologies, Incorporated. All rights reserved. Published with permission)

This is analogous to measuring the velocity in Scrum. A value stream that is stable and working well will maintain consistent Flow Velocities® across multiple periods of time.

Flow Time®

Flow time® is the time to complete an individual Flow Item as it traverses the value stream, from when it is accepted into the value stream to when it is released by the value stream. The Flow Time® includes both active times and wait times. The following diagram illustrates Flow Time®:

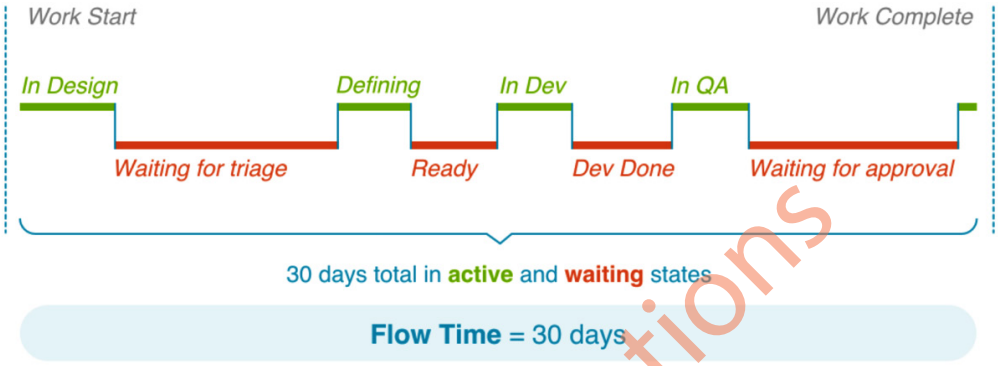


Figure 8.4 – Illustration of Flow Time® (Copyright © 2018 Tasktop Technologies Incorporated. All rights reserved. Published with permission)

The difference between Flow Time® and lead time is that the latter is a customer metric. With Flow Time®, we're looking to determine the length of time needed to develop our product or solution.

Flow Load®

We discussed the problems with having large numbers of WIP in *Chapter 4*, *Leveraging Lean Flow to Keep the Work Moving*. Flow Load® is a measure of WIP. As shown in the following diagram, we can see the number of Flow Items that are in progress with Flow Load®:

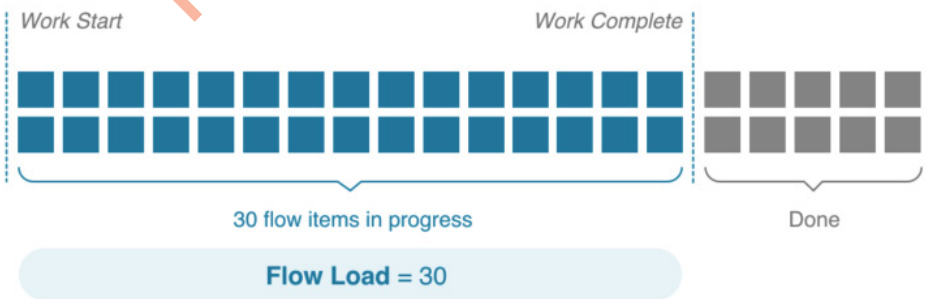


Figure 8.5 – Illustration of Flow Load® (Copyright© 2018 Tasktop Technologies Incorporated. All rights reserved. Published with permission)

Remember that just as high numbers of WIP lead to longer lead times and reduced performance, high Flow Load® values are an indicator of reduced performance that will result in longer Flow Times® and reduced Flow Velocity®.

Flow Efficiency®

We looked at Flow Time® earlier and saw that it includes both times when the value stream is actively working and times when it is idle, waiting at some process step. You can figure out the efficiency by looking at the ratio of active time to Flow Time®.

The following diagram completes our preceding Flow Time® example by calculating the Flow Efficiency®:

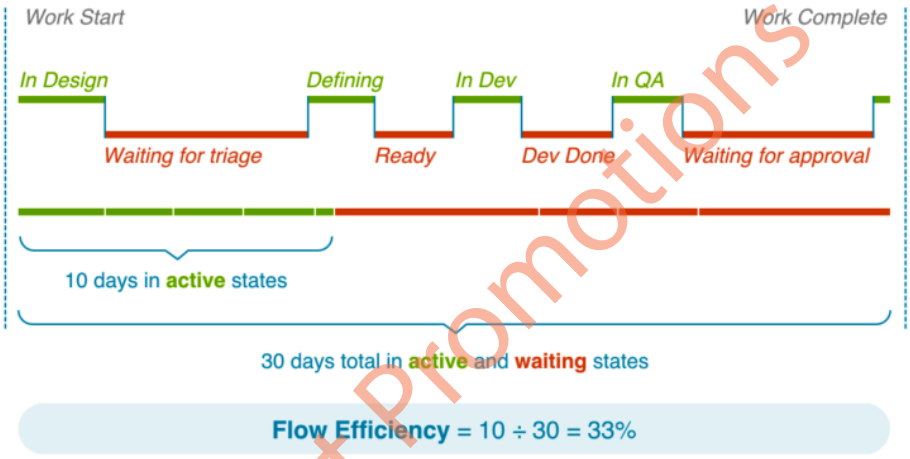


Figure 8.6 – Illustration of Flow Efficiency® (Copyright © 2018 Tasktop Technologies Incorporated. All rights reserved. Published with permission.)

Flow Efficiency® is analogous to activity ratio, which was introduced in *Chapter 7*, *Mapping Your Value Streams*. The difference is that Flow Efficiency® looks at the development perspective of the value stream.

Flow Distribution®

When looking at the Flow Metrics® so far, we have not considered the type of Flow Item that is being measured. We will now look to Flow Distribution® to guide us on whether the work of our value stream is balanced.

Flow Distribution® looks at the number of Flow Items complete for a value stream and measures each type of Flow Item as a percentage of the total number of Flow Items. A calculation of Flow Distribution® is demonstrated by the following diagram:

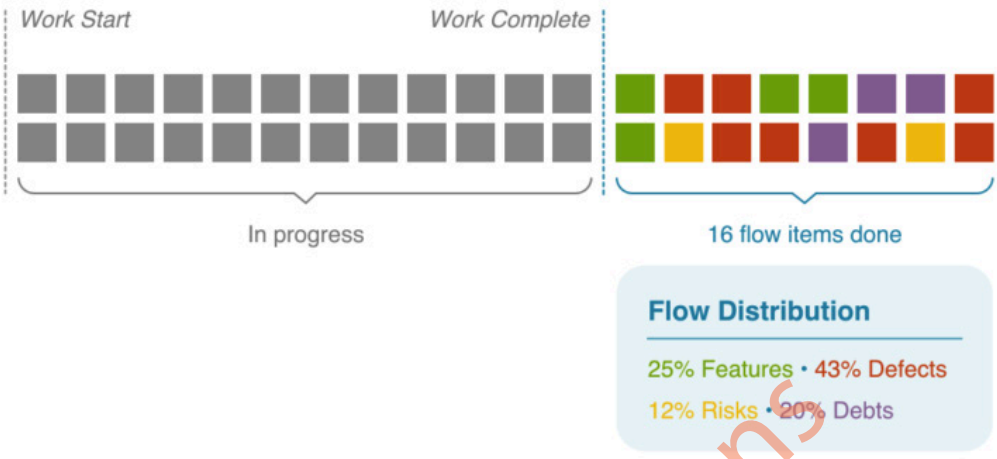


Figure 8.7 – Illustration of Flow Distribution® (Copyright © 2018 Tasktop Technologies Incorporated. All rights reserved. Published with permission)

In SAFe, looking at the Flow Distribution® allows the ART to determine the proper allocation ratio of features and enablers so that a proper balance of delivering the customer value and ensuring that the needed enhancements are made.

Measurements in SAFe

When looking at an organization's value streams realized as ARTs, Scaled Agile recommends viewing their performance in terms of these three following aspects:

- Outcomes
- Flow
- Competency

Let's see how these three aspects are measured in SAFe.

Measuring outcomes in SAFe

The primary mechanism for measuring outcomes comes from establishing and measuring value stream KPIs.

We saw KPI frameworks for measuring customer outcomes, such as **Pirate** (**ARRRR**) metrics and Fit-for-Purpose metrics, in *Chapter 5*, *Measuring the Process and Solution*. Determining the set of KPIs for your team or ART was discussed earlier in this chapter.

Validating the benefit hypothesis that comes from Epics, large pieces of work, may lead to desirable outcomes. Epic development is done experimentally by creating a **Minimum Viable Product** (**MVP**) and using leading indicators to measure the hypothesized value. Closely monitoring the leading indicators produces evidence that either validates or invalidates the Epic hypothesis, allowing us to *pivot or persevere* with further Epic development.

Agile teams on the ART also want to measure the Iteration Goals they create at every iteration, as well as the PI Objectives they create for every PI. These goals and objectives help the teams focus their efforts, not on completing every feature and story, but on making sure customer value is delivered.

Measuring the Flow in SAFe

Some of the value stream KPIs a team or ART decide to adopt will be related to performance in terms of ensuring the delivery of value. Scaled Agile has recommended using the Flow Metrics® from the Flow Framework® to ensure that the flow successfully results in the delivery of the product. The Flow Items and Flow Metrics® against those items have been discussed in the preceding section.

In addition, Scaled Agile recommends an additional Flow Metric: Flow Predictability. This metric measures how well teams and the ART can plan their PI and meet their PI Objectives.

To measure Flow Predictability, teams and the ART use the SAFe **Program Predictability Measure** (**PPM**). To calculate the PPM, teams look at the original Business Value of the PI Objectives determined during PI Planning. They then compare the sum of these against the actual Business Value for both committed and uncommitted PI Objectives determined in the Inspect and Adapt workshop at the end of the PI. The measure is determined by the following equation:

$$Team\ \mathrm{Pr}\,e\,dictability = \frac{sum\ of\ actual\ Bu\,\mathrm{sin}\,e\,ss\ Value\ (committed\ and\ uncommited\ PI\ Objectives)}{sum\ of\ original\ Bu\,\mathrm{sin}\,e\,ss\ Value\ (committed\ PI\ Objectives\ only)}$$

An example of the calculation for a team's PPM is shown in the following table:

	PI Objective	**Planned (Original) Business Value**	**Actual Business Value**
Committed Objectives	Increase indexing speed by 50%	9	4
	Build and release e-commerce capabilities	10	10
	Build and release intelligent search	8	4
Uncommitted Objectives	Add 2,000 new products to the database and index them	7	7
	Support purchase with Bitcoin	5	0

Table 8.2 — Example calculation for a team's PPM

In the preceding table, the sum of actual Business Value for both committed and uncommitted PI objectives is 25. The sum of the planned Business Value is 27. The team PPM is then 25/27:

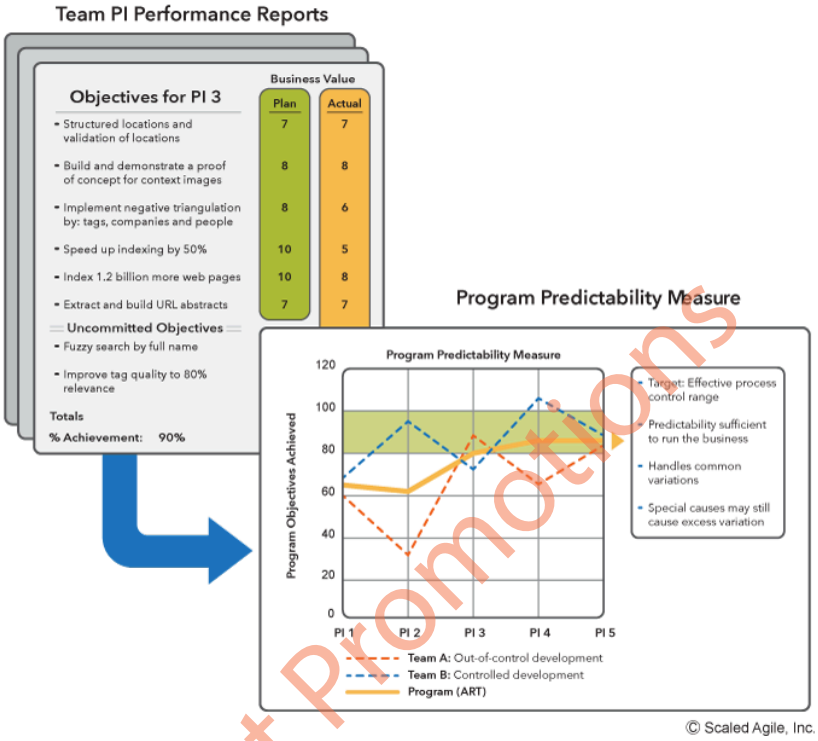


Figure 8.8 — An example team predictability rollup to PPM (© Scaled Agile, Inc. All rights reserved)

The preceding diagram shows the rollup that happens to create the PPM. The ART receives a PPM by averaging the PPM values of the teams that make up the ART.

Measuring competency in SAFe

On a grander scale, an enterprise that utilizes SAFe strives for business agility, where strategy and execution are combined to achieve business results through the frequent delivery of customer value.

Business agility in SAFe is measured through self-assessment of the following seven core competencies practiced by different parts of the organization:

- Team and technical agility
- Agile product delivery
- Enterprise solution delivery

- Lean portfolio management
- Organizational agility
- Continuous learning culture
- Lean-Agile leadership

The teams and the ART use the DevOps Health Radar to assess their competency in adopting DevOps principles and practices. Let's take a look at how to use the DevOps Health Radar.

The DevOps Health Radar

The DevOps Health Radar is a tool that lists all the activities related to the four aspects of the Continuous Delivery Pipeline. An illustration of the DevOps Health Radar is as follows:

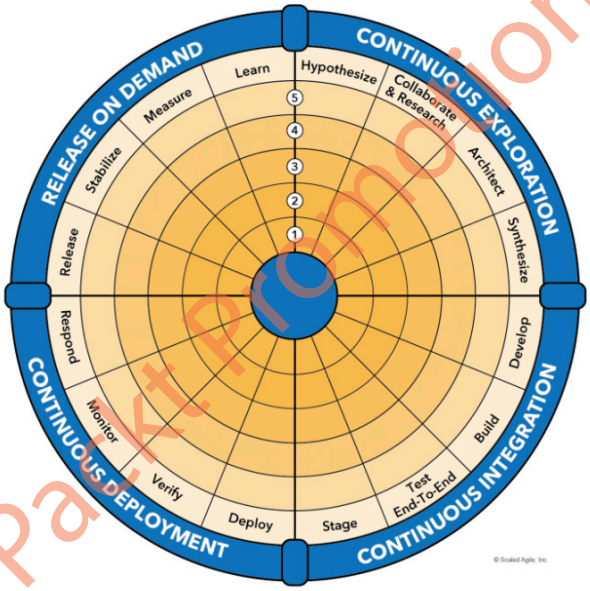


Figure 8.9 – DevOps Health Radar (©Scaled Agile, Inc. All rights reserved)

For each of the 16 activities of the Continuous Delivery Pipeline, the teams and ART rate themselves in terms of their maturity in performing the activity. Ratings range from Sit (1 to 2), Crawl (3 to 4), Walk (5 to 6), Run (7 to 8), and Fly (9 to 10), the apex.

Teams and the ART should periodically self-assess with the DevOps Health Radar to track the maturity of their DevOps performance. This assessment is available for free on the Scaled Agile Framework website at `https://www.scaledagileframework.com/?ddownload=38794`.We will examine the Continuous Delivery Pipeline and its activities in *Part 3*.

Summary

In this chapter, we wanted to ensure that we were on the right track by adopting the Second Way through amplifying feedback loops. The key feedback loops that we will use for our value streams are often metrics.

When selecting metrics, we want to view them as KPIs. We saw how to start with our desired objectives, look at the metrics that line up with our objectives, refine a set of metrics to collect, and collect them as KPIs.

We first looked at one standard of metrics to use as part of our set of KPIs: the DORA metrics that form the basis of the annual Accelerate State of DevOps report. By collecting these metrics and continuously improving, a value stream may be identified with a performance level based on comparison with other organizations as collected by the annual report.

If looking at other types of work beyond those that provide customer value, a value stream may look at the Flow Framework® created by Tasktop. With the Flow Framework®, we outlined the four Flow Items that define the type of work done by the value stream. We also set four Flow Metrics® against individual Flow Items and applied Flow Distribution® to a set of Flow Items.

Now that we've seen how we can view measurements of our feedback, we will move on to the Third Way, where we apply Continuous Experimentation and Learning. We will discover the methods for doing so in our next chapter.

Questions

1. What is the optimal number of KPIs for a value stream to have?

   A. 2 to 4
   B. 5 to 7
   C. 6 to 9
   D. 10 to 12

2. Which is not a characteristic of a KPI?

   A. Are valid and can be verified
   B. Encourage desirable employee behavior
   C. Answer questions about progress toward objectives
   D. Are difficult to collect

3. Which is not a DORA metric KPI?

   A. Cycle time

B. Deployment frequency
C. Change failure rate
D. Mean time to repair

4. Which of the DORA metrics' KPIs measure stability? (pick two)

A. Lead time
B. Cycle time
C. Deployment frequency
D. Change failure rate
E. Mean time to repair

5. Which is not a Flow Item in the Flow Framework®?

A. Features
B. Projects
C. Risk avoidance
D. Technical debt reduction

6. Which is not a Flow Metric® in the Flow Framework®?

A. Flow Velocity®
B. Flow Load®
C. Flow Predictability®
D. Flow Time®

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win* by Gene Kim, George Spafford, and Kevin Behr
- `https://f.hubspotusercontent00.net/hubfs/8944057/The%20State%20of%20Value%20Stream%20Management%20Report%202021.pdf` – the State of Value Stream Management 2021 Report from the Value Stream Management Consortium
- `https://kpi.org/KPI-Basics` – a look at defining what KPIs are and how to develop your set of KPIs

- `https://www.devops-research.com/research.html#reports` – landing page for all versions of the Accelerate State of DevOps reports produced by DORA and Puppet Labs
- `https://cloud.google.com/blog/products/devops-sre/the-2019-accelerate-state-of-devops-elite-performance-productivity-and-scaling` – findings from the 2019 Accelerate State of DevOps report by DORA
- `https://cloud.google.com/blog/products/devops-sre/announcing-dora-2021-accelerate-state-of-devops-report` – findings from the 2021 Accelerate State of DevOps Report
- *Project to Product: How to Survive and Thrive in the Age of Digital Disruption with the Flow Framework* by Mik Kersten – the reference for all aspects of the Flow Framework®, including Flow Items and Flow Metrics
- `https://flowframework.org/flow-metrics/` – a look at the Flow Metrics from the Flow Framework® Institute
- `https://www.scaledagileframework.com/devops/` – an article on the `scaledagileframework.com` website, providing, among other things, a description of the DevOps Health Radar
- `https://www.scaledagileframework.com/metrics/` – this article on the `scaledagileframework.com` website illustrates the interplay between KPIs, Flow Metrics, and other forms of assessment

9

Moving to the Future with Continuous Learning

So far, we have looked at value stream management by using the *Three Ways* identified in *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*. By following the First Way, establishing Flow, we looked at determining and mapping our development value streams. In the Second Way, we amplified the feedback loops in our value streams by adopting meaningful KPIs.

We'll now look at the Third Way: continuous learning and experimentation. A critical part of DevOps is the realization that the journey of transformation has no end destination. That is, your organization will continue improving and setting further goals to create a stronger value stream. You will create more and improved future-state value streams as you discover and optimize continuously with no end in sight.

In this chapter, we will look at finding future states for your value stream by discussing the following topics:

- Adopting a continuous learning culture
- Utilizing the Improvement Kata to identify future state value streams
- Implementing all parts of the Lean Improvement Cycle as part of the Improvement Kata

Let's start our exploration by delving deeper into continuous learning.

Understanding continuous learning

In 1990, Peter Senge published *The Fifth Discipline: The Art and Practice of the Learning Organization*. In it, he described the qualities or disciplines that companies need to become learning organizations.

A learning organization allows for learning to develop from the efforts of the people that work for it. This learning facilitates a continuous transformation of the organization so that it can strive for improvement. In today's business environment, the organization that learns quicker than its competitors has a distinct advantage.

Senge identified the following five characteristics or disciplines that learning organizations must have:

- Personal mastery
- Mental models
- Shared vision
- Team learning
- Systems thinking

As organizations work on the first four disciplines, the fifth discipline, systems thinking emerges to take the organization to the next level in becoming a learning organization.

Let's look at each discipline closely to see what's involved in becoming adept at that discipline and moving toward being a learning organization.

Personal mastery

Organizations cannot learn unless the people inside the organizations learn. The individuals inside an organization seek to grow and continually learn. This drive is referred to by Senge as **personal mastery** and is the seed that spreads to the entire organization.

Individuals building personal mastery will discover two artifacts, which play a role in their development:

- **Vision**: The individual learns what is important to them
- **Changes in perspective**: The perception of the current reality as either something that will help or hinder the vision

As the individual develops the vision and sees the current reality through personal mastery, they encounter tensions between the vision and the current reality. These tensions are natural and include *creative tension* or the difference between the vision and the current reality, and *emotional tension*, the emotions the individual feels when viewing the creative tension.

In addition to creative tension and emotional tension, individuals may also be aware of *structural conflicts*. These structural conflicts are the overwhelming feelings of powerlessness or unworthiness when unable to deal with creative and emotional tension.

When encountering the preceding tensions and structural conflicts, the following reactions may appear as coping mechanisms:

- Letting our vision erode to achieve an *easier* goal
- Conflict manipulation where we focus on avoiding what we don't want
- Adopting a *strategy of willpower* where the individual overpowers the tensions, conflicts, and other resistance to achieve the vision

To ensure we develop personal mastery through the strategy of willpower, the individual must be honest and embrace the truth. This allows viewing the creative tension through multiple perspectives and allows several angles of attack to attain the vision. Another beneficial tactic is that learning is done on more than the conscious level. People adept at personal mastery allow the subconscious to have a role. An example of this is the repetition of a new skill to the point that it becomes *muscle memory*.

As individuals develop personal mastery and move toward their vision, a number of the following changes develop:

- Reason and intuition become integrated. This allows the viewing of multiple perspectives.
- Individuals start seeing more connectedness between themselves and the world.
- Compassion starts building.
- Individuals see the whole and begin committing to that whole.

These traits are what organizations need from individuals. To that end, organizations must work toward allowing and encouraging employees to make journeys to personal mastery. They need to create a climate that encourages the creation of visions. They must grant freedom to individuals to inquire and seek the truth. At times, this could involve questioning the status quo. Such freedoms result in better individuals, which leads to growing the organization.

As individuals grow in personal mastery, they may change their visions and their perception of the current reality. This change affects the mental models they've created and used. We'll now look at what those mental models mean.

Mental models

We previously saw that as individuals develop their personal mastery, they encounter creative tension, or they see the gap between their vision and the current reality. That creative tension may arise from their perceptions of how the world works. These perceptions, simply, are the **mental models** used as a discipline for a learning organization.

The mental model shapes the perceptions of the learning individual and serves the following two main purposes:

- It helps the individual in making sense of the world around them
- It informs the individual on how to take action

Mental models inform individuals and organizations of what has worked in the current situation for a specific issue. As such, a learning organization benefits by changing its mental models in the face of new information.

The following things are needed by individuals and learning organizations to add to or change mental models:

- Tools that promote personal awareness and reflective skills
- Infrastructures that try to institutionalize practices, tying them to mental models
- A culture that promotes inquiry and allows challenges to current thinking

Senge identified a few tools to help allow for easy change of mental models. Let's take a close look at these.

Reflective practice

Reflective practice is the act of reflecting while learning in order to discern whether the new information conforms to the existing mental model or whether new mental models must be created.

Reflection is a key tool for adjusting mental models. People that allow reflection do well with this discipline.

Espoused theory versus theory in use

Building up reflective practice skills allows for easy comparisons. One such comparison is whether gaps exist between an espoused theory and the application of that theory.

The following are typical comparisons to determine whether the existing mental models are valid:

- **Questioning leaps of abstraction**: See if what you're looking at is based on fact or is a generalization.
- **Left-Hand Column Analysis**: A comparison of what someone is thinking (written on the left-hand side) with what they say (written on the right-hand side). This exercise created by Chris Argyris allows us to see the differences between what our thoughts are and what was actually said, exposing our preconceptions and biases.

A left-hand column analysis can allow people to understand what they really think and feel and communicate those thoughts in a more transparent fashion. When used properly, this allows conversations to be more productive because of the transparency.

To perform a Left-Hand Column Analysis, find a piece of paper and divide it into two columns. Take a recent conversation and write what was said in the conversation in the right-hand column. Recall your thoughts and feelings about what was said and record them in the left-hand column, lining up the thoughts with what was said.

An example of a left-hand column analysis is shown in the following table.

What I'm Thinking	What Is Said
Doesn't he know I'm busy enough? I'm not sure I can handle any more.	Boss: Can you do me a favor? Me: I suppose. What's the favor?

Really? I bombed 3 months ago! Does he really want to send me to fail again in front of a larger crowd in New York?	Boss: Since you did a great job speaking at the conference 3 months ago, I and the marketing team want to know if you could speak at the conference in New York next month. Me: That does sound fascinating.
How do I get out of doing this?	Boss: Fantastic! Let me send you details about the conference. Me: Sure. Thanks!

Table 9.1 – Left-Hand Column Analysis demonstrating what is thought versus what is said

The ability to regularly examine and readjust mental models is an important enabler of the fifth discipline, systems thinking. Not adjusting mental models in the presence of new information prevents organizations from seeing the whole, and as a consequence, thinking in terms of the entire system.

An example of developing a mental model comes from Scrum. A common practice that Scrum teams follow is estimating the effort of stories using story points. The Scrum team commonly estimates story points using planning poker. In planning poker, the team convenes and collaboratively develops the concept of the *1-point* story, a reference to the smallest amount of effort for completing a story and the basis for relatively comparing other stories against this reference.

During planning poker, the team members convene and are given a set of cards with numbers detailing the story points. The product owner reads the story, and the team members individually select a card with the number of story points. The Scrum master, acting as facilitator, then counts down for all members to reveal their choice at the same time. Those that reveal a different value are invited to explain the reason for their choice. This conversation with the entire team helps build a model of how the team sees work meant for the entire team.

As agreement builds on the mental models in the organization, it becomes the basis for a shared vision. Let's see how important that shared vision is.

Shared vision

In *Chapter 2*, *Culture of Shared Responsibility*, we looked at a generative culture. Remember that a generative culture has its members focused on a shared mission. The shared mission energizes the members, connects the team members together, and gives them the focus to do whatever is necessary.

This shared mission can be thought of as a **shared vision** that Senge calls one of the disciplines for a learning organization. The shared vision is quite simply the answer to the question *just what are we building?*

Many organizations start with an extrinsic or outwardly focused vision such as *beating the other competitors*. The problem with these types of visions is that they tend to be transitory; what happens once you've accomplished your vision? The best visions, ones that will keep learning organizations moving forward in different situations, will be intrinsic, or inwardly focused in nature.

Another type of vision that may not endure as a shared vision is a *negative vision*. A negative vision describes what the organization wants to avoid as opposed to what it wants to be. The differing mindset diverts the energy that the organization needs, robbing the organization of achieving a long-term vision. An avoidance strategy also implies the organization is powerless to change its destiny. These types of visions are only effective in the short term, failing to give the organization any long-term vision.

Shared visions come from individual visions. The source of an individual vision generated by personal mastery is not necessarily from the leader or a predetermined process. Anyone in the organization, by remaining clear on the vision and actively questioning the current reality, can share what they have discovered and invite others to follow.

As the vision spreads, there are a number of reactions that can come when sharing visions and inviting others to follow. These are the following reactions one can expect:

- Enrollment
- Buy-in
- Commitment
- Compliance
- Non-compliance
- Apathy

Of the preceding list, the first three items (enrollment, buy-in, and commitment) are desired reactions that help turn an individual's vision into a shared vision. The other reactions on the list (compliance, non-compliance, apathy) present challenges for attaining the broad-based agreement that is needed for a vision to become *shared* by the organization.

The process of growing a vision from an individual to an organization does include several challenges. An abundance of divergent views may weaken focus and create conflicts in the organization. The gap between the shared vision and the current reality may discourage people in the organization. People may forget that they are part of a collective whole and lose their connection to each other.

Sharing visions may require innovative ways to collaborate. In this article from the Harvard Business Review (`https://hbr.org/2011/07/are-you-a-collaborative-leader`), Marc Benioff, the CEO of Salesforce, Inc invited all 5,000 employees to a *virtual* off-site management meeting. The results were immediate: conversations erupted that involved the entire company. The dialogue continued for weeks and contributed to a more empowered, mission-driven company.

Other advice for sharing a vision comes from a blog article written by Veena Amin of Empuls (`https://blog.empuls.io/organizational-vision/`). In the article, she includes the following advice:

- **Unify the organization**: The leader's job is to find and bring together all parts of the organization to communicate the vision.
- **Engage everyone**: Once everyone is brought together, the leader should communicate with everyone, even if their role is not seen as important or central to the organization.
- **Set context**: Good leaders find a way of connecting the vision to the current state of the organization.
- **Shift control**: Refinement of the vision involves a collaborative approach, and that often means sacrificing control. Some control at the organizational level may have to be removed to allow this to happen.

In working through these challenges while creating a shared vision, the organization improves other disciplines. The shared vision will approach the ideal reality through personal mastery. The process of personal mastery changes the mental model of the organization. We will see that the process to set personal mastery in motion to accomplish the mental model change to a shared vision is team learning. Let's examine what team learning entails.

Team learning

When looking at the journey of personal mastery from the current reality to a shared vision, individuals need to undertake that journey through learning. In the beginning, that learning is done on an individual basis. Effective groups eventually pool together and learn as a collective. This convergence is **team learning**.

Achieving team learning is not done through group training. The primary mechanism of team learning comes from opportunities for organizations to have frequent discourse. This discourse can take the following formats:

- **Dialogue**: This type of discourse allows the organization to reach a common understanding
- **Discussion**: This type of discourse is usually the exchange of different points of view, followed by examination, usually to see which point of view prevails

In both formats, the organization tries to look beyond an individual's understanding to reach a common understanding. Multiple points of view are often exchanged. The key difference between dialogue and discussion is that dialogue allows for alternatives to emerge, while discussion has a single point of view: *winning*.

Senge talks of three necessities for having dialogue as part of achieving team learning:

- The ability to take assumptions, examine them, and move beyond them

- All participants are able to look at each other as colleagues and equals
- Having a dedicated facilitator that can hold the context of the dialogue for the group

With the preceding components, a leader can keep the group learning through dialogue. They help the group maintain ownership of its objectives.

Another mechanism for team learning is the awareness of defensive routines. Defensive routines may emerge as groups become aware of the gaps between the current reality and the shared vision. A good facilitator can recognize defensive routines and deal with them in the following manner:

- Inquire directly with the organization about the causes of the problem
- View defensive routines as a sign that team learning is not occurring

An example of team learning is mob programming. This practice was developed by Woody Zuill as an extension of the practice of pair programming. In mob programming, one member of the team takes the role of the driver, while the other team members assume the role of navigators. The driver performs the main task, often writing code, while receiving feedback from the navigators. After a set period of time, the driver role rotates to another team member.

Mob programming is effective as a team learning method because, as the team collaborates, they learn and share new insights at the same time.

As teams learn together and gain a common understanding of the shared vision and change mental models, the fifth discipline, systems thinking emerges. Let's look at this final discipline, which allows a learning organization to really flourish.

Systems thinking

Systems thinking is the change in perspective that learning organizations attain through proper practice and skill development of the other four disciplines. Learning organizations can look at themselves and see the components and interconnections.

Systems thinking comes from the culmination of effort in starting and improving on the other four disciplines (personal mastery, mental models, shared vision, and team learning). At certain points, the organization achieves a certain level of maturity in the discipline to expand into systems thinking.

When individuals are tuned to their individual personal mastery, they start seeing the whole and their role as part of that collective whole. This awareness is a necessary part of systems thinking.

The awareness of the collective whole affects the mental models that individuals have established. The individual mental models transition to a shared collective mental model of the learning organization with roles defined.

The change to the collective mental model paves the way for a shared vision for the organization. The members of the organizations become committed to this shared vision. They work together in dialogue with a team leader to learn how to move the shared vision throughout the organization and expand that vision to include everyone in the learning organization.

A way of visualizing systems thinking is the iceberg model. As detailed in this article from ecochallenge.org (`https://ecochallenge.org/iceberg-model/`), systems thinking resembles an iceberg, where only 10% is visible above the water and the 90% that is below the surface affects the part that is visible.

The iceberg model details the following four levels:

- **The event level**: This is the only visible layer in systems thinking. This represents our perceptions of the outside world.
- **The pattern level**: This level attempts to explain our perceptions at the event level.
- **The structure level**: This level explains the causes of the patterns we are observing.
- **The mental model**: This represents attitudes, beliefs, and assumptions that create the structures.

As a learning organization that employs the five disciplines, members work and learn more about themselves and how to improve in a never-ending quest of discovery. The organization incorporates new learning regularly to continuously improve. Let's see what that journey looks like through the use of the **Improvement Kata**.

Applying the Improvement Kata

The Improvement Kata is one of the patterns that came from the Toyota Production System and is described in detail in Mike Rother's book *Toyota Kata: Managing People for Improvement, Adaptiveness, and Superior Results*. In the Improvement Kata, we follow a path toward improvement by examining the following four steps:

1. We envision our ideal future state.
2. We examine our present state or condition.
3. We determine the next target that brings us closer to the ideal future state.
4. We run an experiment, evaluating and learning in a **Plan-Do-Check-Adjust** (**PDCA**) or Lean Improvement Cycle to see if the results bring us closer to the ideal future state.

An illustration of the four steps of the Improvement Kata is as follows:

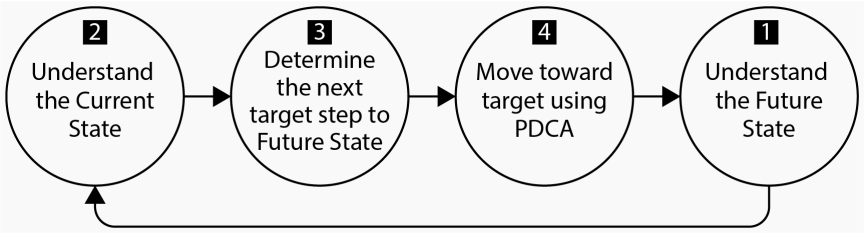


Figure 9.1 – Improvement Kata

An example of teams using the Improvement Kata is an Agile *retrospective*. A retrospective is a meeting held at regular intervals where the team considers the following questions:

- What has gone well?
- What has not gone well?
- What actions should we take (including keeping things that work) to improve?

The first two questions allow exploration of the current state and the future state. The third question sets in motion actions to move the team to its ideal future state. In Scrum, these actions are added to a team's backlog of work so that the team can work on them. In this way, the team is moving toward the ideal state using the PDCA cycle of learning.

Moving to the ideal future state target by target requires continuous experimentation. This experimentation is done in learning cycles that follow the PDCA format. Let's look at what happens in each **Lean Improvement Cycle**.

Closing the Lean Improvement Cycle

Continuous improvement is at the heart of Lean thinking. Improvements should be built into future efforts and measured for their effectiveness.

For our value stream to have this approach, it must view its efforts in terms of a cycle of learning. The most popular model of this cycle is the Lean Improvement Cycle or **PDCA** cycle.

This cycle was associated with W. Edwards Deming, who called it the Shewhart cycle after Walter Shewhart. This cycle is illustrated in the following diagram:

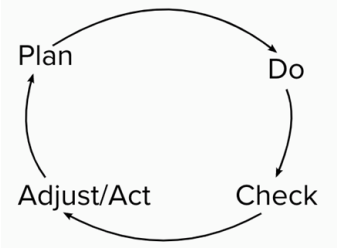


Figure 9.2 – PDCA cycle

As shown in the preceding diagram, the cycle has four phases of activities:

1. **Plan**: Determine what iterative step to take. This may be a step in a backlog that is ranked high in priority. Formulate the hypothesis of what outcomes may happen if this step is implemented.
2. **Do**: Add the step to your value stream's workflow.
3. **Check**: Examine (or keep examining) the metrics you measure for your value stream.
4. **Act** (**or Adjust**): Did performing the step achieve the hypothesis? If so, keep the step. If not, you may need to pivot by formulating another hypothesis.

This cycle can be performed over both short and long durations of time. In SAFe®, a team practicing Scrum on an ART will perform this cycle every sprint to improve the team and work with the other teams on the ART to perform this cycle every program increment or PI.

Summary

In this chapter, we looked at how value streams can follow the Third Way by adopting continuous learning and experimentation. We saw how this starts by looking at the journey organizations take to become learning organizations. They hone their learning through the five disciplines of personal mastery, mental models, shared vision, team learning, and ultimately, systems thinking.

An approach for learning organizations to move toward their shared vision is to follow the Improvement Kata. In the Improvement Kata, after the desired future state is determined, the current state is identified. The value stream then identifies a target and an experiment to run. The experiment is run in a PDCA cycle, and the learning is applied to see if the value stream is closer or farther from the desired future state.

A Lean Improvement Cycle or a PDCA cycle is used in the Improvement Kata as the framework for experimentation. The value stream plans the experiment, runs through the experiment, and checks to see if the results validate the experiment's hypothesis. Based on the results, the value stream will make adjustments.

This completes *Part 2* of this book, where we saw how to align our work in value streams, how we measure the effectiveness of the value stream at delivering value, and how, based on the value stream's metrics, we proceed down a path of continuous learning to improve the value stream. In *Part 3*, we will look at how a value stream in SAFe, embodied as an ART, performs the tasks of delivering value through the marriage of process and technology known as the Continuous Delivery Pipeline.

Questions

Test your knowledge of the concepts in this chapter by answering these questions.

1. In personal mastery, what creates tension with the vision?

   A. Virtual reality
   B. Current reality
   C. Desired reality
   D. Past reality

2. What types of discourse facilitate team learning? (Choose two)

   A. Conflict
   B. Dialogue
   C. Lecture
   D. Discussion
   E. Soliloquy

3. Which is the first step in the Improvement Kata?

   A. Determine the ideal future state
   B. Identify the present state
   C. Identify a target
   D. Execute PDCA cycles to iterate to target

4. What does the *C* represent in the PDCA cycle?

   A. Conclusion
   B. Cease
   C. Check
   D. Capital

Further reading

- *The Phoenix Project: A Novel about IT, DevOps, and Helping your Business Win* by Gene Kim, George Spafford, and Kevin Behr – We've used this book and its Three Ways to discuss how value streams are created and maintained.
- *The Fifth Discipline: The Art & Practice of the Learning Organization* by Peter M. Senge – The reference when learning about how to be a learning organization.
- `https://valshebnik.com/blog/left-hand-column/`: This article provides an excellent description of Left-Hand Column Analysis, including definition, examples, and dangers.
- `https://hbr.org/2011/07/are-you-a-collaborative-leader`: This article looks at how leaders can collaborate and share with their companies.
- `https://blog.empuls.io/organizational-vision/`: This article looks at sharing an organizational vision, the benefits, and tips and tricks to do so.
- `https://ecochallenge.org/iceberg-model/`: An article describing the Iceberg Model, which illustrates Systems Thinking.
- *Toyota Kata: Managing People for Improvement, Adaptiveness, and Superior Results* by Mike Rother – Detailed method for the Improvement Kata.

Part 3: Optimize – Enabling a Continuous Delivery Pipeline

In SAFe®, the Continuous Delivery Pipeline is a marriage of processes and automation that allows value streams, implemented as **Agile Release Trains** (**ARTs**), to develop and deploy following a cadence, but release on demand when it suits the organization.

We will start our exploration of the Continuous Delivery Pipeline with Continuous Exploration. With Continuous Exploration, we will set up our ideas for new solutions or enhancements as experiments to elaborate, research, and prioritize for PI planning.

In Continuous Integration, the next stage of the Continuous Delivery Pipeline, we will develop our experiments. We will begin the automation process after coding to allow testing and packaging if testing succeeds.

During Continuous Deployment, we will strive to deploy our solutions to a production environment. We will strive to deploy these changes without disrupting our production environment and customers, as well as to allow testing in production. We will discuss methods for performing the separation of deployment and release in *Chapter 12*.

Finally, we will Release on Demand. This allows us to deliver value to our customers. We will monitor the release, ensuring it validates our experiment's hypothesis, with no adverse effects on the production environment in terms of security or availability.

This part of the book comprises the following chapters:

- *Chapter 10, Continuous Exploration and Finding New Features*
- *Chapter 11, Continuous Integration of Solution Development*
- *Chapter 12, Continuous Deployment to Production*
- *Chapter 13, Releasing on Demand to Realize Value*

10

Continuous Exploration and Finding New Features

In Continuous Exploration, product management works with people both inside and outside the **Agile Release Train** (**ART**) to find features that will provide value to the customer, explore the needs and wants of the customer, verify the feasibility of new features in the current architecture, and prepare new features to be developed by the ART.

As we can see in the following illustration, Continuous Exploration, the first stage of the Continuous Delivery Pipeline, establishes the *trigger* for subsequent development:

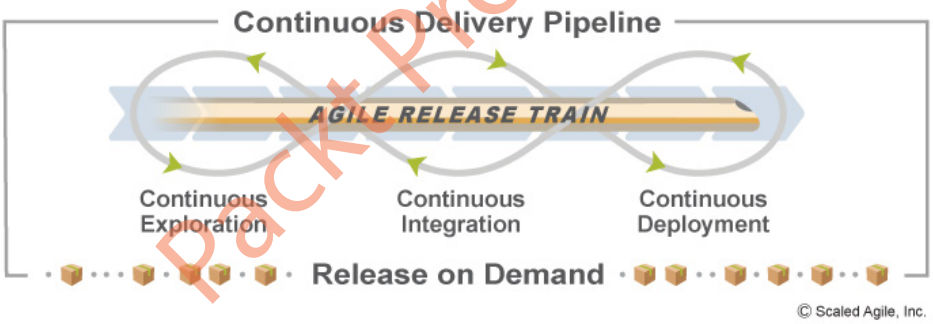


Figure 10.1 – Continuous Delivery Pipeline (© Scaled Agile, All Rights Reserved)

In a nutshell, the following activities will be discussed in this chapter:

- Hypothesizing the customer value
- Collaboration and research
- Discussions about architecture
- Synthesizing the work

The work that product management does during Continuous Exploration is important for the ART in that it helps set the context for the ART to execute the shared mission or vision. Let's view this work that product management performs to prepare for upcoming **Program Increments** (**PI**).

Hypothesize customer value

"*If I asked people what they wanted, they would have said faster horses.*" This quote, commonly (and possibly incorrectly) attributed to Henry Ford, highlights a problem that product managers have when looking for new features or new products. Customers may simply not know what they want or cannot imagine innovations that may come from different approaches or out-of-the-box thinking.

A way of working through the unknowns in product development is to take the approach highlighted in Eric Ries's book, *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. In this book, Ries shows a way of collaborating with the customer for iterative product development. Some of those ways are captured in the learning cycle he proposes, the **build-measure-learn cycle**.

The build-measure-learn cycle is an iterative product development cycle where Lean Startups discover the product and feature qualities that resonate with their customers through experimentation. The following parts of the cycle are done with the participation of the customer:

- **Build**: Often, what emerges in the first iteration is a **Minimum Viable Product** (**MVP**) – something that kicks off the learning process with the customer
- **Measure**: Conversations with the customer or application of metrics determine what is working and what is not
- **Learn**: Armed with the knowledge from previously collected metrics, the choice is made to pivot or persevere

The build-measure-learn cycle is illustrated in the following diagram:

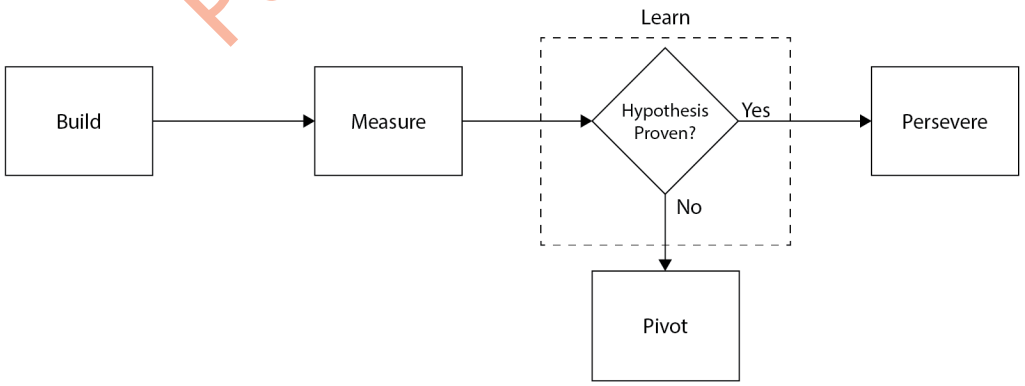


Figure 10.2 – Build-measure-learn cycle

The build-measure-learn cycle follows the same cycle as the Lean Improvement or PDCA cycle discussed in *Chapter 9*, *Moving to the Future with Continuous Learning*. Building takes place in the *plan* and *do* phases of the PDCA cycle. Measuring allows us to *check* in the PDCA cycle. Once we've performed the Measure step, we *learn* (or *adjust*) by pivoting or persevering.

With the three parts of the build-measure-learn cycle identified, let's examine the first cycle and its use of the MVP.

Building with an MVP

Creating an MVP is the initial step of the learning journey that Lean Startups take. It provides a check for these startups to determine whether they are taking the correct path. While different people have different ideas of what constitutes an MVP, including SAFe® (which we will explore later), let's examine the definition proposed by Eric Ries in *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*.

According to Ries, an MVP doesn't necessarily have to be an actual product or the product in its final form. The only thing necessary is that it conveys to the customer an indication of the product or service.

The following examples are cited by Ries as examples of MVPs:

- Dropbox created a video to help customers understand the idea of cloud-based file storage and synchronization and explain the advantages it had over competitors
- Groupon began as a WordPress blog where customers would email asking for a PDF coupon of the offering presented as a new blog post
- Food on the Table, a startup based in Austin, started their service of aggregating ingredients from customer shopping lists and recipes and fulfilling purchases at local grocery stores by closely working in person with their first customer and defining what the service was and how it worked

These examples show that an MVP is constructed to answer the following questions:

- What should our product or service look like?
- Can our product or service resonate with customers?
- Can we provide this product or service?

Startups find the answers to these questions after creating the MVP by talking with customers and applying innovation accounting to metrics that are important to them. Let's take a look at the measures used in innovation accounting.

Measuring with innovation accounting

An MVP is created to prove or disprove what Ries calls a "*leap of faith assumption*." When looking at the build-measure-learn cycle, that MVP goes through three milestones. These milestones help evaluate the progress of the MVP and illustrate good inspection points for learning:

1. **Establishing the baseline**: The creation of the MVP is a baseline used to see whether the "*leap of faith assumption*" is valid. This assumption can also be thought of as a hypothesis that an experiment (the MVP) can validate.
2. **Tune the engine**: Based on objective data, we should look at making adjustments that move us closer to the goal.
3. **Pivot or persevere**: These adjustments, based on the data we collect, may prompt us to continue the path we are on and add further refinement, or pivot. Pivoting may lead us to take different paths for the product or service.

For the objective data, we look to avoid vanity metrics. We previously discussed the criteria Ries originally wanted for his metrics in *Chapter 5*, *Measuring the Process and Solution*. The following qualities are repeated here:

- **Actionable**: Does the metric show clear cause and effect? In other words, could you replicate the results by performing the same actions?
- **Accessible**: Does everyone on the value stream have access to the same data and is that data understood by all?
- **Auditable**: Is there credibility to the report?

Of course, the best data points will come from direct consumer feedback. Often, in many situations, if you cannot understand what the metrics are telling you, you may have to resort to interviewing your customers.

With the objective data collected from true metrics or customer feedback, then comes the necessary decision to ensure the continuity of the startup. Do we need to move in a different direction for our MVP (pivot) or should we continue in this direction and add more enhancements (persevere)? Let's examine the factors involved in that decision.

Learning to pivot or persevere

Based upon customer feedback or objective data, the Lean Startup has a question to answer: does the current direction of our product or service allow us to meet our goals and provide value to the customer? If it does, then we continue on the same path for that product, adding additional features. If not, we need to move in another direction, or pivot, without remorse.

A pivot may be a change to one or more aspects of the product or service. Ries cites the following pivots used to allow a product or service to provide better value:

- **Zoom-in pivot**: One feature of the product or service resonates with customers far more than any other feature of the product or service. You then focus on that feature, making it the product or service.
- **Zoom-out pivot**: The product or service doesn't provide enough customer value on its own but bundled as a feature of another product or service, it may prove its worth.
- **Customer segment pivot**: The product or service meets the needs of all the customer and just not of the customer the product or service was originally intended for.
- **Customer need pivot**: While working with your customer, you may discover that you can provide better value through a different product or service. An example of this is Potbelly Sandwich Shops, which started as an antique store that provided food to its customers.

With most types of pivots, the product or service is changed, often to the point that it does not resemble the original product or service, but there may be some pivots where the MVP, product, or service is abandoned.

When facing the prospect of any type of pivot, it's important to stay objective on the decision to pivot or persevere. Many startups often fail because they don't pivot or pivot too late.

Now that we've looked at how we create our MVP and validate our hypothesis with the build-measure-learn cycle, let's look at how SAFe® has adapted Build-Measure-Learn and innovation accounting into the SAFe Lean Startup Cycle, which applies experimentation to the execution of epics.

The SAFe® Lean Startup Cycle

In SAFe, an **epic** is a significant product development effort that is not scoped into a specific timebox. Epics describe the long-term changes an organization may want to make to a product. ARTs use epics as experiments to guide possible product development.

Note that while an epic and a project may have similar definitions, an epic is flexible with its scope. A project starts with an established start and end date and a fixed scope where all requirements must be met through the completion of tasks that build the deliverable. An epic really forms the basis for an experiment that may or may not run to completion.

An epic is described by a **Lean business case**, a brief document that outlines the need for the epic, possible solution alternatives, and a proposal for an experiment. The experiment is written in the form of an epic hypothesis statement that describes the proposed value, a hypothesis of the business outcomes, measurements of the experiment through leading indicator metrics, and any **Non-Functional Requirements** (**NFRs**) that may act as constraints. The MVP is included in the Lean business case as the implementation of the experiment.

The hypothesis statement of an epic sets the tone for the experiment by outlining the proposal, its potential benefits, proposed measures, and constraints. The following example of an epic hypothesis statement details a proposed pizza drone delivery service:

Epic Description	• FOR customers that live in urban areas… • WHO desire quick and convenient pizza delivery, • THE PizzaBot 2022… • IS a drone-based autonomous pizza delivery system… • THAT easily delivers pizza from the restaurant quickly and easily. • UNLIKE current automobile-based pizza delivery, which is the standard, • OUR SOLUTION reduces overhead costs by using cheaper electricity instead of gasoline.
Business Outcomes	• Better customer experience with quicker delivery times (and hotter pizza) • Lower overhead costs by reducing delivery drivers and saving on fuel
Leading Indicators	• Reduction in average delivery time • Reduction in overhead costs • Higher NPS survey scores
NFRs	Must comply with local ordinances regarding commercial aerial drone use

Table 10.1 – Example epic hypothesis statement

An MVP in the epic differs from Ries's definition. An MVP in this sense refers to the minimum set of features that are meant to form an initial product to be used by customers. These features would be developed by the ART.

Leading indicator metrics are used to validate the hypothesis of our experiment. They serve to tell us whether we are venturing down the correct path. We want our metrics to be actionable, accessible, and auditable so that they are not vanity metrics. We also want them to be true leading indicators, metrics that are reliable indicators of the possible value at the earliest possible moment, without waiting for trends to emerge.

The SAFe Lean Startup Cycle does model itself on the build-measure-learn cycle proposed by Eric Ries. In the SAFe Lean Startup Cycle, the ART works with the epic in the following manner:

- **Build**: The MVP is developed and released to the customer
- **Measure**: The leading indicator metrics defined in the Lean business case of the epic determine the response from the customer
- **Learn**: Based on the leading indicator metrics, the decision must be made to persevere and continue developing features beyond the MVP; pivot, finish the epic as is, and create a new epic with a new hypothesis; or stop the epic so that no development happens beyond the MVP

The SAFe Lean Startup Cycle is shown in the following diagram, outlining the path of the epic and the possible paths that may occur based upon the validation of the hypothesis:

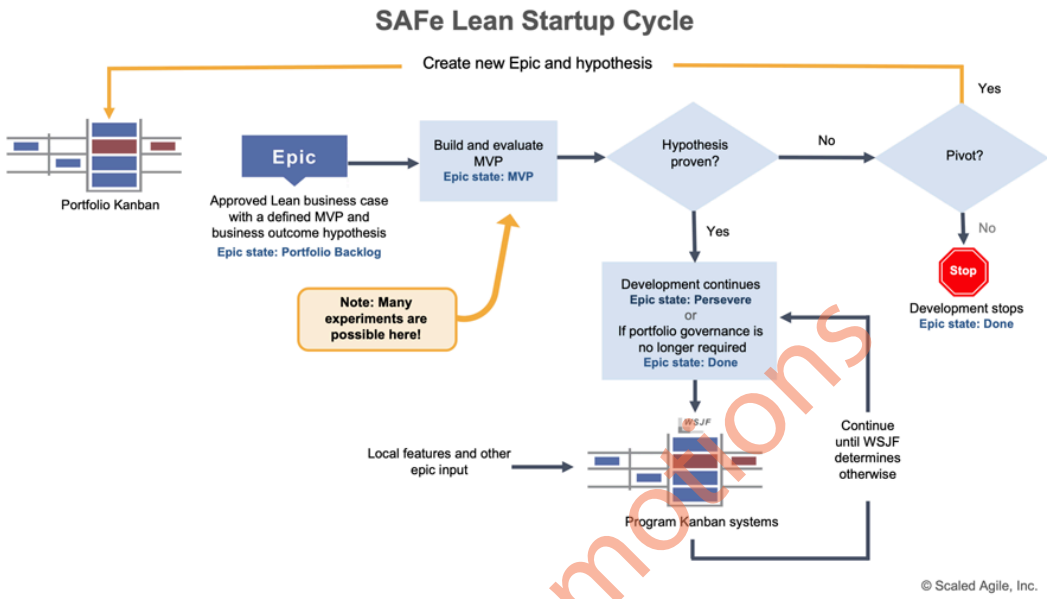


Figure 10.3 – SAFe Lean Startup Cycle (©Scaled Agile, All Rights Reserved)

The result of this activity is a backlog of epics. Each epic outlines an experiment containing a hypothesis statement of potential value and an MVP that allows us to carry out the experiment.

Using the SAFe Lean Startup cycle based on the build-measure-learn cycle, product management sets up a hypothesis of value and an MVP as an experiment to validate the hypothesis, but product management doesn't do this alone. They collaborate with others to refine both the hypothesis and the experiment in the Build portion of the cycle. We will examine this collaboration in the next section.

Collaboration and research

Product management requires input from different people, each with a unique perspective on what the solution should fulfill. Good product managers know that they must work together with these people and discover the qualities that can form the basis of a benefit hypothesis or the features that an MVP must have.

Here, we will look at the two aspects good product management needs to form the MVP. The following aspects form the basis of the activities product management does to elaborate on the MVP:

- Collaboration with customers and stakeholders
- Research to elicit product qualities and NFRs

Let's begin with the primary collaborations that product management coordinates.

Collaboration with customers and stakeholders

The best products emerge from teams. This is true from the early phases to the design, implementation, and testing stages – finally leading to release.

Product development collaborates with the following individuals to define the features of a product:

- Customers
- System architects or engineers
- Business owners
- Product owners or teams

Let's examine the relationships created by the collaboration of product management and these groups.

Customers

The customer is the final arbiter of value. They are, after all, the ones for whom you are building your product. Their input is the most direct source of feedback on whether the product is meeting their needs.

In addition to customers that may not know what they want in a solution, product management must pay attention to customers that only focus on making incremental changes, as that may not contribute to the product's long-term strategy.

System architects

System architects know the most about the product from an architectural standpoint. They understand the capabilities as determined by the enablers as well as the constraints, identified by NFRs.

As important as it is for product management to collaborate with system architects to understand the balance between new features, development for the long term using enablers, and maintenance and paring technical debt, it is just as important that system architects understand customer needs and concerns. For that reason, close collaboration between product management, system architects, and the customer is paramount.

Business owners

Business owners are the key stakeholders from the organization's point of view. They need to make sure the solution developed by the ART aligns with the mission and the overarching strategy of the organization.

Product management collaborates with business owners to understand the prioritization of features that the ART may work on.

Product owners and Agile teams

The Agile teams on the ART do the work of developing, deploying, releasing, and maintaining the solution. A key person on each Agile team is the product owner, who acts as the content authority, helping the team elaborate stories and acceptance criteria and accept stories as done.

Because teams are closest to the work and actual implementation, their insights into product and user concerns should not be ignored. Good product managers will accept this feedback from the Agile teams.

We now know the different roles that product management collaborates with and receives feedback from. At this point, we should examine the forms that this feedback will take.

Research activities

Product management collaborates with the customer, business owners, and product owners by using the following types of research activities to gain insight into customer needs and how the product will enable value:

These types of activities can be classified as the following:

- Primary market research with the customer
- Gemba walks and customer visits to see the customer experience
- Secondary market research to further delve into the customer's mindset
- Lean UX to establish experiments

Let's examine each of these activities.

Primary market research

Primary market research features direct collaboration between product management and the customer. This direct collaboration may involve the following methods:

- Focus groups
- User surveys or questionnaires
- Innovation games

Focus groups, surveys, and questionnaires ask direct questions about the product or service. They may inquire about possible future needs, but sometimes the customer can't imagine beyond the short-term use of the product.

This is where innovation games come in. innovation games are several activities described in *Innovation Games: Creating Breakthrough Products Through Collaborative Play* by Luke Hohmann. In the book, games allow for the discovery of unspoken needs, how your customers look at success, and where your products fit with the customer.

Primary market research may often be done at the organization's premises, but other insights can be gathered elsewhere. This is where Gemba Walks and customer site visits come into play. Let's look at them now.

Gemba walks

Genchi genbutsu means "*go see and understand*" in Japanese. A Gemba walk is an activity used to experience *genchi gembutsu*. It was first used in the Toyota Production System and is a staple in Lean thinking. On a Gemba walk, people go to where the product is used to see the actual environment.

An example of a Gemba walk comes from *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses* by Eric Ries. The design of the 2004 Toyota Sienna minivan was led by Yuji Yokoya. He had little experience in North America, the target market for the Sienna. So, he proposed an undertaking: a road trip of the provinces of Canada, the fifty states of the United States, and parts of Mexico in a current Sienna minivan while interviewing customers.

Yokoya discovered that North American customers took more long-distance car trips than customers in Japan. Another finding was that minivans needed to cater to the passengers that typically occupied two-thirds of the vehicle: the kids. Based on this data, Yokoya added features that had more *kid appeal* and would accommodate long-distance trips.

The selection of these features had a profound effect. Sales of the 2004 Sienna model were 60 percent higher than the previous year's model.

Secondary market research

Secondary market research is activities that do not involve direct collaboration with the customer. These activities can help get an understanding of the customer and the market.

Some of the activities that allow you to learn about a customer's wants and needs include the following:

- Creating a persona, a fictional representation of your customer
- Understanding your customer's thoughts and emotions using empathy maps
- Examination of your customer's journey, including sentiments, using journey maps

These artifacts can be refined and amended when meeting with the customer when available.

Lean UX

When developing features, we want to use a similar PDCA learning cycle as Build-Measure-Learn. An incremental learning cycle such as this one helps us refine our epics into features.

One cycle of this kind comes from the book *Lean UX: Designing Great Products with Agile Teams* by Jeff Gothelf and Josh Seiden. In the book, they talk about **Lean User Experience** (**Lean UX**), a mindset and process to incrementally discover product features and validate customer value.

Scaled Agile has adapted the process model for use beyond **User Interface** (**UI**) and **User Experience** (**UX**) teams. The following diagram from Scaled Agile illustrates the process:

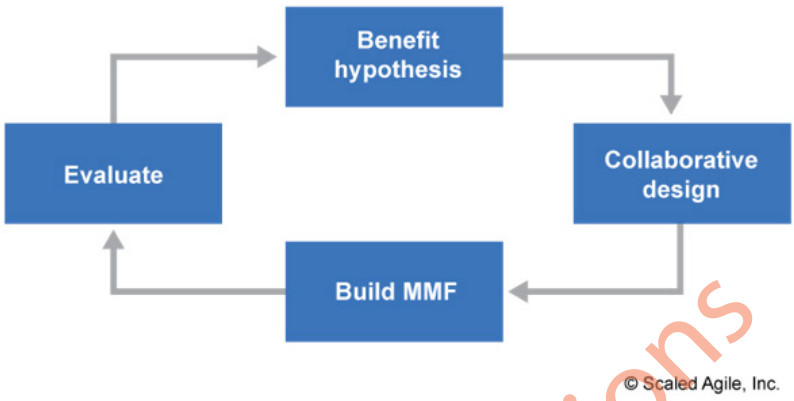


Figure 10.4 – Lean UX process diagram (© Scaled Agile, Inc. All rights reserved)

Let's examine each of the steps of the process.

Constructing a benefit hypothesis

It is impossible to know, at the start of development, what features will delight the customer in the face of unknowns in the environment and risks. The first part of this incremental design cycle looks to establish a hypothesis of the intended measurable business result of the feature if the feature is developed and released. This benefit hypothesis may be related to an epic's hypothesis if this feature is part of an epic's MVP or is part of further development of the epic.

Collaboratively working on the design

With a benefit hypothesis in hand, it is up to members of the ART (product management, system architects, business owners, product owners, and the Agile teams) and the customer to work together and generate artifacts that work as design elements for the product.

Building the Minimum Marketable Feature

A **Minimum Marketable Feature** (**MMF**) is the minimum amount of functionality a feature contains to prove or disprove a benefit hypothesis. The ART may iteratively implement this so they can learn about their progress toward the benefit hypothesis.

Sometimes, the MMF may be a lightweight artifact with no functionality created to generate customer feedback, such as a prototype or wireframe. Other times, the MMF may be developed and released so customers can evaluate and give their feedback.

Evaluation

The MMF is released and we wait to see how the customer reacts. We can collect objective data through observation and A/B testing. We can also query the customer through surveys.

Based on the data, we can decide whether or not the benefit hypothesis was proven. This may allow us to continue development, refactor, or even pivot to abandon the feature.

The results of the collaboration and research activities allow us to understand our customer's needs and to design the features to those needs. The following artifacts may be generated by this activity:

- An understanding of the customer needs
- Style guides
- Logos
- UI assets
- Prototypes
- Mockups or wireframes
- Personas
- Customer journey maps

As product management works to understand the features of the product, the system architect needs to understand the product's architecture and which enablers are required to keep the product features flowing. Let's examine the system architect's role in Continuous Exploration.

Architecting the solution

As the maintainer of a product's architecture, the system architect keeps track of the capabilities of the product and enhances them through the creation of enablers and understands the constraints of the system as identified by NFRs.

Working with others on the ART or others in the organizations, the system architect will explore the following aspects of the product to ensure that NFRs are satisfied:

- Releasability
- Security
- Testability
- Operational needs

Let's examine these aspects in further detail.

Architecting releasability

It is often desirable to release new features to the customer at the organization's discretion. We still want a deployment to a production environment as part of the development cadence, but the actual release becomes a business decision. For this reason, we look to separate the deployment from the release.

Architecture may play a key role in allowing the separation of the deployment from the release. Separating deployment and release relies on technology such as feature Flags, which the architecture must accommodate. These feature Flags easily allow the Continuous Deployment of new features without disrupting the current functionality when switched *off*. A new feature is considered released when the Feature Flag is switched *on*. Applications of Feature Flags include canary releases and dark launches. An architecture with loosely coupled components allows each component to have its own separate release schedule. This allows components that require different release strategies to have them.

Ultimately, release strategies are usually related to an organization's strategy or business objectives. Releases may need to be done to respond to the marketplace or outmaneuver a competitor. The ability to allow for a flexible release can be a competitive advantage. We will see the benefits of this foresight in *Chapter 13*, *Releasing on Demand to Realize Value*.

Designing security

Although DevSecOps places an emphasis on *shifting left* to test for security vulnerabilities and concerns, the DevSecOps approach begins here as architects look to incorporate security as new features are drawn up. This allows security to be included from the very beginning, as opposed to being considered an afterthought.

To ensure security concerns are included during the initial design, the following practices are performed:

- **Threat modeling**: Looking at the current product's infrastructure, architecture, applications, and proposed features to identify possible security vulnerabilities, attackers, and attack vectors.
- **Compliance management**: Ensuring that the product complies with known industry-based security regulatory standards, such as the HIPAA, FedRAMP, and PCI. The requirements in these standards primarily deal with security and privacy.

Outputs from these practices are usually maintained and communicated to the ART as NFRs. NFRs are constraints on the work that the ART develops as features. These constraints should be part of the continuous testing suite as development proceeds.

Ensuring testability

Testing is the primary way of ensuring the correct function, quality, security, and readiness for deployment of a feature. If a solution is not testable, the ART has no idea of its progress or whether value can be captured.

We saw before that a loosely coupled architecture allows for flexibility through the ability to release components on different schedules. This flexibility extends to designing systems that allow more testing to occur. Systems whose components have well-defined interfaces allow for more frequent system-level testing by having components that are not ready for integration with other components, replaced with dummy code or logic that returns valid outputs.

Existing legacy architectures can evolve to a modern loosely coupled **Application Programming Interface** (**API**)-based architecture by adopting the **Strangler pattern**. The Strangler pattern, as coined by Martin Fowler, takes a legacy monolithic system and establishes a facade interface to those entities communicating with the legacy system. New code is incrementally written to replace pieces of the interface. Eventually, all the functions of the legacy system are replaced by the new code.

Another design aspect to consider for testability is the ability to execute automated tests at various levels from individual code functions to user stories – and ultimately to features. Automated testing allows for tests to be executed more frequently. Frequent testing allows for greater confidence in the quality and security of the system.

Testable architecture has benefits for the Agile teams that do the implementation work. **Test-driven development** (**TDD**) looks to create the tests first, forming an initial understanding of the behavior of the system. **Behavior-driven development** (**BDD**) continues the understanding of the system's behavior at higher levels.

Maintaining operations

An architecture's design considerations should not change once the feature has passed all its tests. The system architect must ensure that the architecture operates easily in non-production or staging environments, as well as the final production environment.

The first aspect of this is measurability. The architecture should allow the staging and production environments to monitor its resources to determine whether any adverse performance is present with the active system. Alerts should be designed when thresholds are violated. This ability to measure resource use from low levels to higher levels is commonly referred to as full-stack telemetry.

Another aspect is the ability to record all measures into logs for easy retrieval when incidents occur. Architects should understand what aspects of the architecture can yield insights captured as logs so that operations personnel can add these measurements into a logging tool that includes timestamping and search capabilities.

Releasability plays a role in ensuring that architecture allows for easy operation. Feature Flags, a common tool for separating deployment into production environments and the release to a customer, may work as a mechanism for rollback by deactivating the affected feature in the case of an incident. Another rollback mechanism to consider is the establishment of blue/green deployment in production environments. Reliable automation in the CI/CD pipeline, including robust automated tests, can enable *fix-forward* situations where incident fixes can be rolled into production as soon as possible.

Outputs of the architect's work include solution intent, an idea of the minimum architecture needed to prove the benefit hypothesis of an epic, and the NFRs, which serve as constraints on all features and stories that come from the epic.

At this point, the initial thinking about the feature in terms of thoughts on the value and effects on the architecture has been performed by product management and the system architect. Others must bring their contributions into play, from further refinement and prioritization to getting the feature ready for PI planning. Let's examine those steps now.

Synthesizing the work

Product management has gained knowledge and collected additional research on the customer's needs that may contribute to anticipated value. The system architect has looked into ensuring that architectural support already exists or may be forthcoming in an upcoming enabler. Product management collaborates with business owners, product owners, Agile teams, and others to complete the following activities:

- Complete the definition of the feature
- Use BDD to outline the acceptance criteria
- Prioritize the feature on the Program Backlog using **Weighted Shortest Job First** (**WSJF**)
- Prepare for PI planning

The goal of synthesis is to ready the ART for the upcoming PI. For this, we will create the following artifacts:

- A clear vision of what the ART will develop
- A roadmap that details the product's evolution by showing when possible solutions will be delivered
- A backlog of defined features

Let's examine these activities done during synthesis in detail.

Completing the feature

Remember that we started with epics that were significant product efforts conducted as experiments. We started our experiment by developing the MVP. The MVP will be a set of features that validates the benefit hypothesis of the epic. Using Lean UX, we further define the MVP into MMFs that initially began as a *what-if* conjecture after learning about the customer's needs and wants. Research from product management and the system architect added more detail. It's now time to complete the refinement so the rest of the ART can take over.

A good feature will have the following three parts: beneficiaries, a benefit hypothesis, and acceptance criteria. Let's discuss these in greater detail.

Beneficiaries

While we have been thinking about the value to the organization or the customer, we now must consider the end user of our product or service. Sometimes, the end user is not the same person that purchases the product or service. In these cases, we may need to challenge customer assumptions of what the needs and wants of the end user are.

Benefit hypothesis

We want to include the benefits to the customer or organization if we develop and release the feature. So, we can create our benefit hypothesis using the following format:

If {proposition}, {then benefit}

Our *proposition* is a description of the feature. The *benefit* is the expected value delivered.

We may want to include the metrics we believe will prove or disprove our benefit hypothesis. In that case, we may use the following format to include our measures:

We believe that {proposition} will lead to {benefit} and this will be proven when {metric}.

As an example, let's take the epic for the aerial pizza drone that we defined earlier in the chapter. A possible feature could be the addition of a built-in heating unit to keep the pizzas warm while delivering the pizzas. The following statement could act as the benefit hypothesis for the feature:

We believe that a heating unit built into the drone will lead to increased customer satisfaction by ensuring that pizzas are not delivered cold and this will be proven when we view NPS survey results.

Remember here to avoid vanity metrics – measures that yield positive reactions but often don't indicate whether the value is really realized.

Acceptance criteria

Acceptance criteria are measures that confirm the implementation is complete and the benefit is delivered. They are a statement of the system's behavior with the feature included.

Typically, to gauge whether a benefit hypothesis is proven, you can map any number of acceptance criteria to a single benefit hypothesis statement.

Because the acceptance criteria describe the system behavior, we will examine a way of writing the acceptance criteria using BDD techniques in our next section.

Writing acceptance criteria using BDD

In the previous section, we looked at the function of acceptance criteria. One key role acceptance criteria play is that they describe the system behavior with the feature's inclusion. This rolls down to the desired behavior of components in the form of user stories or code functions.

We specify the desired behavior in the Gherkin format. The Gherkin format uses the following structure:

```
GIVEN (the initial conditions)
WHEN (an input that triggers the specific scenario occurs)
THEN (the desired behavior happens)
```

`WHEN` and `THEN` may have one or more clauses that describe the conditions and behavior accordingly.

Extending our example of the built-in heating unit feature for our pizza drone, we may want the following statement to act as acceptance criteria:

GIVEN the heating unit is installed and warmed up…

…WHEN a pizza is placed in the heating unit while on a delivery run…

…THEN the pizza will be hot when it reaches its destination.

Acceptance criteria in this format can be used as the basis for acceptance tests that can be automated. These automated acceptance tests are executed using Cucumber, a testing tool that runs BDD tests.

Prioritizing using WSJF

Once product management has specified the feature in terms of its beneficiaries, benefit hypothesis, and acceptance criteria, it is placed in the **Program Backlog**.

The Program Backlog is the list of features that the ART can work with. Because the ART is limited in terms of how many features it can handle at once, it's important that product management prioritizes the features to make the work more focused.

There is a variety of criteria that can be used to establish the prioritization of the features in the Program Backlog. SAFe advocates looking at Principle 1: Take an economic view, which we originally saw in *Chapter 2*, *Culture of Shared Responsibility*, when sequencing the features.

We start our economic view by focusing on the **Cost of Delay** (**CoD**) of the feature. The CoD is how much the value will diminish if we delay the release of the feature. It may be easier to look at the CoD as being made up of the following factors that can be relatively estimated:

- **User Business Value**: How much value is anticipated to be generated by a feature in comparison to the other features on the Program Backlog?
- **Time Criticality**: If a feature is not implemented promptly, is there a drop in value? Do we miss an important market window?
- **Risk Reduction or Opportunity Enablement** (**RR/OE**): Is there another important way that a feature may reduce risk or expose the organization to new markets?

Another factor SAFe focuses on is the size of the job. We want to focus on the shortest jobs first. It's easier to release jobs that take less time and immediately receive value, rather than starting with larger jobs.

SAFe then combines the focus on the CoD and the size of the job into a formula called WSJF. This formula can be specified as follows:

$$WSJF = \frac{Cost\ of\ Delay\ (CoD)}{Job\ Size}$$

If we look at the preceding formula and substitute our CoD factors, we can rewrite our formula as follows:

$$WSJF = \frac{User\ Business\ Value + Time\ Criticality + RR|OE}{Job\ Size}$$

Product management can collaborate with business owners, the system architect, and other stakeholders in refinement sessions to determine the WSJF value for each feature. The participants determine the relative value of the four components (User Business Value, Time Criticality, RR/OE, and Job Size), often using values in a modified Fibonacci sequence (1, 2, 3, 5, 8, 13, 20, 40, 100).

Let's look at how product management collaborates with other parties to calculate the WSJF value for different features.

The group will convene and look over the set of features. They will then decide which feature has the smallest User Business Value – that feature has its User Business Value marked with a *1*. They then look at the other features and decide how their User Business Values compare to the reference feature. If they are considered to be the same, they also receive a *1*. If the other feature is bigger, they place a number on the Fibonacci sequence that describes how much bigger than the reference they are.

The following table illustrates the collaborative process of determining the User Business Value in progress:

Feature Name	User Business Value	Time Criticality	RR|OE	Cost of Delay	Job Size	WSJF
Built-in heating unit	8					
Drone defense system	1					
Repulsor-based thrusters	5					

Table 10.2 – Example of WSJF collaboration – User Business Value defined

The process is repeated for Time Criticality, RR/OE, and Job Size. For the preceding table, on the columns that represent User Business Value, Time Criticality, RR/OE, and Job Size, there must be at least one *1*. The following table continues our example with Time Criticality, RR/OE, and Job Size also defined:

Feature Name	User Business Value	Time Criticality	RR|OE	Cost of Delay	Job Size	WSJF
Built-in Heating Unit	8	8	1		1	
Drone Defense System	1	13	3		2	
Repulsor-based thrusters	5	1	1		5	

Table 10.3 – Example of WSJF continued – Time Criticality, RR|OE, and Job Size defined

The CoD is calculated by adding User Business Value, Time Criticality, and RR/OE together. WSJF is calculated by dividing the CoD by the Job Size. The following table completes the determination of WSJF for our features:

Feature Name	User Business Value (UBV)	Time Criticality (TC)	RR|OE	Cost of Delay (CoD) = UBV+ TC+RR|OE	Job Size (JS)	WSJF (CoD/JS)
Build-in Heating Unit	8	8	1	17	1	17 (1st)
Drone Defense System	1	13	3	17	2	8.5 (2nd)
Repulsor-based thrusters	5	1	1	7	5	1.4 (3rd)

Table 10.4 – Example of WSJF determination – complete

Regular refinement sessions of new features in the Program Backlog allow product management to understand the features to be completed first. This will influence the program vision and the roadmap that product management will communicate to the ART in PI planning.

Preparing for PI planning

With a prioritized Program Backlog, product management and the rest of the ART can understand the initial scope of work for the upcoming PI. Ideally, these activities happen at least a month before the actual PI planning event so that any surprises are found early.

Business owners will prepare a presentation that will help the ART understand the business context and speak to how the development of the selected features may line up with the overall business strategy.

The system architect will also prepare a presentation outlining the important architectural notes and changes. This may also include any important enablers that the ART is developing for release.

What remains are the outputs of synthesis. Product management works to assemble the following artifacts before PI planning:

- A vision of what solutions the ART can deliver

- A roadmap of features and where they fit into the product's evolution
- The list of prioritized features that the ART commits to developing in the upcoming PI

Product management outlines the program vision, intending to set the focus of the ART on a common mission. This vision will be communicated to the entire ART in a presentation made after the business owner's presentation.

The selected group of features intended to be worked on in the upcoming PI is communicated to the Agile teams on the ART. The teams can choose which features they may want to contribute to and begin looking at these features closely. They may want to start breaking those features down into stories, as we will see in *Chapter 11*, *Continuous Integration of Solution Development*, and may also look for any risks or dependencies.

Summary

In this chapter, we looked at the activities that trigger the Continuous Delivery Pipeline. With Continuous Exploration, the ART examines the marketplace and the wants and needs of its customers to generate ideas for new products or new features.

The process begins with the understanding that development is really a series of incremental build-measure-learn cycles that start by creating a hypothesis of the customer value that can be achieved.

The collaborative process then begins for product management and the system architect. Product management collaborates with the customer to gain a greater understanding of the customer and marketplace. The system architect researches the hypothesis to understand the impact that the changes will have on the architecture.

When research is complete, the ART will take what it has learned and create features. They will elaborate on these features, place them into the Program Backlog, and prioritize them. A set of the highest sequenced features will be selected for the upcoming PI. Those selected are communicated to the ART to prepare for the next phase of PI planning.

After PI planning, the ART sets to work on development. This development work starts on the next phase of the Continuous Delivery Pipeline, Continuous Integration. We will explore what Continuous Integration involves in detail in our next chapter.

Questions

1. Which is not an activity of Continuous Exploration?

   A. Hypothesizing

   B. Collaboration and research

C. Development

D. Synthesis

2. Who does product management collaborate with during Continuous Exploration (select three)?

A. Release train engineers

B. Customers

C. Product owners

D. Scrum masters

E. System architects

F. Solution train engineers

3. Innovation accounting is used in which of the phases of Build-Measure-Learn?

A. Build

B. Measure

C. Learn

D. All phases

4. The Lean UX process creates a _____ to prove or disprove a benefit hypothesis.

A. Most Valuable Product

B. Mean Measurable Feature

C. Minimum Viable Product

D. Minimum Marketable Feature

5. Which areas does the system architect focus on when evaluating architectural changes for a new Feature (select three)?

A. Security

B. Cost

C. Testability

D. Reuse

E. Releasability

Further reading

- A series of articles on the scaledagileframework.com website that act as a reference for the Continuous Delivery Pipeline: https://www.scaledagileframework.com/continuous-delivery-pipeline/
- The reference article in the Continuous Delivery Pipeline series that talks about Continuous Exploration: https://www.scaledagileframework.com/continuous-exploration/
- A reference article on scaledagileframework.com that details epics and the SAFe Lean Startup Cycle: https://www.scaledagileframework.com/epic/
- A reference article on scaledagileframework.com detailing how to use innovation accounting and leading indicator metrics: https://www.scaledagileframework.com/guidance-applied-innovation-accounting-in-safe/
- This article describes the Lean UX process adapted by Scaled Agile for SAFe: https://www.scaledagileframework.com/lean-ux/
- An explanation of the Strangler pattern with examples of how to implement it: https://www.techtarget.com/searchapparchitecture/tip/A-detailed-intro-to-the-strangler-pattern
- A guide to writing Features in SAFe, with the incorporation of a Feature Writing Canvas: https://medium.com/product-manager-tools/writing-effective-features-58a240e69222
- *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses* by Eric Ries – This book defines the Build-Measure-Lean cycle, innovation accounting, pivots, and what an MVP can be
- *Lean UX: Designing Great Products with Agile Teams* by Jeff Gothelf and Josh Seiden – this book originally explained the Lean UX process, an iterative way to approach product design
- *Innovation Games: Creating Breakthrough Products Through Collaborative Play* by Luke Hohmann – this book explains innovation games, a number of collaborative exercises done by product management and the customer to unearth customer needs and wants

11

Continuous Integration of Solution Development

After PI planning, the teams on the ART are set to work. They'll look at the outputs of Continuous Exploration and the features selected for the PI and carry them to the next stage of the Continuous Delivery Pipeline, **Continuous Integration** (**CI**).

In this chapter, we will discover the following activities in the CI stage of the Continuous Delivery Pipeline:

- Developing the solution
- Building the solution package
- Performing end-to-end testing
- Moving the packages to a staging environment

We will also discover that the processes described in the Continuous Delivery Pipeline will meet up with automation in a **CI**/**Continuous Deployment** (**CD**) pipeline in the CI stage.

Let's join the ART now as they develop a solution as defined by the features created during Continuous Exploration.

Developing the solution

Teams on an ART work in markedly different ways than those that worked in product development using waterfall methodologies. An emphasis on Lean thinking and a focus on the system dictates newer ways of working.

We will examine the following aspects of engineering practices that are used by Agile teams today:

- Breaking down the work
- Collaborative development

- Building in quality
- Version control
- Designing to the system

These practices strive to allow for a continuous flow of work, ready for the next state of building.

Breaking down into stories

An important part of the Lean flow we picked up on in *Chapter 4*, *Leveraging Lean Flow to Keep the Work Moving*, was to keep batch sizes small. A feature, as presented to us before PI planning, is a large batch of work, meant to be completed by the end of the PI. In order to ensure a smooth flow of work, the feature must be broken down into smaller batches of work.

User stories often describe small pieces of desired user functionality meant to be delivered by the end of a sprint or iteration, which is commonly two weeks. They are commonly phrased in a user-voice form that briefly explains who the story is for and what the desired functionality and intended value are. An example follows of the user-voice form:

As a customer, *I want* an itemized receipt of services emailed monthly *so that* I can understand and organize my spending.

An essential part of the story is its acceptance criteria. The acceptance criteria outline the correct behavior of the story and are the way the team determines that the story is complete.

Acceptance criteria can be written using the Gherkin format. This helps outline pre-conditions, inputs, and the desired behavior and outputs. These are described in clauses that begin with GIVEN-WHEN-THEN.

The following example of acceptance criteria for our story is written in the Gherkin format. Note how it describes the initial conditions, inputs, and outputs:

Initial conditions	**GIVEN I have configured the notification date…**
Input	…WHEN the notification date passes…
Output or desired behavior	…THEN I receive an email notification containing my itemized receipts.

Table 11.1: Acceptance criteria in the Gherkin format

Enabler stories can also come from features. These stories do not provide direct user value but allow for easier development and architectural capability for future user stories, creating future business value. SAFe® outlines the following four types of enablers.

- Infrastructure
- Architectural

- Exploration
- Compliance

Splitting a feature into *user* and *enabler* stories can be done in a variety of ways. The following methods can be used to create stories that can be completed in a sprint.

- **Workflow steps**: Set up stories to perform the necessary steps of a workflow first. The other steps can be released in later sprints.
- **Variations of a business rule**: Divide up stories according to different rules for business, such as different classes of service, different product lines, and so on.
- **Major effort**: Examine the possible stories that could be followed. Pick the story that appears to be most difficult to do first.
- **Simple vs. complex**: When evaluating the feature, is there a story that could be written that would provide the core functionality? That's the first story to work. Subsequent stories would elaborate on the core functionality.
- **Variations present in data**: Start with a story that works with one data type and move to different stories to handle the other data types.
- **Different data entry methods**: Start by creating a story that enters the data manually, then progress with further stories with automated data entry.
- **Different system qualities**: One example of system qualities could be the different devices or interfaces our application would work with and establishing a story for each.
- **By operation**: Divide into stories that handle different operations. A common breakdown is **Create, Read, Update, and Delete** (**CRUD**).
- **Use case scenarios**: Create a story for every use case.
- **Setting a spike and follow-up**: A spike is used to set aside development time to research a technical approach or an unknown. Once the research is complete, proceed with stories that implement the approach.

Note that splitting a large story further may be required so that the story can be completed and delivered by the end of the sprint. The preceding methods can be used to split larger stories into smaller stories.

Collaborative development

Although teams can choose how they develop their stories, high-performing teams have found that team members working together instead of working solo produces higher quality products and enables effective knowledge sharing, and this creates stronger, more collaborative teams.

In this section, we'll discuss two practices that allow teams to collaboratively develop products together, promoting better quality and stronger team cohesion: pair programming and mob programming or swarming.

Pair programming

Pair programming is a practice that originated with **Extreme Programming** (**XP**). Instead of two developers working separately in front of two computers, the developers are working in front of a shared computer, exchanging ideas back and forth, and simultaneously coding and reviewing their work.

With two developers working together, the following patterns emerge for how they collaborate.

- **Driver/navigator**: In this pattern, one developer takes control of the computer (driver) while the other developer reviews and guides by commenting on what is typed on the screen (navigator). At certain times during the session, the roles are switched. This is the most common pattern used in pair programming. This pattern is frequently used when one of the developers is an expert programmer paired with a novice.
- **Unstructured**: In this ad hoc style of pair programming, there are no set roles for the developers. The collaboration tends to be unguided and loose. Typically, this pattern is adopted when neither developer knows what approach will work. This pattern works well with developers that are at similar levels of expertise, but two novice developers may have problems working with this approach.
- **Ping-pong**: This pattern is frequently used where one developer writes the test and the other developer works to pass the test. The developers switch roles frequently between writing the tests and writing code to pass the tests. This style works well with two developers at an advanced level.

Pair programming has proven to be an effective way of working together. Code written during a pair programming session is frequently reviewed and debugged, resulting in higher-quality code. Knowledge is shared between developers, creating faster learning for novices or those new to the code base. If the code breaks, there is also more than a single developer with an understanding of the code that can help with repairs.

A common misconception is that pair programming requires twice the effort or resources. This belief is not supported by studies done on the effectiveness of pair programming, including one done by the University of Utah (`https://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF`) that found that while development costs increased by 15%, defects discovered at later stages decreased by 15% and code functionality was accomplished using fewer lines of code, which is a sign of better design quality.

Mob programming or swarming

Mob programming can be considered to be pair programming taken to the highest level. Instead of a pair of developers, the entire team is seated in front of a single computer and controls. The team is working on the same thing, in the same time and space, and on the same computer.

A typical pattern for this is a variation of the driver/navigator pattern. One person on the team has control of the computer, typing and creating the code or other pieces of work. The other members of the team review and guide the driver as navigators. After some time (usually 10 minutes), the controls are rotated to another member of the team. The rotation continues until all members of the team have had an opportunity to play the driver.

Mob programming benefits the entire team. Knowledge sharing of the code is applied to the entire team, instead of a pair of developers. Communication is easier with the entire team present. Decisions are made with the most current and relevant information.

Building in quality by "shifting left"

During this time, not only is the product being developed but the ways of ensuring that the product is high-quality are also being developed simultaneously. This is a change from traditional development where tests were created and run after the development of code. This change is often referred to as *shifting left* as illustrated in the following representation of the development process. This is one of the important practices in SAFe, which is described in more detail in the SAFe article on *Built-In Quality* (`https://www.scaledagileframework.com/built-in-quality/`):

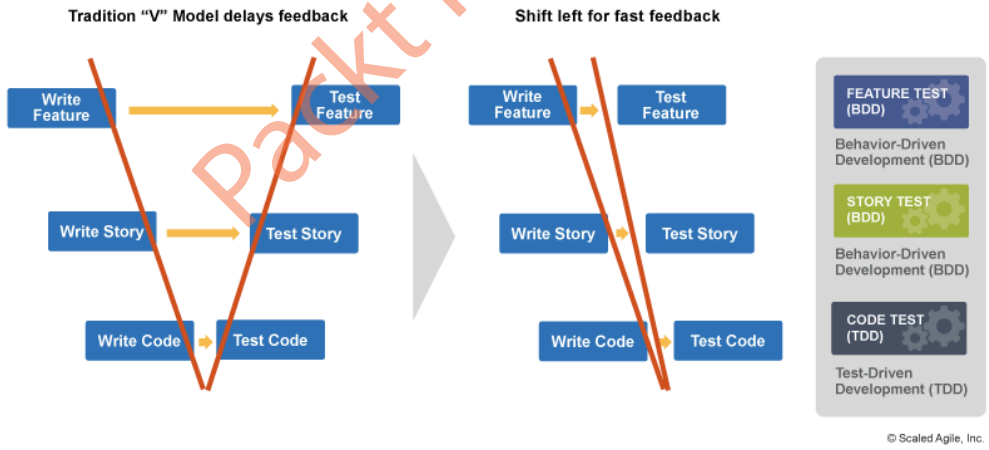


Figure 11.1 – Comparison of testing with "shift left" (© Scaled Agile, Inc., All Rights Reserved)

In the preceding diagram, we see on the left that traditional testing may test stories and features long after the stories and features were originally conceived. This delayed feedback may take as long as 3 to 6 months, which may be too late to know whether we are moving in the right direction.

With the diagram on the right, we see that we can accelerate the feedback using TDD and BDD tests to evaluate whether the behaviors of the feature and story are what is desired. Ideally, these tests should be automated so that they can be run repeatedly and quickly.

Another thing we can see from the preceding diagram is that there are many levels of tests, some of which should be run repeatedly with the help of automation, and others that may take some time or can only be run manually. How do we know which tests to automate and which tests should be run frequently?

Mike Cohn described the levels of testing as a "testing pyramid" in his book *Succeeding with Agile*. He initially described the pyramid with the following three levels from bottom to top.

- Unit testing
- Service testing
- UI testing

Other types of testing can be added and applied to the testing pyramid. This allows us to view the testing pyramid as follows:

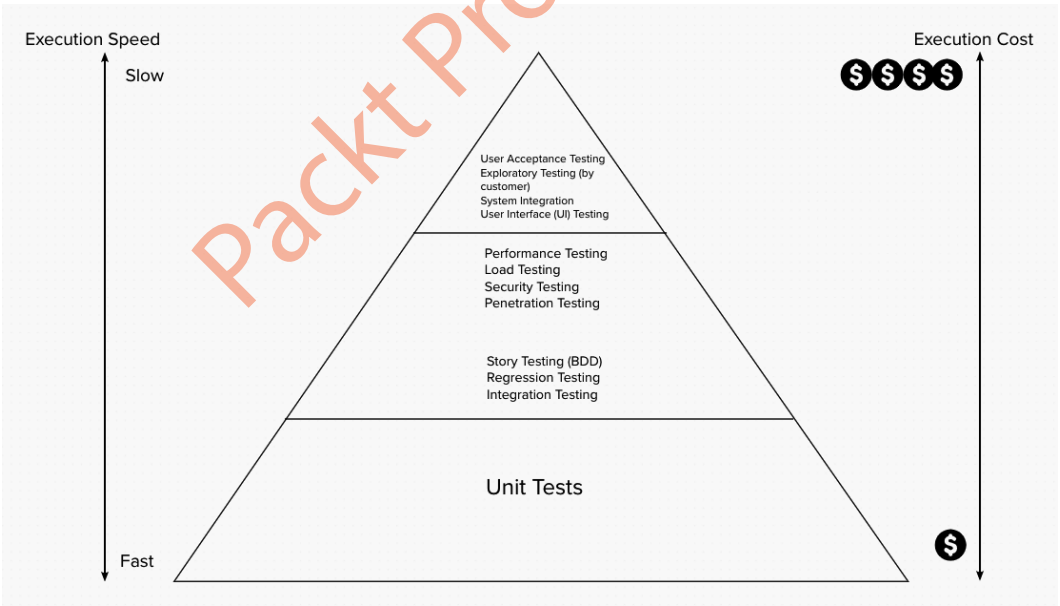


Figure 11.2 – Test pyramid

Note that at the bottom of the pyramid, unit tests are both the quickest to execute and the cheapest to run. It makes sense to automate their execution in the pipeline and frequently run them at every commit into version control, ideally during the build phase.

As you move further up the pyramid, the tests gradually take longer to execute and are more expensive. Those tests may not be run as frequently as unit tests. They may be executed through automation, but only upon entering the testing phase. Examples of these types of tests include story testing from BDD, integration testing, performance testing, and security testing.

The tests at the top of the pyramid take the longest to execute and are also the most expensive to run. These are mostly manual tests. These tests may be run just before release. Examples of testing here include user acceptance testing and exploratory testing done by the customer.

Most tests look to verify either proper code functionality and correctness as well as verification of the behaviors of the story and feature. The primary methods of creating tests to measure these criteria fall into the following methods: TDD and BDD. Let's look at how these tests are developed.

TDD

TDD is a practice derived from XP. With TDD, you practice the following flow:

1. Create the test. This is done to understand the behavior.
2. Watch the test fail (even with no code written). This gives us confidence in the test execution environment and demonstrates system behavior when a test fails.
3. Write the simplest code necessary to pass the test.
4. Ensure all tests pass. This may mean any new code created is revised until the tests pass.
5. Refactor the tests and code as needed.

This flow is repeated as new functionality is developed. A graphical representation of this flow is as follows:

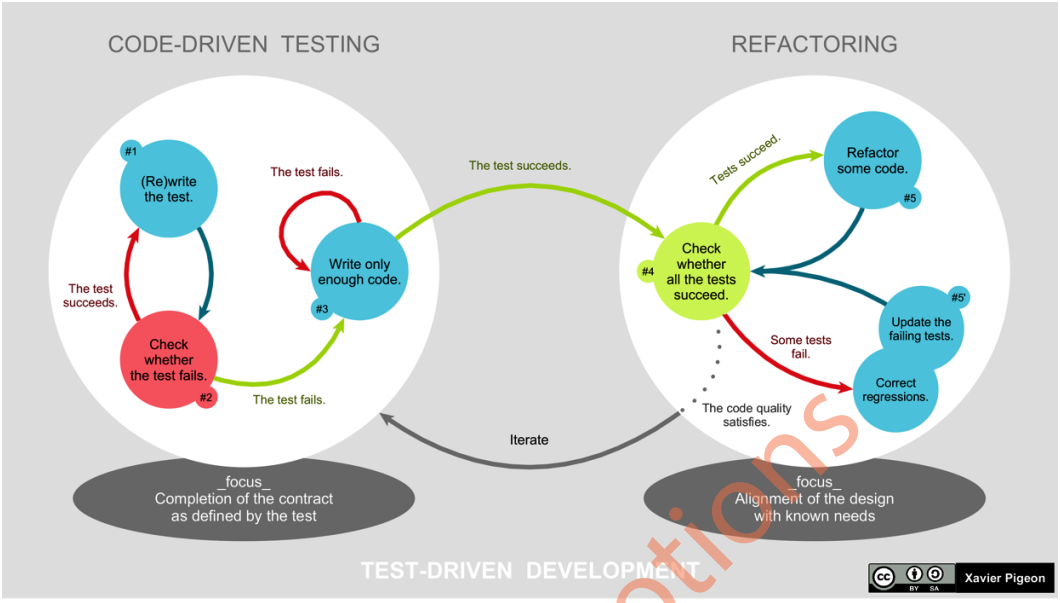


Figure 11.3 – TDD (https://en.wikipedia.org/wiki/Test-driven_development#/media/File:TDD_Global_Lifecycle.png licensed under CC BY-SA)

The tests usually written using TDD are unit tests; small, easily executed tests designed to verify the correct functionality of a code module. Broader tests use BDD to develop tests that verify the systemic behavior of features and stories. Let's look at BDD now.

BDD

BDD is often seen as an extension of TDD, but while TDD looks to verify the correct behavior of individual code functions and components, BDD strives to verify the correct behavior of the system as an executable specification expressed in features and stories. One application of BDD was seen earlier in this chapter when we created the acceptance criteria for the story.

Looking at the correct systemic behavior involves three perspectives that work together to bring their point of view of what is eventually specified, what is developed, and what gets tested as correct. The following three perspectives include the following:

- Customers who understand the business needs and look for the desirability and viability of new features
- Developers who understand the feasible technical approaches
- Testers who view the edge cases and boundary conditions of the systemic behavior

BDD brings these three perspectives together using specifications. These specifications are written in a **Domain-Specific Language** (**DSL**) that employs natural language syntax so that technical and non-technical people can collaboratively develop the specification. One of these DSLs is Gherkin, which divides behavior into the following three clauses:

- **GIVEN** outlines the initial conditions that must be present for the desired behavior in a scenario
- **WHEN** describes the input that triggers a scenario
- **THEN** describes the desired behavior for the scenario

Multiple GIVEN, WHEN, and THEN clauses may be joined together using **AND** to indicate multiple conditions, inputs, and behaviors accordingly

The specification, using the DSL, can become several artifacts. Product owners and product management create the acceptance criteria for features and stories with the other members of the development teams. The creation of acceptance criteria can be seen as the discovery of the desired systemic behavior.

The next phase of creating the specification is formulation. In this phase, developers and testers work together to create acceptance tests. They can take the acceptance criteria as written and elaborate specific criteria in each clause, including allowable initial conditions and values to measure for inputs and outputs so that the specification for a specific scenario becomes a test. Ideally, acceptance tests are written in the same DSL as the acceptance criteria.

We can create an automated test by taking the acceptance criteria for our story and adding specific pre-conditions, input, and desired output or behavior. Let's look at our previously seen acceptance criteria converted into a test in the following table:

Acceptance Criteria	**Test**
GIVEN I have configured the notification date…	Given the date state is not x…
…WHEN the notification date passes…	…when it's one business day after the date state has changed to x…
…THEN I receive an email notification containing my itemized receipts.	…then send an email notification to all users with xxx content.

Table 11.2 – Conversion of acceptance criteria into a test

The last phase of the specification is automation. The acceptance test, written in the DSL, can be executed in a tool that allows for automated testing. Acceptance tests written in Gherkin can be executed by tools such as Cucumber, JBehave, Lettuce Behave, and Behat.

Version control

Version control software allows for multiple developers on a team to develop in parallel on the same code base, test scripts, or other bodies of text without interference from other developers' changes. Each change in the version control system is recorded. Changes are consolidated using merge operations to bring together and resolve changes in the bodies of work.

Important practices for the use of version control include the following ideas:

- **Save EVERYTHING in version control**: A lot of design decisions are captured as artifacts in version control beyond source code. Code, test scripts, configuration files, and any other text-based artifacts can be tagged together to show they are part of the same release. Version control also allows for the retrieval of previous versions to roll back any changes or to view the evolution of design decisions.
- **Everyone uses the same version control system**: As we saw in *Chapter 1*, *Introducing SAFe® and DevOps*, the photo-sharing website Flickr had a common version control system between its development and operations people. This allowed for the easy retrieval of artifacts by anyone when production failures occurred.

Other best practices of version control that take place during the build process will be identified in our next section.

Designing to the system

The features and stories are not the only criteria teams have to consider when developing new capabilities for their product. **Non-Functional Requirements** (**NFRs**) are qualities that may impact every feature and story acting as a constraint or limitation. These NFRs may deal with security, compliance, performance, scalability, and reliability, among other things

Two practices are performed to ensure compliance with some NFRs: designing for operations and threat modeling. Let's take a look at these practices.

Designing for operations

The collaboration between development and operations is a hallmark of the DevOps movement. Ensuring that operations can easily examine system resources is something that is easily incorporated in the early stages of development rather than added as an afterthought.

A key part of ensuring capabilities are present for proper maintenance of the product is application telemetry. The product as a system must allow for the easy measurement of system resources, including how an application uses resources such as server memory and storage. In addition to system measurements, application telemetry should also allow for the measurement of business data, used as a leading indicator to validate the benefit hypothesis.

Other considerations include ensuring that changes brought by new features can be easily rolled back or that fixes can be rolled forward through the Continuous Delivery Pipeline. When doing this, be aware of components that may represent the state of the system, such as a database. These may not be easily rolled back.

Threat modeling

Moving toward a DevSecOps approach requires a *shift-left* mindset toward security. This mindset allows for including security concerns in the design and development stages of the Continuous Delivery Pipeline, which gives a more holistic view of the product.

We first saw that threat modeling was part of architecting the system in *Chapter 10*, *Continuous Exploration and Finding New Features*. As part of threat modeling during CI, we may be asking the following questions:

- What are we working on? This helps you get an idea of the scope.
- What can go wrong with it? This allows you to start your assessment using brainstorming or a structured threat modeling process, such as the **Application Security Framework** (**ASF**) or **Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, or Elevation of Privilege** (**STRIDE**), which identifies the types of possible information security threats.
- What can we do about the things that go wrong? Based on the assessment, devise countermeasures or mitigation steps.
- Are we doing a good enough job so far for the system as is? Continually evaluate the assessment, countermeasures, and mitigation steps.

Developing toward DevSecOps is then based on the countermeasures and mitigation steps identified during the assessment.

As development changes are completed, they must be integrated with the current product as it stands and be tested. This may start the incorporation of automation into a CI/CD pipeline. Next, let's examine how entry into the CI/CD pipeline begins with a build process.

Building the solution package

The CI/CD pipeline can be triggered by version control system actions such as saving a new change as a commit. Before the changes are accepted by the version control system, they should go through a testing process to ensure they will not adversely affect the current code base. This process of testing and version control integration is an important part of the CI process where practices are divided into a version control perspective and a testing perspective.

Let's look at each of these perspectives and the practices within.

Version control practices

Good version control practices ensure that changes introduced into version control are evaluated through testing before they are saved and merged with the existing code base. This ensures that the code base is robust, without changes that may prevent the code base from being built or packaged correctly.

Version control practices can further be divided into three types that help ensure a robust code base as changes come in. Let's look at what these practices are in detail.

CI of code

The practice of CI has its origins in optimizing the build process through automation by a build script or a CI tool. Saving a change to version control through a commit operation sets off a chain of the following steps:

1. The application would be built, incorporating the saved changes. If building this code resulted in an error, notifications would be sent regarding the error. The changes would not be allowed to merge with the code base.
2. If the build succeeded, tests would run against the build with the code changes. These tests are often small tests, measuring a small part of the functionality, that can be quickly executed and don't take a lot of time. If a test failure was detected, a notification would be sent, and the changes would not be allowed to merge with the code base.
3. Another type of test that could be executed would be a scan of the code base. Scanning could look for deviations from coding style standards and syntax errors to known security vulnerabilities. Depending on the severity of the findings, changes would not be allowed to merge with the code base. Notifications would be sent on all findings.
4. Upon successful results from the building, testing, and scanning steps, the code change would be recorded into the version control system. Merging the change into the main trunk of the code base would proceed on the version control system, integrating the changes with the rest of the code.

The preceding steps are outlined in the following diagram:

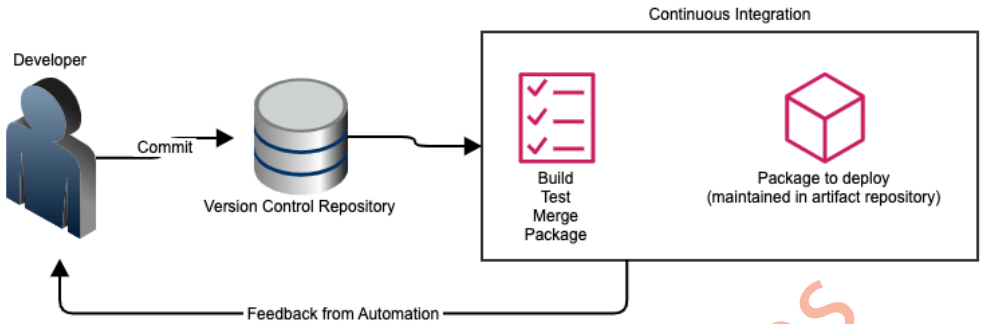


Figure 11.4 – CI automation

When performing the preceding chain of steps, teams eventually figured out that successful code integration relied on the following factors:

- **Performing the integration frequently**: Many teams started with performing the build and testing processes nightly, examining changes that had been saved over the previous day. With a lot of changes, the *nightly build* often grew until it couldn't be finished until the next day or later. More frequent builds, often occurring several times a day, allowed people to see the build results and act on them swiftly.
- **Performing integration on fewer changes**: A *nightly build* that absorbs multiple changes from multiple developers produces problems when the build fails. It becomes more complex to troubleshoot to determine which developer's change *broke the build*. Performing a build based on fewer changes, with the idea of building on each saved change, allows for quicker troubleshooting when errors occur.

CI results in the following outputs, regardless of success or failure:

- **Fast feedback**: The results of CI should occur in minutes. Any errors that prevent the successful completion of CI will be given in a short amount of time, resulting in fewer delays.
- **Deployable artifacts upon success**: With a successful build, a build package able to be deployed into non-production environments will be produced.

CI tools such as Jenkins, GitLab pipelines, and GitHub Actions form the basis of the automation that allows the CI of code to occur.

Trunk-based development

Multiple developers in a single team or even multiple teams (such as the team of teams that is present in an ART) often must work on the same code base that is saved in version control. Version control systems allow for parallel development of the same code base by allowing changes to be branched out. A developer or team could work on a branch without their change affecting the rest of the team or ART until it was ready to merge and be shared with other developers or teams.

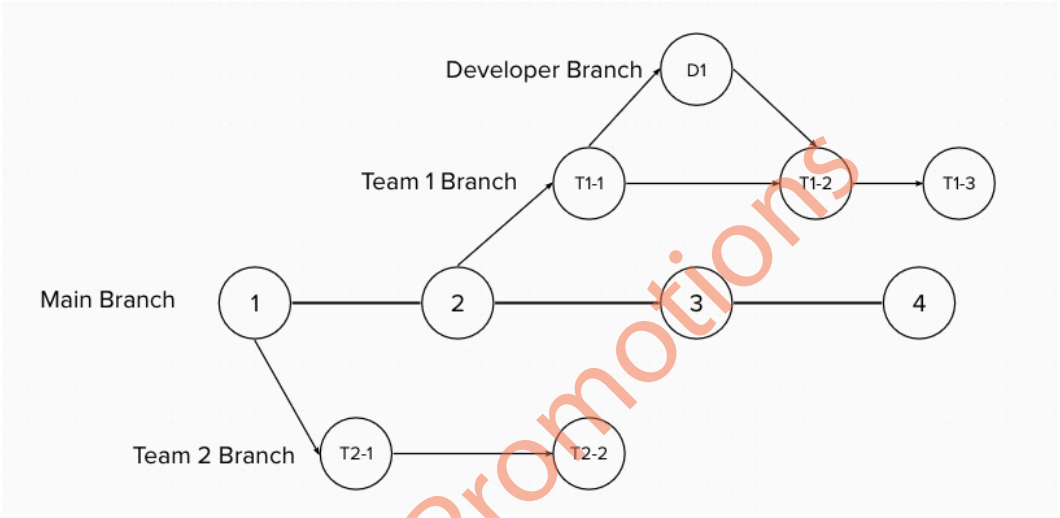


Figure 11.5 – Branching structure example

In the preceding illustration, **Team 1** and **Team 2** have their own branches of the code base, based on different versions of the main branch (commonly referred to as the *trunk*). A developer on **Team 1** has created a change (**D1**) and merged it back into the **Team 1** branch as a change, **T1-2**. With separate team branches, how do we know that a necessary change from **Team 2** is visible and can be used by **Team 1**?

Another problem occurs as **Team 1** and **Team 2** develop on their branches without receiving updates from the trunk and wait to merge when they release or at the end of the sprint. Keeping track of the multiple changes from multiple teams and resolving a large number of merge conflicts results in an incredible challenge.

To keep things simple and ensure changes are visible to all teams, we want to avoid branches that are permanent or last a long time. Developers and teams may form branches to allow for parallel development, but when ready, they must merge back to the main branch, destroying the offshoot branch in the process. This process is known as trunk-based development. The following diagram highlights the process of trunk-based development:

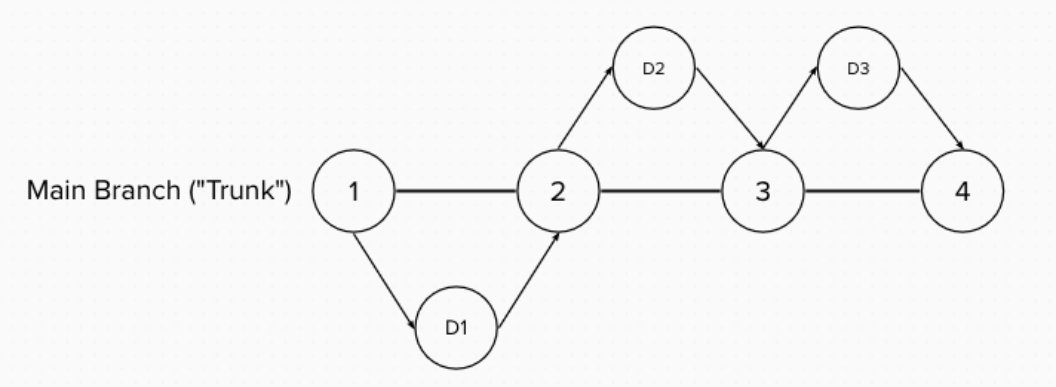


Figure 11.6 – Branching structure with trunk-based development

Trunk-based development allows for easier merge operations to occur since a merge to the main branch is happening on each validated change instead of a group of changes.

Gated commits

With trunk-based development, we are merging changes to the main branch of the code base as often as possible. This main branch is used by all the teams on the ART. Since the integrity of the main branch is vital for multiple teams, how do we ensure that any errant changes don't break the current code base?

To ensure a robust code base, we must follow a gated commit process where before a change is allowed to merge with the main branch, it must successfully pass the build and test process. Additional measures, such as a review of the change, may also be taken.

In Git-based environments, Git servers from Bitbucket, GitLab, and GitHub define gated commits as pull requests or merge requests that allow for closer scrutiny when a merge operation is requested.

Testing practices

We saw that build processes relied on testing to ensure that changes to a code base did not adversely affect the functionality or the security of the product. The tests that were run proved to be important since ideally, the build process would be performed on every saved change and if successful, the next step would be merging the change to the main branch, which is visible to the team or multiple teams in the case of the ART.

The build process involves two types of tests that are run against a potential new version of the code base: automated unit testing and static analysis for application security.

Let's take a deeper look at each type of test run as part of the build process.

Automated unit testing

Unit tests are often written at the same time as code, if not beforehand. These unit tests may be run by an individual developer on their workstation while the code is being developed. If that's the case, why run them again as part of the build process?

The main idea of CI is to ensure a standard, reliable process. Automation through a CI/CD pipeline ensures that this occurs for all developers. Adding unit testing during the build process on an automated CI/CD pipeline ensures that the unit tests are run on every developer's code change every time.

It's also important to make sure that any unit tests that have been updated are simultaneously part of the potential change to the code base in version control. This ensures that code changes are validated against correct tests preventing a situation where the CI/CD pipeline stops because of incorrect tests. Collaborative development efforts between those creating the code and those creating the tests are required to ensure this situation doesn't occur.

Static analysis for application security

Static analysis is a process by which a tool scans the text of the code base, including the potential code change that is being checked in, to find specific text patterns. These text patterns can be used to identify the following issues:

- Coding errors
- Known security vulnerabilities
- Non-adherence to coding guidelines or coding standards

The analysis is performed without the need for executing the application. Because of this, static analysis is an efficient means of checking for problems in the build process.

As we saw in *Chapter 3*, *Automation for Efficiency and Quality*, static analysis can fall into the following two categories:

- **Static code analysis**: Look through the code for possible coding errors. Linting is an example of static code analysis.
- **Static security analysis**: Look through the code for possible security vulnerabilities and attack vectors. Applications that perform static security analysis may perform the following scans:
    - **Dependency scanning**: Scanning code dependencies and references to third-party libraries to find vulnerabilities
    - **Static Application Security Testing** (**SAST**): Scanning code to find attack vectors and vulnerabilities
    - **License compliance**: Scanning libraries to determine their opensource licensing model

- **Container scanning**: Scanning Docker containers to find embedded vulnerabilities
- **Secret detection**: Scanning code to find embedded credentials, keys, and tokens

Our application has passed the first set of tests, but is it ready for the rigors of a production environment? To answer that question, we look to perform system-level testing. Our next section examines the practices that enable system-level testing.

Testing end to end

At this point, we have performed tests on individual pieces of code and ensured the correct functionality while maintaining security. Here, we start to integrate the new changes of code with the existing code base and evaluate the system as a whole by testing the system end to end.

The practices that allow for true end-to-end testing of the system will be examined in the upcoming sections. Let's dive in.

Equivalent test environments

System-level testing should be performed in an environment that resembles the production environment as closely as possible. Testing the solution in an environment with as many similarities to the production environment as possible enables higher confidence that the solution will work when actually released into the actual production environment. The more similarities a test environment has with the production environment, the fewer variables come into play when problems are found and troubleshooting for the root cause begins.

A key factor in ensuring the equivalence between test environments and the production environment is the use of configuration management. With configuration management, key resources such as the operating system version, versions of key drivers, and versions of applications are recorded in a text-based configuration file. Ideally, the configuration file is maintained in version control with labels that indicate the version of the solution and its application in the test environments and production environment.

Because the cost of allocating exact duplicates of the resources in production may be prohibitive, the important view is that exact versions of resources, rather than the exact number of resources, are key to maintaining equivalence.

Test automation

System-level testing can encompass a variety of levels, most of which can be automated. When looking at a variety of tests at various levels, which of these tests could be automated?

In addition to the testing pyramid mentioned earlier, we may need to consider the people who need to understand the test results. A second consideration is whether the test is to verify that the solution is meeting requirements or whether the test is to allow developers to see whether their design approach is correct.

The Agile testing matrix looks at the various kinds of tests and organizes them from these considerations. The following diagram depicts the Agile testing matrix as seen in the SAFe article on *Agile Testing* (`https://www.scaledagileframework.com/agile-testing/`):

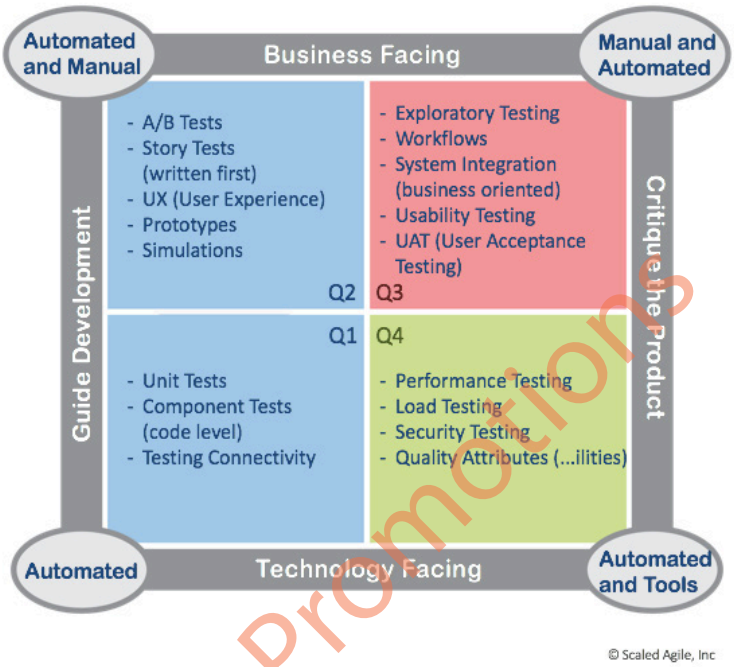


Figure 11.7 – Agile testing matrix (© Scaled Agile, Inc., All Rights Reserved)

We can see from the preceding diagram that the first consideration looks at the perspective of either the business or the technology. Developers look at the technology tests to ensure the correct functionality and proper operation of the solution. End users look at the business-facing tests to ensure an understanding of the solution and the validation of the benefit hypothesis.

We can also see the second consideration: whether the test informs the complete solution or the implementation. Tests that guide development assist in TDD and BDD approaches where the test is written first. Tests that critique the product look to see whether the solution complies with user requirements.

With two areas of concern within each of the two considerations, we can divide tests into the following four quadrants:

- **Q1**: These contain unit and component tests. These tests may be created as part of a TDD approach.
- **Q2**: These contain functional tests and tests for stories and features. These may be created using a BDD approach to allow for automated testing. Otherwise, some of this validation may be manual.

- **Q3**: These are acceptance tests of the entire solution. These may be the final validation before release. These are often manually run with alpha and beta users.
- **Q4**: These test overall system qualities, including NFRs. These verify the system in the production environment.

We will see that tests in Q3 are done during CD in *Chapter 12*, *Continuous Deployment to Production*. Tests in Q4 are done during Release on Demand as mentioned in *Chapter 13*, *Releasing on Demand to Realize Value*.

Management of test data

A key part of ensuring the similarities between a testing environment and the production environment is the data used to test the solution. Using data that could be found in production environments allows for more realistic test outcomes, leading to higher confidence in the solution.

Realistic test data can come from either synthetic test data or real production data. Test data may come from a backup of production data restored into the test environment. The test data should have any information that is considered private removed.

Synthetic test data is *fake data* created by a data generation tool such as DATPROF Privacy and Gretel. It offers the advantage of not requiring an anonymization step to redact private information.

Regardless of whether the data is anonymized production data or synthetic data, the test data should be maintained in version control, using artifact repository software for large binary-based data.

Service virtualization

Service virtualization allows for test environments to behave like production environments even when the test environment is missing resources available in production. The production environment may have crucial dependencies on key components that are impossible to copy because of the following factors:

- The component is not complete yet
- The component is being developed by a third-party vendor or partner
- There is limited access to the component in a test environment
- The component is difficult to configure in a test environment
- Multiple teams with differing setups require access to the component
- The component is too costly to use for performance or load testing

Systems made up of components that communicate together using well-known interfaces can take advantage of service virtualization to simulate the behavior of one or more components. If the virtualized service is a database, it can return synthetic test data.

Components that are simulated in test environments are called virtual assets. Virtual assets are created by tools that measure the true component's behavior by the following methods:

- Recording the messages, responses, and response times of a component as it communicates on a common channel or bus
- Examination of the component's logs
- Viewing the service interface specifications
- Manually applying inputs and measuring behavior

Once the virtual asset is created, it takes its place in the test environment. An illustration of the difference between the production environment and the test environment with virtual assets is as follows:

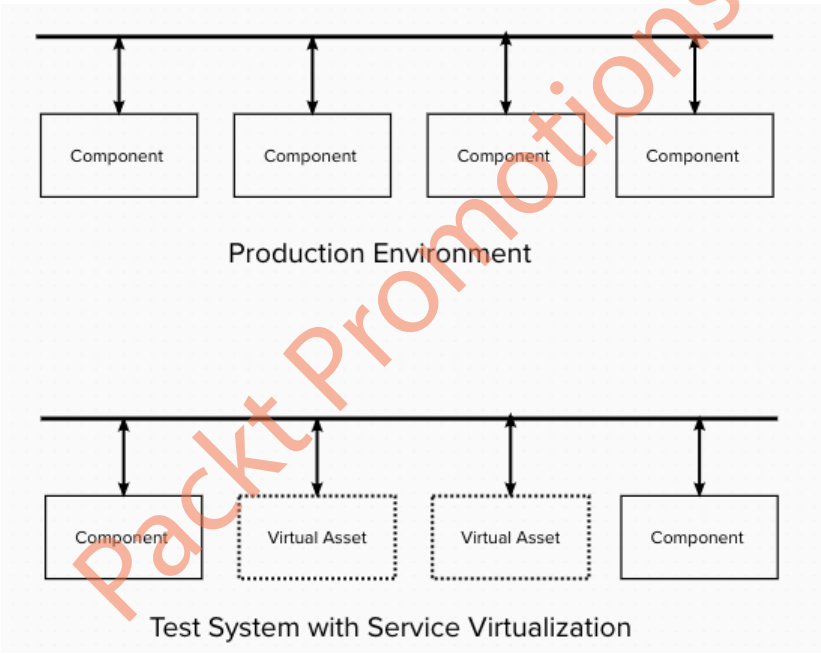


Figure 11.8 – Production vs. test environment

Popular tools for creating virtual assets include SoapUI and ReadyAPI from Smartbear, MockLab, Parasoft Virtualize, and WireMock.

An important difference to consider is that while service virtualization may seem similar to *mocking* or stubbing a component, the two concepts are not similar. Adding a mock component or a stub may be done during development when the component is not ready for release. The behavior of a mock object only returns one type of output – a success message – so the development of other components is not impeded. Service virtualization allows for the proper behavior for a wide variety of scenarios.

Environments with virtual assets should be maintained in configuration management tools. The configuration files and interface definition files for virtual assets should be kept in version control, in close proximity, and with labels identifying their role as test assets for versions of an application.

Testing nonfunctional requirements

As we perform end-to-end system testing, we need to remember the constraints our system has, which we have previously identified as NFRs. NFRs affect every story and feature, acting as a constraint that must be heeded. Qualities such as security, performance, reliability, and scalability, among other things, should be examined through testing to verify that these constraints aren't broken.

The testing of NFRs is often automated, involving specialized testing tools. Agile teams on the ART often work with the system team to ensure that the tooling is established to perform testing for NFRs as part of the end-to-end testing.

After the portfolio of tests is performed on our change, we may want one more opportunity to see whether our change is ready to be deployed to production. We place our changes into a staging environment, a stand-in for the production environment for a final examination before deploying changes to production. Let's look at the activities involved in deploying changes to staging.

Moving to staging

We may want verification that we can deploy the change to a production-like environment and verify that our solution still works. To enable this last look, we employ certain practices.

Let's look at these practices in depth.

Staging environments

A staging environment is a facsimile of the production environment, which has several uses throughout the PI. It is the place where demonstrations of the system as it currently stands are performed in the system demo. User acceptance testing can be performed in this environment, which is as close to production as possible.

As the changes to the product are being developed, the staging environment shows the state of change before deployment to production. At the very least, changes to the staging environment happen every sprint or iteration for the ART. More frequent changes are allowed as long as the build process and end-to-end testing are completed successfully.

A staging environment may also act as an alternative environment for production in a configuration known as blue/green deployment, which may allow for easy rollback in the event of production failures. Let's take a look at this configuration now.

Blue/green deployments

In a blue/green deployment, you have two identical environments. Of the two environments, one is the production environment and facing traffic, while the other is idle and on standby.

The idle environment receives the latest change where thorough testing occurs. At the appropriate time, the change is released by making the idle environment live and the other environment idle. This transition is illustrated by the following graphic:

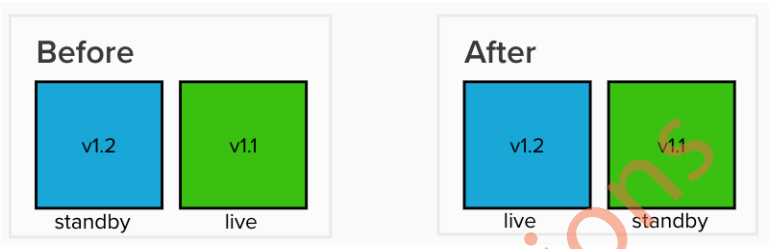


Figure 11.9 – Blue/green deployment release of a new version

If problems are discovered, a switchback can be made to roll back changes. This transition back and forth is easy for systems that do not track states. Otherwise, blue/green deployments must be carefully architected so that components that are capable of storing states, such as databases, do not get corrupted upon transition.

System demo

At the end of every sprint or iteration in the PI, after each team's iteration review, the teams get together to integrate their efforts. Working with the system team, they demonstrate the current state of the product as it stands so far in the PI in the staging environment. Business owners, customers, and other key stakeholders of the ART are present at this demonstration to view progress and supply feedback. This event provides fast feedback on the efforts of the ART so far.

Note that the system demo does not prevent the deployment of changes into production. That may continue to happen automatically as part of CD (which we will visit in the next chapter), but feedback may prevent the release of the change to customers until changes resulting from the feedback make their way into production and are released on demand.

Successful testing in a staging environment gives us confidence that our change has the correct functionality and is robust enough in a production environment, but the only true way to prove that is to deploy our change into the actual production environment.

Summary

In this chapter, we continued our discovery of the Continuous Delivery Pipeline by looking at CI, the part that implements the features created in Continuous Exploration. Features are divided into more digestible stories. Development of not only the product but also the tests to verify the product begins. Security and designing for operation concerns are included in the development.

The build phase introduces automation into the pipeline. When a commit to version control occurs, unit tests are run to ensure continued, correct functionality. The build is also scanned for coding errors and to find security vulnerabilities. If everything is correct, the commit will be allowed to merge with the main branch, or trunk, of the version control repository.

A successful build can trigger further testing in a testing environment that may be similar to the production environment. Here, system-level, end-to-end testing happens to guard against any production failures. The testing here is as automated as it can be. Accurate test data and service virtualization may offer a reasonable facsimile to a production environment for testing.

When building and testing are complete, the change may find itself in a staging environment, a copy of the production environment, or one-half of a blue/green deployment. A staging environment is also a place where changes are shown during a system demo, an event where the ART receives feedback on the development of the system at the end of each sprint or iteration.

After making its way to the staging environment, we must move our changes into production. That happens in CD, which we will explore in our next chapter.

Questions

1. What are two examples of collaborative development?

   A. Solo programming
   B. Pair programming
   C. Gauntlet programming
   D. Mob programming
   E. Cross-team programming

2. What is the first step in TDD?

   A. Write the test
   B. Write the code
   C. Refactor the test
   D. Refactor the code

3. When performing trunk-based development, a successful build and test will allow the committed change to merge with which branch?

   A. Release branch
   B. Fix branch
   C. Main branch
   D. Test-complete branch

4. According to the testing pyramid, what types of tests are the quickest to execute?

   A. Unit tests
   B. Security tests
   C. Story tests
   D. User acceptance tests

5. What text-based artifacts should be stored in version control?

   A. Code
   B. Tests
   C. Configuration files
   D. A and C
   E. All of the above

6. What can be used to allow test environments to be similar to production environments?

   A. Using old production servers
   B. Sanitized backups of production data
   C. Service virtualization
   D. B and C
   E. All of the above

7. What can a staging environment, identical to the production environment, be used for?

   A. User acceptance tests
   B. An idle environment for blue/green deployment
   C. System demos
   D. All of the above

Further reading

- Guidance from Scaled Agile on how to decompose features into stories and what good stories contain: https://www.scaledagileframework.com/story/
- A guide to common patterns used to decompose features into stories or to split big stories into smaller stories: http://www.humanizingwork.com/wp-content/uploads/2020/10/HW-Story-Splitting-Flowchart.pdf
- A good article detailing the practice of pair programming: https://www.techtarget.com/searchsoftwarequality/definition/Pair-programming
- The study done by the University of Utah detailing the benefits of pair programming: https://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF
- The original article from Woody Zuill, detailing how mob programming works and its benefits: https://www.agilealliance.org/resources/experience-reports/mob-programming-agile2014/
- An article from Scaled Agile on "shifting testing left" and adopting TDD and BDD: https://www.scaledagileframework.com/built-in-quality/
- Article from the **Open Web Application Security Project** (**OWASP**) detailing threat modeling: https://owasp.org/www-community/Threat_Modeling
- Article from the OWASP detailing processes used for threat modeling, including ASP and STRIDE: https://owasp.org/www-community/Threat_Modeling_Process
- An article expanding on Mike Cohn's testing pyramid: https://martinfowler.com/articles/practical-test-pyramid.html
- Scaled Agile article on Agile testing and the Agile Testing Matrix: https://www.scaledagileframework.com/agile-testing/
- An article by Smartbear, vendor of two leading service virtualization tools, on what service virtualization is, its benefits, and how it compares to stubs: https://smartbear.com/learn/software-testing/what-is-service-virtualization/
- Guidance from Scaled Agile on what a system demo is: https://www.scaledagileframework.com/system-demo/

12

Continuous Deployment to Production

As we continue our journey through the Continuous Delivery Pipeline, we have designed features to test a benefit hypothesis during Continuous Exploration. Implementation of these features happened during Continuous Integration where we created stories, developed changes and tests, and put our changes through the build and test process. Ultimately, our change is placed in a staging environment.

With Continuous Deployment, we complete the journey of the change to the production environment. However, the activities don't end with the deployment to production.

In this chapter, we will take a look at the practices that enable the following actions in Continuous Deployment:

- Deploying a change to production
- Verifying proper operation of changes in the production environment
- Monitoring the production environment
- Responding to and recovering from production failures

It's important to remember that we separate deployment from release. We viewed the architectural concerns of enabling new changes to be automatically deployed to production environments in *Chapter 10*, *Continuous Exploration and Finding New Features*. While we continuously deploy to production, those changes are not visible to the customer until we release them. We talk about allowing only select people to view those changes in production in this chapter. We will talk about releasing on demand in our next chapter, *Chapter 13*, *Releasing on Demand to Realize Value*.

Let's begin our exploration of Continuous Deployment by looking at the first activity: deploying to the production environment.

Deploying to production

The goal of deployment is to get our solution in terms of a new product or enhancements to an existing product into a production environment. With Continuous Deployment, we want to move this solution as frequently as we can while minimizing the risk to our production environment.

The following practices allow us to deploy more frequently and reduce the risk of failures in the production environment:

- Setting up dark launches
- Employing feature flags
- Automating deployment
- Infrastructure as code
- Selective deployment
- Self-service deployment
- Version control
- Blue/green deployment

We have looked at version control in previous chapters, notably *Chapter 11*, *Continuous Integration of Solution Development*. We have also discussed blue/green deployment in that same chapter.

Let's examine the remaining practices and how they can increase deployment frequency while reducing risk.

Increasing deployment frequency

We saw in *Chapter 1*, *Introducing SAFe® and DevOps*, the set of practices that allowed Flickr to announce at the Velocity conference that they were able to perform 10 deploys in a day. They accomplished this with the creation of scripts that would deploy upon passage of all tests.

Today's automation and practices are an evolution of Flickr's initial successes with automated deployment. To that end, we will examine which practices are in use today and how.

Let's begin our exploration with a look at today's deployment automation.

Deployment automation

The scripting that allowed Flickr to automate their deployments reduced the time from code commit to deployment from days to hours. With the modern evolution of that scripting, the CI/CD pipeline further reduces the time of deployment from hours to minutes or even seconds. Let's see how that is done.

In the overall Continuous Delivery Pipeline, we saw the introduction of automation during Continuous Integration. The automation allows for building, testing, merging, and packaging operations without manual intervention as much as possible, as shown in the following diagram.

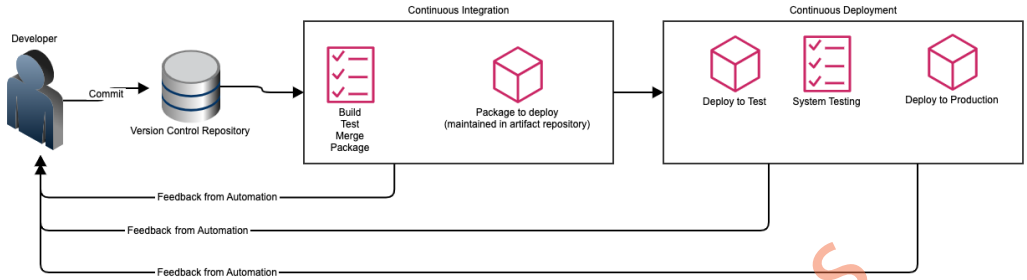


Figure 12.1 – CI and CD automation

We continue the use of automation in Continuous Deployment as illustrated in the preceding diagram. In Continuous Deployment, we take the application packaged at the tail end of Continuous Integration and deploy it to the test environment. After system-level testing is performed on the test environment, the application package is then deployed to the production environment. Such automation may be done by tools that combine Continuous Integration and Continuous Deployment, such as Jenkins and pipelines from GitLab and Bitbucket. Continuous Deployment automation may be separate from Continuous Integration in tools such as ArgoCD. Several pieces in terms of technology and practices help enable effective automation in Continuous Deployment.

The first important piece is version control. We've seen the importance of version control for text-based artifacts as the trigger for the CI/CD pipeline and as a way of connecting all artifacts together for understanding everything involved in a particular deployment or release.

Artifact repositories can act as version control for large, binary-based artifacts that can't be stored in text-based version control systems. They store intermediate builds, code libraries, virtual machine images, and Docker containers that may be created as part of a build process. If a component does not require a rebuild and test, its artifacts may be directly retrieved from the artifact repository, saving time and effort.

The size of changes is another factor that can also reduce the time automation takes to deploy to production. We've seen in *Chapter 4*, *Leveraging Lean Flow to Keep the Work Moving*, that batch size is an important factor in encouraging flow. A small independent change will go through much faster than a large change that may start a lot of rebuilding and testing in the CI/CD pipeline.

Automating as many steps as possible, including the push to deployment, will dramatically reduce the lead time for deployment. Allowing automation to proceed to further steps in building and testing throughout the Continuous Integration and Continuous Deployment stages keeps the momentum happening without delays because manual actions are needed to proceed to the next step. This may reduce the deployment time to seconds instead of minutes.

Infrastructure as code

A key part of deployment automation is the creation and configuration of resources in production environments. **Infrastructure as Code** (**IaC**) allows us to define the desired infrastructure in terms of the resources and their configuration in terms of text-based descriptions. These configuration files are used by tools in conjunction with configuration management to create new resources, update resource configurations, or even tear them down if required. As mentioned in *Chapter 3*, *Automation for Efficiency and Quality*, popular IaC tools include Hashicorp's Terraform and AWS CloudFormation for Amazon Web Services environments.

Version control plays a key part in establishing a smooth IaC process by ensuring that the evolution of configuration files is recorded and maintained. If a change to a product requires a change to the configuration in the production environment, the changes to the configuration file are created and tested in the staging environment. Tags in version control would serve to link together all artifacts associated with the change, from source code to tests to configuration file changes.

This process of building and testing the configuration files is no different than developing the product or its tests. This helps ensure that the resulting resources created in the production environment are reliable and in sync with any product changes.

Selective deployment

In some organizations, a production environment can be separated into multiple production environments. This separation can be based on some of the following factors:

- Infrastructure/resources
- Geography
- Customer

A selective deployment takes advantage of the separation by allowing deployment to happen at one instance of the production environment. An example of a selective deployment to an environment dedicated to a single customer is shown in the following illustration.

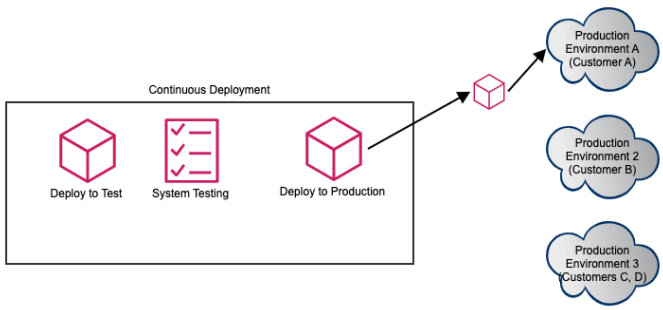


Figure 12.2 – Selective deployment to Customer A's production environment

Our preceding example allows for testing to occur in a limited production capacity with Customer A, while other customers in other production environments do not see the change. Deployment in one of several environments allows for a flexible release strategy, such as a canary release where changes are released to specific regions or customers before releasing to the entire customer base. We will discuss canary releases in more detail in *Chapter 13*, *Releasing on Demand to Realize Value*.

An example of real-world selective deployment happens at Facebook. As Facebook grew in popularity, the release engineering team kept up with the development activity by working on a *push from master* system that allowed for changes to be released more frequently. As detailed in `https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/`, deployments started with 50% Facebook employees, then 0.1% of the Facebook production traffic, eventually rolling out to 10% of Facebook production traffic.

Self-service deployment

There may be justifiable reasons that your organization cannot allow automation to deploy to your production environment. Generally, this is done to adhere to compliance policies.

If this is the case, self-service deployment (often called one-button deployment) allows anyone, usually developers, to deploy changes that have passed Continuous Integration into production. This method still uses automation to perform the actual deployment, so developers do not have unrestricted access to the production environment.

The deployment using automation is still recorded and audited so that the complete activities are tracked. This traceability of every automated deployment may give confidence to business compliance offices to allow a move to automating deployment.

Reducing risk

Jez Humble, in the book *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, famously said the following about deployment: "*If it hurts, do it more frequently, and bring the pain forward*." We have seen key methods for doing deployment more frequently. We will now look at practices that may help us deal with the pain of deployments.

The following practices allow us to mitigate the risk of introducing failures in the production environment:

- Dark launches
- Feature flags
- Blue/green deployments
- Version control

Blue/green deployments and version control were discussed in previous chapters. For this chapter, let's focus our exploration on the other mitigations.

Feature flags

Feature flags (or feature toggles) are the primary mechanism for implementing both dark launches and canary releases. By selecting the visibility of the feature in terms of whether the flag is enabled or disabled, you effectively separate the deployment of new features with the release of the feature where it is available to all users.

Feature flags also allow for quick rollbacks should problems arise in production. At the first sign of a problem, simply disable the feature flag.

When setting up feature flags, it is important to test the behavior of the feature flag at both positions: test the effects when the feature flag is enabled, and then when it is disabled. This testing should be done in the staging environment, well before it is deployed to production.

The presence of feature flags does add to the overall complexity of the solution by adding more permutations for testing behavior. Too many feature flags in play, especially obsolete feature flags for features long released, introduce more complexity in testing and technical debt. Obsolete feature flags should be removed at the first opportunity.

Dark launches

Dark launches allow features in the production environment to be visible to only developers, testers, and generally beta or select customers and not to the entire customer pool of users. To allow for dark launches, organizations typically use feature flags to allow or disallow the visibility of the feature based on the group that requires visibility.

An example of a dark launch for developers and testers using feature flags is detailed in the following illustration.

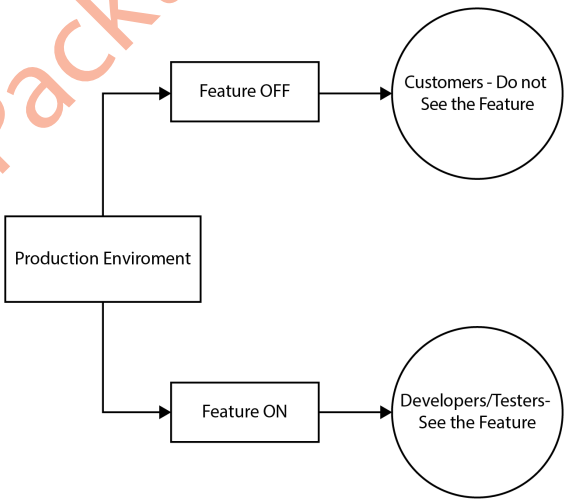


Figure 12.3 – Example of a dark launch using feature flags

In our preceding example, a new feature is visible to the developers and testers in the production environment. This visibility allows them to run experiments on this feature without affecting the customers, who do not see the feature.

Developers and testers can use dark launches in the following ways:

- Testing new application infrastructure
- Dynamically control which early customers can see new features
- Experimentation of new features

Dark launches are often synonymous with canary releases. The primary difference between a dark launch and a canary release is that the feature flag allows visibility to a select group of customers to determine their reaction to the new feature in canary releases.

We now need to see that these changes do not adversely affect our current production environment, even if these deployed changes are not released. To do this, we need to test. Let's examine our testing process in production.

Verifying proper operation

The changes to the product, in the form of new features, are now in the production environment, but this doesn't mean that the work is over. In fact, it has only begun. We need to see that our features function as we expect. This is true from a functional perspective, but also from other perspectives relating to NFRs.

The following practices allow us to verify correct behavior in the production environment:

- Production testing
- Test automation in production
- Test data management
- Testing NFRs

Some of these practices were introduced in the previous chapter but now that we are looking at the production environment, we can examine them closely. Let's look at all these practices and their application to production.

Production testing

With feature flags in place as part of a dark launch, testing can continue against the resources of the production environment. While testing in the production environment is no greater simulation of the conditions that the production environment has, the tester should still approach testing in the production environment with caution. Production testing offers the following advantages:

- Allows for monitoring application performance in real-time, realistic scenarios
- Monitoring application performance against real-time traffic
- Further detection of bugs and malicious attacks
- Helps maintain the quality of the application

The tests run in production really should be the ones near the top of the testing pyramid in combination with a subset of tests run in the previous stage of Continuous Integration as a sanity check. We are really looking to confirm that the behavior of the features we are about to release is what we expect. All other aspects of the testing pyramid should have been covered in the tests run in test environments and in the staging environment. Testing in production complements the earlier testing. It is not meant to substitute for earlier testing.

Successful testing in production requires a detailed understanding of the following factors:

- Use real browsers and devices in the production environment. Emulators or simulators may have been used in testing environments, but they may not exhibit the same behavior as the "real thing."
- Allow real traffic of the production environment for the measuring of application performance under load. This is, after all, the traffic the application will face.
- Use feature flags to allow for a small population of developers, testers, and beta customers to experience the new feature. Feature flags also allow for the quick disabling of features to everyone if problems occur.
- While production testing is occurring, the monitoring of the production environment must be continuous. This allows for the rapid shutdown of any tests at the first sign of a problem. In addition, reversion of test operations may be necessary.
- Use dedicated test user accounts so logging can determine what test transactions are versus real transactions.

A key test performed in production is an A/B test. In A/B testing, feature flags may direct beta customers to the new feature under test (option "A") to gauge whether that new option has a change in behavior over the current application available to the rest of the user population (option "B").

Test automation in production

Feature flag management is important when testing in production. Feature flags will not only determine whether a feature is visible, but also to whom that feature is visible.

A key use case of feature flag management occurs at Facebook. In the blog article at `https://www.facebook.com/notes/10158791573022200`, an engineer at Meta, Facebook's parent company, describes how they use Gatekeeper to establish A/B testing for every UI change with real

Facebook users. Gatekeeper ensures that real users are subjected to the testing of a single UI element and that the A/B tests don't conflict.

Because they are evaluating small changes, Facebook knows that some users may have a less than ideal user experience, but Facebook is striving for a better overall product. To that end, if enough users don't use the UI element change, it is regarded as a failed test and never makes its way to the entire Facebook population.

Test data management

While A/B testing may work to determine whether a given feature will be used by the end users, other testing may have to occur in production to see whether the desired flow and messaging occur for that feature.

Synthetic transactions use automated testing scripts to verify the end-to-end performance of our application in production. The scripts simulate the actions a user would make to complete a transaction. The synthetic transactions, and their responses made by the application, are recorded by synthetic monitoring tools. Allowing synthetic monitoring and testing with synthetic transactions allow testers behind feature flags to verify the following characteristics of the application:

- **Functionality**: Is the application moving through the correct pathways?
- **Availability**: Is the performance of the application in the production environment adequate?
- **Response time**: This is another measure of the application's performance in the production environment.

Synthetic monitoring allows you to understand the key flows of your application and how well they are performing. These may also provide important things to check with monitoring tools when released.

Testing NFRs

Having the new features deployed into the production environment while visible to a select few through feature flags allows for the evaluation of key NFRs before release.

Synthetic transactions and monitoring allow for performance testing while in the production environment. The reception of a sample of the real traffic in the production environment allows for vulnerability testing.

Testing of NFRs in the production environment is one of the important last checks of a feature before release. This is an important step that cannot be missed. Synthetic transactions and feature flags can allow for low-risk testing of NFRs to ensure that the production environment remains robust.

As we examined performing tests in the production environment, we have discovered that a pivotal part of the verification is continuous monitoring. We will now discover what is needed for continuous monitoring to occur.

Monitoring the production environment

Monitoring the production environment allows us to understand whether the NFRs of the environment are still being maintained as well as whether new functionality deployed as features are functional and performing based on the constraints identified in NFRs.

There are a few practices that are instrumental in providing continuous monitoring. Let's examine what these practices are and how to ensure that they are established in our production environment.

Full-stack telemetry

We need to monitor at various levels important measurements of our solution for a number of reasons. At the lower levels, we want to ensure that our production environment is stable and in no danger of failure. As we move to look at our system at higher levels, we want to ensure that we have measurements that allow us to determine whether our business hypothesis that started the development can be proven. Ultimately, at the business level, the measurements we take can gauge whether development is in line with our strategy or whether strategic alignments must be made.

These measurements taken for the range of levels we want are called full-stack telemetry. An example detailing the levels and sample measurements is in the following table.

Level for Measurement	Sample Measurements
Business	Value Stream KPIs (Cycle Time, Lead Time) and Solution KPIs (revenue, NPS, and conversion rate).
IT Service Management (ITSM)	Service Level Objectives and other Service KPIs (Server uptime and network availability).
Product or Solution	Garbage Collection metrics, Response Time, Availability, and Application logs.
Infrastructure	CPU Utilization, RAM Utilization, Networking metrics, and Event logs.

Table 12.1 – Sample measurements for levels of the solution

The required measurements must be designed during the Continuous Exploration stage of the Continuous Delivery Pipeline to ensure they can be easily collected during Continuous Deployment and beyond. The planned measurements should include data for both business use for measuring the benefit hypothesis as well as technical data that can tell us the state of the system in production.

Visual displays

Mountains of data can be collected through full-stack telemetry. What makes the data useful is organizing the data so that users can tell the state of the environment at a glance as well as identify when action must be taken quickly.

Dashboards provide a key way of visualizing data collected. The visualization is beneficial for identifying trends or understanding whether an important metric has exceeded a threshold that may indicate a failure in production.

The following is an example dashboard. This is the public dashboard for the cloud version of Grafana, a product used to create dashboards.

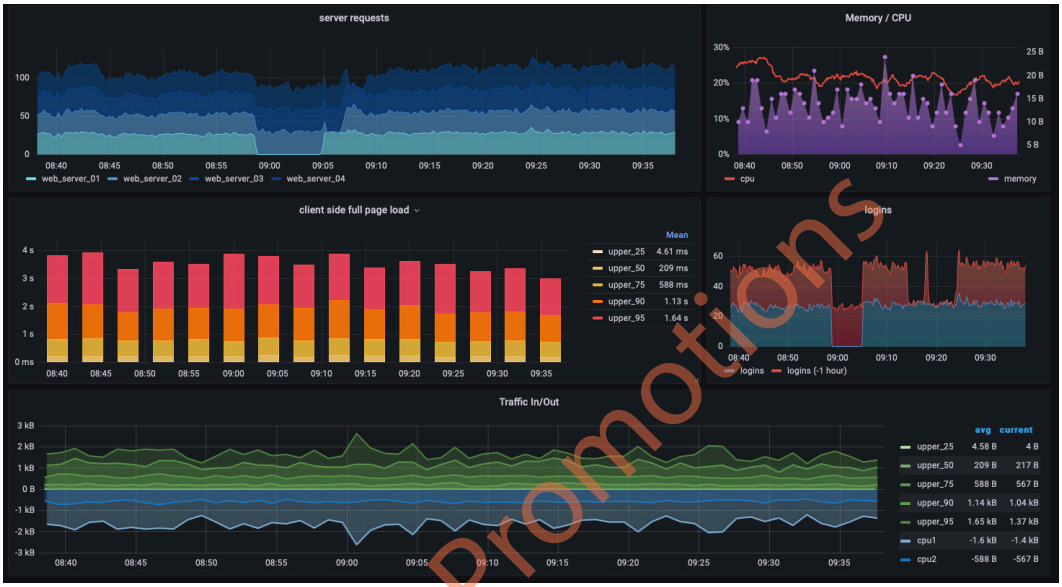


Figure 12.4 – Example dashboard from grafana.org

As important as it is to have a dashboard, it is just as important that the dashboards are visible to everyone in the organization. Transparency allows everyone on the value stream to have all the necessary information without waiting for approvals or debating whether they're *allowed* to see individual measurements.

Federated monitoring

Transparency in displaying data visualization may be difficult in complex organizations with many lines of business, business units, and other groups. To ensure information is not placed in silos, thought may be placed onto how to ensure the information is federated and easy to share and exchange. The following figure is an illustration of a dashboard that displays federated information.

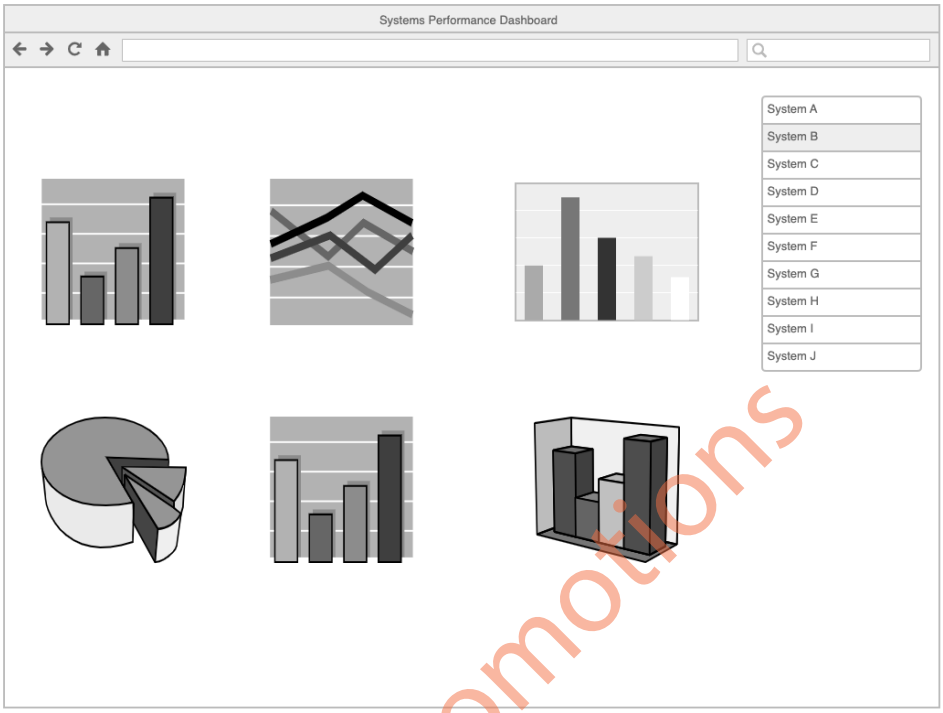


Figure 12.5 – Example of a federated dashboard

In the preceding figure, the dashboard displays the measurements for the selected system (System B). Information for the other systems is available by selecting the appropriate system in the control.

Sharing information in a federated structure allows for more transparency while encouraging agility in individual business units. This data from other sources should be combined with the local business unit information, allowing for a more holistic view of the business unit's system. Dashboards and other information display mechanisms can display the system data and allow ways to drill down and view the data and its source.

With the wide variety of data shared and visible on dashboards, organizations can continuously monitor their production environment. This can prepare them for production failures that may occur when releasing new features. Let's examine the practices they can employ when a production failure occurs.

Responding and recovering when disaster strikes

The ability to recover quickly is a key feature of a DevOps approach. One of the key DORA metrics is **Time to Restore Service**, with elite functioning DevOps organizations able to do so in minutes. Preparing for recovery is a major part of the CALMR approach.

To facilitate recovery, we look at the following practices:

- Proactive detection
- Cross-team collaboration
- Chaos engineering
- Session replay
- Rollback and fix forward
- Immutable infrastructure

A proactive response is important in production because this is the environment where the end users are. Problems here are visible and affect our customers. Problems not immediately handled can affect other work in other parts of the Continuous Delivery Pipeline.

Let's examine the practices that allow us to be proactive in the production environment.

Proactive detection

Because we are using feature flags to separate deployment from release, we can proactively test and look for problems without disrupting customers, or worse, having our customers discover the problems. Feature flags enabled for testers allow them to examine new features in the production environment.

With this unique opportunity to have new features in the production environment without disturbing customer traffic, testers can perform additional testing, employ "what-if" scenarios, and eventually plan disaster recovery procedures involving the release of the new feature.

Cross-team collaboration

Problems that are visible to the customer and that have little information to start with may be a breeding ground for people to blame one another. The pressure increases on a very visible problem with feedback from angry customers. A solution may not be apparent early on. It is this type of stress that may test the idea of cross-team collaboration working, but it has shown itself the most effective way to solve such problems.

Having people from various disciplines in Development and Operations was one of the keys to the success at Flickr that we originally discussed in *Chapter 1*, *Introducing SAFe® and DevOps*. This collaboration across the different groups is still key to this day.

To have true collaboration, we need to work toward the mission-based, psychologically-safe generative culture we identified in *Chapter 2*, *Culture of Shared Responsibility*. With the movement to that type of culture, collective ownership happens and teams work together across disciplines to identify the root cause of problems and find solutions quickly. The problems then become opportunities for learning.

Chaos engineering

A worthwhile exercise that you can do proactively before releasing the feature is to perform a chaos engineering exercise, taking advantage of the feature flag so the effects are not seen by active customers.

The most famous example of chaos engineering is done by Netflix. As detailed in the blog article at `https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116`, Netflix runs a set of tools they call the Simian Army to simulate production outages in the Amazon Web Services cloud. The most famous of these is Chaos Monkey, which simulates a virtual server going down. These experiments are run during the business day, with plenty of engineers watching and addressing problems found.

You can run an exercise that is similar to an execution of Chaos Monkey. Run the experiment to simulate an outage condition to see whether your new feature is resilient enough. At the end of the experiment, determine the next steps with a debrief.

The complete approach is detailed in *Chapter 6*, *Recovering from Production Failures*. Chaos engineering exercises can be performed on existing features that have been already released if there is sufficient time available in the error budget. Exercises can be performed on unreleased features with robust feature flag management.

Session replay

A useful tool for troubleshooting is session replay. Session replay is the ability to record individual users' transactions and replay those transactions. The ability to perform session replay has the following benefits:

- Developers can understand where users may have problems with the usability of a website and it gives insight on how users are really using a given feature
- Developers can see the actions used to perform a fraudulent transaction on a website, which can lead to closing vulnerabilities
- For a production failure, developers can see the exact sequence of actions that users perform to cause the failure

Session replay that is performed on the client side presents the action from the end user's point of view. These tools allow developers to see the position of the cursor, what clicks are made, and what gets typed in a video-like replay session. Dynatrace and Datadog are examples of tools that offer session replay.

Server-side-based session replay captures all website traffic and includes input of what gets typed and what gets clicked. Scrolling and mouse movements are not captured.

When using session replay, attention must be given to the session data. Such data typically contains private information such as passwords and may require large amounts of storage capability.

Rollback and fix forward

When production failures do occur, two quick methods to return to a stable environment include rollback and fix forward.

Rollback involves reverting the production to a previous iteration, one that doesn't include the most recent change that probably caused the production failure. As we saw in *Chapter 6*, *Recovering from Production Failures*, two common methods of rolling back are blue/green deployments and feature flags.

Blue/green deployment is an easy way to roll back a production environment to an idle environment and the formerly idle environment back to the active state. Attention must be paid to components that represent state, such as databases or volatile storage. Those components require careful reversion when the transition back occurs.

Feature flags are an easy way to remove the visibility of the new feature when its release causes a production failure. A toggle of a feature flag back to off does not require an extensive code or configuration change.

Fixing forward is another method for getting the production environment stable after a production failure. To fix forward, you develop and propagate the fix for the failure through the Continuous Delivery Pipeline so that it is deployed and released to production. When performing a fix forward, it is recommended that you use the standard deployment process through the Continuous Delivery Pipeline and not skip any testing. Bypassing testing to create a "quick fix" can lead to greater technical debt.

Immutable architecture

A primary reason for automating deployment to testing, staging, and ultimately the production environment is to make the architecture immutable. That is, any change in the environment cannot be made manually.

Any change in any environment must be made through the Continuous Delivery Pipeline and every artifact needed for that change must be recorded in version control. The close coupling with version control and the Continuous Delivery Pipeline prevents configuration drift or the difference in changes between environments.

Summary

In this chapter, we continued our exploration of the Continuous Delivery Pipeline into the production environment. Our feature after having finished design in Continuous Exploration and development and testing in Continuous Integration, now finds itself ready for deployment to production. Automation plays a key role here in executing the steps to bring the change into the production environment, possibly using IaC to create and configure new production resources.

Even with the new change in production, testing is performed to build confidence before release. Feature flags allow engineers and select beta customers to perform testing on new changes in production

while concealed from the general user population. Test data in the form of synthetic transactions allow functional testing and testing of NFRs to occur.

Monitoring in the production environment allows us to see the success or failure of the testing in production. We want to ensure we are looking at the correct measurements from system resources all the way to those metrics that may serve as leading indicators that the changes we want are realizing our benefit hypothesis. We want this data visible on dashboards and transparent to everyone. If monitoring indicates problems in production, we are ready to act. The entire value stream works together to find the root cause. We can roll back to a previous version or fix the problem and propagate the fix using the Continuous Delivery Pipeline.

Our change is now in the production environment. We now wait for the last event: releasing to our users so they can use it to their advantage. For this, we will examine the last stage of our Continuous Delivery Pipeline, release on demand, in our next chapter.

Questions

1. What helps reduce the lead time to deploy to production? (pick 3)

    A. Unit testing
    B. Small batches of change
    C. Version control
    D. Behavior-driven development
    E. Automating deployment

2. What practice allows you to perform canary releases?

    A. Infrastructure as code
    B. Blue/green development
    C. Selective deployment
    D. Self-service deployment

3. Feature flags allow … (pick 3)

    A. Testers to view unreleased features in production
    B. You to run unit tests faster
    C. Developers to start a deployment to production
    D. Rollback of a new feature in the event of a production failure
    E. A select group of customers to do A/B testing
    F. The execution of a CI/CD pipeline

4. Running a synthetic transaction in production can help measure … (pick 3)

   A. Cycle time
   B. Functionality
   C. Scalability
   D. Availability
   E. Response time

5. What levels of operation should full-stack telemetry measure?

   A. IT service management
   B. Business
   C. Solution
   D. Infrastructure
   E. All of the above

6. What information can be played back in a server-side session replay?

   A. Scrolling down the web page
   B. Input fields on a web form
   C. Moving the mouse cursor from left to right
   D. Horizontal scrolling to a button

7. How are changes to production made in an immutable architecture?

   A. The administrator changes a file in the production environment.
   B. The administrator changes a configuration file and executes the IaC tool to create the change.
   C. The administrator changes a configuration file, submits the change into version control, and executes the CI/CD pipeline.
   D. The administrator restarts the production server.

8. Name two practices that help enable immutable architecture.

   A. Feature flags
   B. Version control
   C. CI/CD pipeline
   D. Blue/green deployment
   E. Behavior-driven development

Further reading

- A summary from Scaled Agile feature guidance on Continuous Deployment in the Continuous Delivery Pipeline: `https://www.scaledagileframework.com/continuous-deployment/`
- A look from a Meta (Facebook's parent company) engineer describing how Facebook performs testing, deployment, and release: `https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/`
- *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* by Jez Humble and David Farley – The authoritative guide on creating a CI/CD pipeline and detailing integration, deployment, and release.
- A blog article from LaunchDarkly, a vendor for feature flag management, describing the uses and benefits of feature flags: `https://launchdarkly.com/blog/guide-to-dark-launching/`
- An article describing the advantages of performing testing in production: `https://www.softwaretestingmaterial.com/testing-in-production/`
- A detailed blog article describing the uses and advantages of testing in production: `https://www.tothenew.com/blog/testing-in-production-environment-what-why-and-how/`
- An article describing how testing is done in production at Facebook using Gatekeeper to monitor the user tests in production: `https://www.facebook.com/notes/10158791573022200/`
- A blog article describing the use of synthetic transactions in testing new features: `https://www.netreo.com/blog/synthetic-transactions/`
- A blog article from Netflix engineers describing how they perform chaos engineering with the set of tools they call "the Simian Army": `https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116`

13

Releasing on Demand to Realize Value

We have now reached the end of our journey through the Continuous Delivery Pipeline. We started with a benefit hypothesis to deliver value to our customers and turned it into features to develop in Continuous Exploration. In Continuous Integration, we developed our feature, story by story, and applied those changes into version control, which built and tested the change using the automation of the Continuous Delivery Pipeline until it was ready to go to the production environment. In Continuous Deployment, we propagated our changes to the production environment, keeping the change hidden from the general user population, until we were ready to release.

Now we are ready to release our change to customers. Releasing our change on-demand involves the following four activities:

- Releasing that value to customers
- Stabilizing our solution in operations
- Measuring the value
- Learning the outcomes

Let's begin by looking at the release process.

Releasing value to customers

Up to this point, we have our changes in the production environment, testing them to ensure functionality, security, and reliability. Now we are ready to release. We want to release our changes to the customer for the following reasons:

- We think the timing is right for the customer to take advantage and when the organization thinks there is high market demand
- We have confidence that the change will not have a negative impact on the production environment

Even with those reasons, we may not want to introduce our release all at once. On April 23, 1985, the Coca-Cola Company announced the first major change to the formula for its flagship soft drink. *New Coke* had succeeded in over 200,000 blind taste tests against Pepsi, Coke's chief competitor. However, upon release, the reaction was swift and negative. The outcry against the new formula had forced Coca-Cola to reintroduce the original formula as *Coca-Cola Classic* after only 79 days. Since that time, companies have used progressive releases before releasing to the entire market.

If we want to approach releasing incrementally and progressively, we will use the following practices:

- Feature flags
- Dark launches
- Decoupling releases by component architecture
- Canary releases

We have previously examined feature flags and dark launches in our previous chapter, *Chapter 12*, *Continuous Deployment to Production*. Let's take a look at the other practices of decoupling releases by component architecture and canary releases.

Decoupling releases by component architecture

In *Chapter 10*, *Continuous Exploration and Finding New Features*, we talked about how one of the key activities in looking at new features to develop was architecting the solution. A part of that is allowing for releasability to meet the organization's business priorities.

One way of achieving this releasability is to architect your product or solution into major decoupled components. These components can have their own separate release cadences.

In *Chapter 2*, *Culture of Shared Responsibility*, we first introduced the idea of operational and development value streams. We discussed how development value streams designed, developed, tested, released, and maintained a product or solution and we identified several development value streams that had solutions that the Operational Value Stream of our video streaming service relied on.

Turning back to this example, let's examine one of those development value streams, one that maintains the mobile application. This value stream has several components, each with a different release cadence as shown by the following diagram.

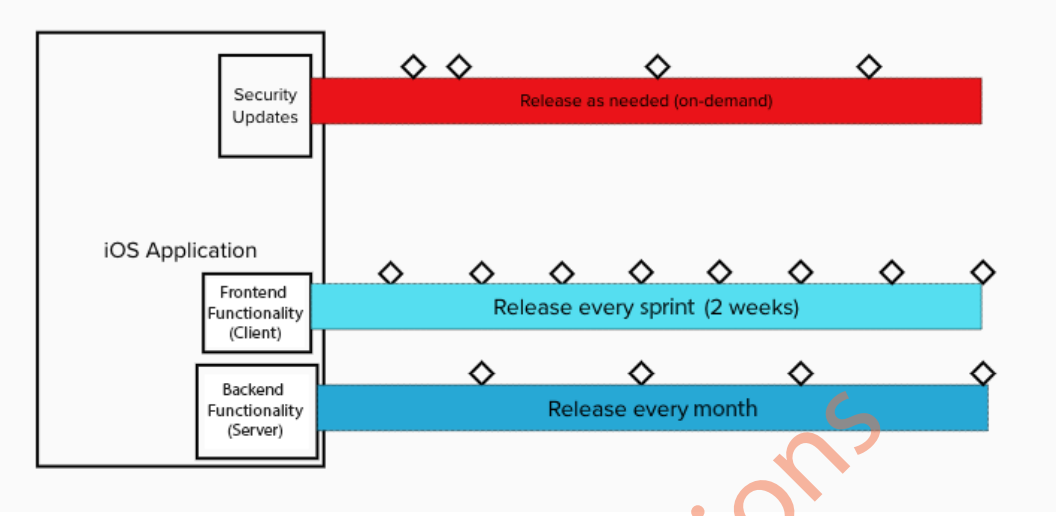


Figure 13.1 – Decoupled release schedule for mobile application value stream

In our mobile application value stream example, we release security updates as fixes to vulnerabilities appear, after moving them through the Continuous Delivery Pipeline.

One component is the interface and logic seen on the mobile devices themselves, otherwise known as the frontend. The development here can be released on a quick cadence, effectively at the end of every sprint.

The other component deals with the logic and processing found on the streaming service's data centers or cloud, known as the backend. In this example, releases for that component occur every month.

Canary releases

The term canary release comes from the practice in mining of carrying a canary into the coal mine. The canary would act as a warning of the presence of toxic gases. Because of its small size, if it died in the coal mine, a toxic gas was present, and the miners should evacuate immediately.

In terms of modern product development, a canary release is the release of a product or new features to a small select group of customers to get their feedback on a release before the entire user population.

To set up a canary release, feature flags are used again to route who receives visibility into the change in the production environment. This feature flag configuration is shown in the following diagram.

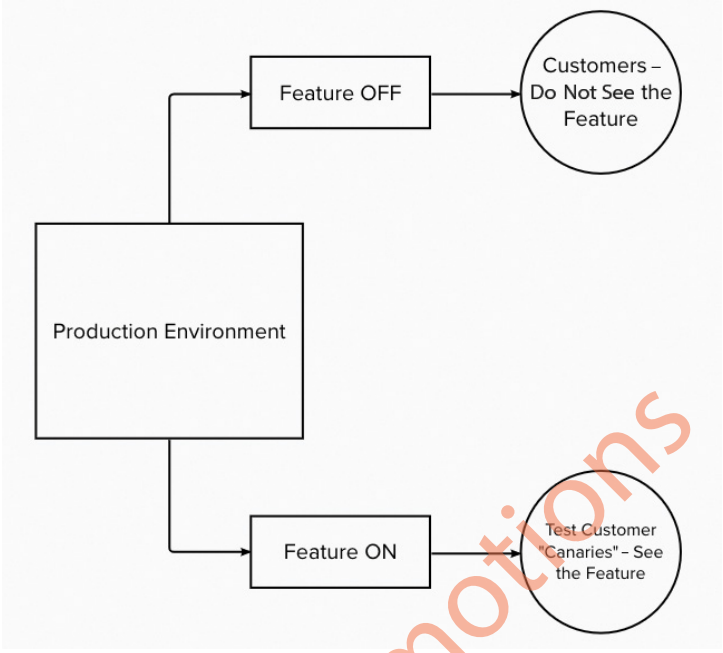


Figure 13.2 – Canary release configuration using feature flags

Another possible way to perform a canary release is in distributed production environments. If the production environments are in different geographic regions, the changes are released in one production environment to try on one set of users, while the other production environments remain on their versions. If all goes well, eventually the environments in the remaining regions are upgraded.

Canary releases offer the advantage that they allow for A/B testing where the *A* group that receives the change can be measured against the *B* group or control group to see whether the new change creates a desired change in user behavior. Running canary releases as experimentation does require the ability to measure user and system behavior as part of full-stack telemetry.

There may be situations where canary releases should not be done. These may include the following reasons:

- If the solution is part of a mission-critical, medical, or safety system where there is low tolerance for failure
- If the end users will react negatively to being *guinea pigs* or treated as beta testers
- If the changes require changes on backend configurations such as database schemas that are not compatible with the current production version

As we progress in the release from our initial canary users to the entire user population, we want to be able to ensure that our production environment remains resilient. This may require us to stabilize our solution and ensure proper operation. We will examine the steps needed for this in our next section.

Stabilizing and operating the solution

Our goal is to ensure that our production environment remains stable, is resilient to handle the new changes, and that we continue to have sustainable value delivery. To maintain this activity, we want to apply the following practices:

- Site reliability engineering
- Failover and disaster recovery
- Continuous Security Monitoring
- Architecting for operations
- Monitoring NFRs

We have previously looked at testing and monitoring NFRs in *Chapter 12*, *Continuous Deployment to Production*. Let's examine the remaining practices.

Site reliability engineering

We first learned about **Site Reliability Engineering** (**SRE**) in *Chapter 6*, *Recovering from Production Failures*. In that chapter, we saw the following four practices that site reliability engineers use to maintain the production environment when high availability is required for large scaled systems:

- Formulation of an error budget using **Service Level Indicators** (**SLI**) and **Service Level Objectives** (**SLO**)
- Creating standards for release through release engineering
- Collaborating on product launches with launch coordination engineering
- Practicing recovery with chaos engineering and incident management procedures

In *Chapter 6*, we saw that if availability SLOs were at *four-nines* (99.99% availability or higher), the monthly allowable would be 4 minutes, 23 seconds. To maintain that availability, SREs use the previously mentioned practices to ensure reliability and have standard incident management policies defined and rehearsed to minimize downtimes when problems do occur.

Other principles and practices adopted by Google can give us a better picture of the discipline of SRE and how it contributes to the DevOps approach. To understand these additional principles and practices, it may be necessary to look at the origins of SRE at Google.

SRE started at Google in 2004 by Ben Treynor Sloss. His original view was to rethink how operations were performed by system administration. He wanted to approach problems found in operations from a software development perspective. From that perspective, the following principles emerged in addition to the preceding practices:

- Eliminating toil, that is, finding the repetitive tasks and seeing whether they could be eliminated

- Increased use of automation as a way of cost effectively eliminating toil
- Monitoring every aspect of the production environment, which leads to observability

The people Sloss enlisted for his initial SRE team would spend half of their time in development and the other half in operations to follow changes they developed from beginning to end. This allowed them to develop skills necessary for operations as well as maintain their development expertise.

Since 2004, the number of practices that have been pulled into the discipline of SRE has expanded as technology has evolved. Nevertheless, many site reliability engineers have kept to the principles previously outlined. Adopting these principles and practices may provide tangible benefits when reliability is a key nonfunctional requirement.

Failover and disaster recovery

Murphy's Law famously states that "*anything that can go wrong, will*." In that sense, it's not *if* a disaster will strike your system, but *when*. We've looked at ways of preventing disaster and using chaos engineering to simulate disasters, but are there other ways of preparing for disaster?

Disaster recovery focuses on ensuring that the technical aspects that are important to a business are restored as quickly as possible should a natural or manmade disaster strike. Disaster recovery incorporates the following elements to prepare for the worst:

- **Disaster recovery team**: A group of individuals that are responsible for creating, implementing, and managing a disaster recovery plan. The disaster recovery plan outlines the responsibilities for the team to follow in an emergency, including communication with other employees and customers.
- **Risk evaluation**: The disaster recovery team should identify the possible scenarios and the appropriate responses for each. For example, if a cyberattack happened, what would the steps be in the disaster recovery plan?
- **Asset identification and evaluation**: The disaster recovery team should identify all systems, applications, data, and other resources. Part of this identification includes how important they are for business continuity as well as instructions for restoring them.
- **Resource backups**: The disaster recovery plan should identify what resources should be backed up, the frequency of backup, where those backups are stored, and for how long the backups are kept.
- **Dress rehearsals**: All parts of the disaster recovery plan should be practiced regularly. Restores of backups should be attempted to find flaws in the backup process and to determine whether the backups are sound. Any flaws found while rehearsing should be fixed to improve the disaster recovery plan. Rehearsals should examine evolving threats to see whether new measures should be added to the disaster recovery plan.

The disaster recovery plan will look at the following two measurements as goals to determine the overall strategy:

- **Recovery Point Objective** (**RPO**) is a measure of the state of the data, measured from the time of the last backup when the resource of record is restored
- **Recovery Time Objective** (**RTO**) is a measure of the allowable downtime after a disaster

Let's explain these objectives using an example shown in the following diagram.

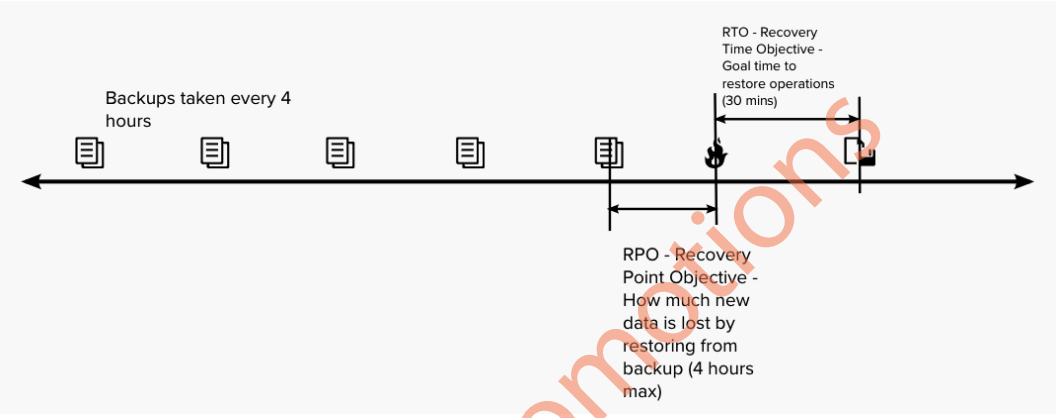


Figure 13.3 – Illustration of RPO and RTO

In the preceding diagram, we take a backup of our resource every four hours. We have practiced performing disaster recovery on our resource and have reliably restored operations in 30 minutes. When disaster strikes, if it is something familiar and has been rehearsed, our Time to Restore Service will match our RTO objective and be 30 minutes.

For the RPO, we need to see the point of the last backup. In our example, the time difference between the last backup and the disaster could be as large as four hours. The higher the frequency of backups, the lower the RPO.

The mechanisms for disaster recovery can take many forms. Organizations may opt to use one or a combination of the following methods:

- **Backups**: Backups are the simplest form of disaster recovery. Note that this ensures that the data is kept safe; this does nothing for infrastructure.
- **Cold site**: A redundant second production environment. This allows for business continuity, but there is no way to restore data. A blue/green deployment is an example of a cold site.
- **Hot site**: A redundant second production environment that has its data regularly synchronized with the active production environment.

- **Disaster Recovery as a Service (DRaaS)**: A vendor moves an organization's processing capability from the organization to its own (often cloud-based) infrastructure.
- **Backup as a Service (BaaS)**: Backups are taken and reside off-site or stored in a cloud infrastructure by a third-party provider.
- **Data center disaster recovery**: These are devices on the organization's premises used to deal with disasters such as fire or power loss. Examples of these include backup power generators or fire suppression equipment.

Continuous Security Monitoring

In previous stages of our Continuous Delivery pipeline, our focus on security was prevention. We wanted to ensure that the changes we designed and developed did not introduce security vulnerabilities. So, the automated security testing we performed on the pipeline examined the code changes.

With the code released, we shift our focus from prevention to detecting threats that come from malicious actors. We look for breaches or attacks using currently unknown vulnerabilities.

The **National Institute of Standards and Technology** (**NIST**) looks at **Continuous Security Monitoring** (**CSM**) as Information Security Continuous Monitoring. In a white paper published in September 2011 (`https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-137.pdf`), they describe Information Security Continuous Monitoring as the following:

Information security continuous monitoring (ISCM) is defined as maintaining ongoing awareness of information security, vulnerabilities, and threats to support organizational risk management decisions.

The white paper further defines a process to implement CSM by incorporating the following steps:

1. Defining a strategy by looking at risk tolerance including visibility into assets, vulnerability awareness, current threat information, and impacts on the mission or business.
2. Establishing a program including definitions of metrics, status monitoring frequency, and the technical architecture.
3. Implementing the program and collecting security-related information for metrics, assessments, and reporting. Automate the collection, analysis, and reporting as much as possible.
4. Analyze the findings and report.
5. Respond to the findings.
6. Review and update the monitoring program.

The assets that need to be monitored include not only ones directly maintained by the organization but may also extend to third parties and vendors. Monitoring tools may look at the following assets:

- **Known assets**: Assets that are part of an organization's inventory

- **Unknown assets**: Forgotten assets including development websites or old marketing sites
- **Rogue assets**: Assets created by malicious actors that may impersonate the organization's domain
- **Vendor assets**: Assets owned by third-party vendors

Once identified, the assets should be examined for possible threats and vulnerabilities. A sampling of common threats and vulnerabilities are the following:

- **Unnecessary open TCP/UDP ports**: Any open ports may pose a problem if the service that communicates on those ports is misconfigured or unpatched, potentially allowing a vulnerability.
- **Man-in-the-middle attacks**: This is a cyber attack where the attacker is between two parties connected together. The parties believe they have a direct connection, but the attacker may listen in and even change the messages before transmitting to the other party.
- **Poor email security**: This may leave your organization open to email spoofing.
- **Domain hijacking**: An attacker changes the registration of an organization's domain name without permission of the domain's owner.
- **Cross-site scripting (XSS) vulnerabilities**: Attackers can inject client-side scripts on web pages allowing access control.
- **Leaked credentials**: Discovered through data breaches, they allow attackers access into an organization.
- **Data leaks**: Exposure of private or sensitive data.
- **Typosquatted domains**: This is a form of cybersquatting where attackers claim a domain name similar to a known organization's domain name in hopes that someone will incorrectly type a URL and enter the attacker's site.

Identification of these attacks may be part of an assessment created after automated monitoring. The mitigation steps and action will outline who in the organization will perform the remediation steps and coordinate the response.

Architecting for operations

The support activities performed in this stage of the Continuous Delivery Pipeline will have a profound effect on the architecture of the system and may even drive the future direction of the product or solution. These things may be part of the architectural decisions the system architect makes when looking at new capabilities in the Continuous Exploration stage of the Continuous Delivery Pipeline.

Decisions found in this stage that may be taken to Continuous Exploration include the following items:

- Fixes and new automation created by the site reliability engineers

- Changes because of flaws discovered in the disaster recovery plan that affect the configuration of the test, staging, or production environments.
- New vulnerabilities that have been discovered by CSM that the product's architecture must prevent from occurring in the future.

The system architect thus becomes the balance point of intentional architecture, overseeing the desired architecture of the product or solution, and of emergent design, where other factors such as the environment play a role in requiring changes to the architecture.

The learning at this stage is not limited to the architectural aspects. We also have to evaluate from a product performance standpoint whether the benefit hypothesis was fulfilled by our development. For this, we need to measure our solution's value. Let's look at this in our next section.

Measuring the value

Throughout the design and development journey through the Continuous Delivery Pipeline, we have subjected our changes to an array of testing. We will now look at the final test to answer the question: is our development effort bringing value to the customers to the point that this benefits both the customer and the organization?

To aid us in answering this question, we will look at the following activities:

- Innovation accounting
- Proving/disproving the benefit hypothesis

We will first revisit innovation accounting and its source: the Lean Startup Cycle. Based on that knowledge, we will see how leading and lagging indicators prove or disprove the benefit hypothesis we created in Continuous Exploration.

Innovation accounting

We first saw innovation accounting in *Chapter 5*, *Measuring the Process and Solution*. In the chapter, according to *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses* by Eric Ries, we saw that measures were important to gauge whether the benefit hypothesis had been proven.

Ries expands upon innovation accounting in his follow-up book, *The Startup Way: How Modern Companies Use Entrepreneurial Management to Transform Culture and Drive Long-Term Growth*. In his book, he gives the following definition for innovation accounting:

Innovation Accounting (IA) is a way of evaluating progress when all the metrics typically used in an established company (revenue, customers, ROI, market share) are effectively zero.

He proposes three levels of innovation accounting, each with a different set of metrics to collect. Let's look at these levels now.

Level 1: Dashboard metrics

These metrics serve as a starting point. Ries advises setting up a dashboard. On this initial dashboard are customer-facing metrics the development teams think are important. These types of metrics are based on per-customer input. The following per-customer metrics for learning include the following:

- Conversion rates (usually the percentage of customers who move from a free version to a paid version of a product)
- Revenue per customer
- Lifetime value per customer
- Retention rate
- Cost per customer
- Referral rate
- Channel adoption

Metrics like these help enforce the idea that development can affect these metrics after observing the outcomes of their efforts. Metrics here that are visible help keep the development process aligned with feedback.

Level 2: Business case metrics

We proceed deeper with a different set of metrics in Level 2. Here, we try to quantify the "*leap of faith assumptions*" as Ries calls them. Leap of faith assumptions come in the following two categories:

- **Value assumptions**: These describe the value that users receive from the product or solution
- **Growth assumptions**: How do new users find the product?

These types of assumptions are needed for any new development, otherwise nothing new would be developed. These assumptions get tested by the development of the MVP and other validated exercises.

The following metrics illustrate the assumptions and categories of assumptions:

- Retention rate (value assumption)
- Referral rates (value assumption)
- Word-of-mouth referrals (growth assumption)

Value assumptions look for customer behavior. They are met based on positive user behavior. Growth assumptions are looking for sustainable growth.

Level 3: Net present value

At this level, we look at performance over a long period of time. Changes will come as you acquire new data and re-evaluate or compare the present data over what was forecasted. You look at long-term drivers of the product's future performance.

The following metrics may provide guidance over the long term:

- Number of website users
- Percentage of visitors that become users
- Percentage of paid users
- Average price paid by a user

These metrics often involve more than the development teams. Finance may also be involved with the goal of shifting the focus at this point to the financial performance of the product.

Proving/disproving the benefit hypothesis

Now that we've seen the steps taken with innovation accounting, let's see how they play against the benefit hypothesis we created in Continuous Exploration.

In our journey through the Continuous Delivery Pipeline, we created our benefit hypothesis in Continuous Exploration. To measure the validity of our hypothesis, we may be incorporating Level 2 metrics in our dashboard. Our dashboard is acquiring the metrics automatically through the full-stack telemetry that we have designed into our system in our test, staging, and production environments.

We may get some indications from testing done at Continuous Deployment, but the real measurements will come from Release on Demand, first during any A/B testing or canary releases, but also upon general release. We want to see all correlations between the performance and our initial benefit hypothesis.

The data we collect and analyze now in this step may not be just to improve the product, but also to improve our value stream and our development process. Let's explore our next section to see how both are accomplished.

Learning from the outcomes

Based on our learning, we now need to figure what's the best next step. This is true from the perspective of our product as well as the perspective of our value stream.

For our product, it's a matter of determining the best direction. This may mean whether it's time to pivot, change direction in our overall product strategy, or persevere or continue in the same direction.

For our value stream, this time is used to reflect on how to improve. What lessons are there that we can learn from to improve?

The following practices are used to determine future product direction as well as future direction of our value stream:

- Pivot or persevere from Lean Startup
- Relentless improvement
- Value stream mapping sessions

We perform this learning so we can begin again with renewed focus at the start of the Continuous Delivery Pipeline in setting up ideas to execute. We improve our value streams to improve the performance of our Continuous Delivery Pipeline.

Let's look at these practices that allow us to improve.

Pivot or persevere

In *Chapter 10*, *Continuous Exploration and Finding New Features*, we viewed the application of Eric Ries's Build-Measure-Learn cycle into the SAFe® Lean Startup cycle where we saw how Epics are created with a benefit hypothesis. The Epics are implemented with a **Minimum Viable Product** (**MVP**), which acts as an experiment to prove or disprove the benefit hypothesis. The MVP is evaluated through innovation accounting and tracking of leading indicators that will be used to determine the pivot or persevere decision.

We are now at that point of that decision. For our ART, the MVP may be a few features created at the beginning of the Continuous Delivery Pipeline. We have developed these features, tested them, and deployed them into the production environment through the Continuous Integration and Continuous Deployment stages of our pipeline. Now, in Release on Demand, we show our MVP as features to the user population to see whether the benefit hypothesis is proven. Based on the innovation accounting metrics we are collecting through our full-stack telemetry, we come to the following two decisions:

- **Pivot**: Our feature didn't meet the benefit hypothesis. It's time to move in a different direction. This may include stopping development in that product direction.
- **Persevere**: Our benefit hypothesis was validated. We should continue to develop further features to enhance our MVP.

Note that even after our persevere decision for the MVP, our features will still be evaluated to determine that they prove to be valuable and that the product direction still resonates with our customers.

Relentless improvement

We first saw relentless improvement when we first examined the SAFe House of Lean in *Chapter 2, Culture of Shared Responsibility*, We mentioned that in relentless improvement, we looked for opportunities to be better because of that "hidden sense of danger."

Throughout development, the teams and the ART have looked for such opportunities to improve the flow of value. The teams have been regularly holding retrospectives at the end of each iteration to identify problems at the team level and ART holds the **Inspect and Adapt** (**I&A**) event at the end of the Program Increment to look at systemic issues.

Other improvements may come to the Continuous Delivery Pipeline itself. Newer tools, additional testing, and continued maintenance allow the ART to maintain or improve how they create the experiments to validate benefit hypotheses.

Additional value stream mapping sessions

Another important part of relentless improvement comes from Value Stream Management and continuous learning, ideas originally discussed in *Chapter 9*, *Moving to the Future with Continuous Learning*.

One activity we originally performed in our value stream mapping was not only mapping our value stream as it stands currently, but also identifying an ideal *future-state* value stream. Improvement actions could come from small, iterative changes toward that ideal value stream.

One other step for improvement is to hold a value stream mapping session at least once a year to evaluate the value stream in its current iteration. This allows the ART to view the present bottlenecks impeding the flow of value. During this value stream mapping session, a new future-state value stream can be identified for new improvements.

Summary

In this chapter, we have reached the last stage of the Continuous Delivery Pipeline: Release on Demand. After using feature flags to allow testers to view new changes in the development environment, we can use them to incrementally release those changes to a small population of users in a canary release. We may also want to set up our architecture to allow for each component to have different release cadences. After release, we want to ensure that the changes don't disrupt the environment and that our overall solution is stable. To do that, we adhere to the principles and primary practices of SRE, including preparing for disaster recovery.

With the changes released in a stable environment, it's time to measure business results through full-stack telemetry. We selected our measures during Continuous Exploration, relying on innovation accounting principles. Looking at these measures on dashboards visible to everyone, we can try to determine whether our benefit hypothesis that we created in Continuous Exploration is valid.

Based on whether the benefit hypothesis is valid or not, we must decide to pivot, change direction, or persevere and continue developing in the same product direction. After making that decision, the ART looks at other improvement opportunities through retrospectives, I&A, and regularly mapping their value stream.

This brings us to a close on *Part 3*. We now will take a look at emerging trends as well as some tips and tricks for success in your DevOps adoption in our last chapter of the book.

Questions

1. Which of these are key practices or principles of SRE? (pick 3)

   A. Use feature flags for A/B testing
   B. Reduce toil mainly through automation
   C. Run unit tests in production
   D. Know how much risk you can handle through error budgets
   E. Chaos engineering
   F. Test in production

2. Which of these should be part of a disaster recovery plan? (pick 3)

   A. Disaster recovery team identified
   B. Resource identification
   C. Vendor contact information
   D. Server schematics
   E. Backup schedule
   F. Hard-copy versions of important files

3. Backups are performed on the database every two hours. If the database server crashes, what is the expected RPO?

   A. 2 hours
   B. 4 hours
   C. 8 hours
   D. 16 hours

4. What problems can CSM detect? (pick 2)

   A. Man-in-the-middle attacks
   B. License violations
   C. Cross-site Scripting (XSS) vulnerabilities
   D. Weak passwords

5. What practices are NOT examples of relentless improvement? (pick 2)

   A. Retrospectives
   B. Code reviews

C. Inspect and Adapt (I&A)

D. Adding tests to the Continuous Delivery Pipeline

E. Code comments

Further reading

- An account of the introduction of New Coke: `https://www.coca-colacompany.com/company/history/the-story-of-one-of-the-most-memorable-marketing-blunders-ever`
- Another account of the introduction of New Coke, with an additional perspective of the competitive landscape: `https://www.vox.com/2015/4/23/8472539/new-coke-cola-wars`
- `https://blog.getambassador.io/cloud-native-patterns-canary-release-1cb8f82d371a` – Blog article that defines canary releases and details implementation, benefits, and examples of use
- An interesting reference on canary releases: `https://martinfowler.com/bliki/CanaryRelease.html`
- Blog article from LaunchDarkly, a vendor for feature flag management solutions, about using feature flags for canary releases: `https://launchdarkly.com/blog/what-is-a-canary-release/`
- YouTube video from Google detailing the history behind site reliability engineering including perspectives from Ben Treynor Sloss: `https://www.youtube.com/watch?v=1NF6N2RwVoc&t=6s`
- Details from VMWare on different types of disaster recovery: `https://www.vmware.com/topics/glossary/content/disaster-recovery.html`
- A white paper from NIST detailing Information Security Continuous Monitoring, which is used a lot when talking about CSM: `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-137.pdf`
- Blog article from Upgard, a vendor of CSM tools, defining the major vulnerabilities: `https://www.upguard.com/blog/continuous-security-monitoring`
- *The Startup Way: How Modern Companies Use Entrepreneurial Management to Transform Culture and Drive Long-Term Growth* by Eric Ries – The follow-up book to *The Lean Startup*; this further defines concepts such as innovation accounting
- A blog article outlining additional thoughts on innovation accounting: `https://www.boldare.com/blog/lean-startup-innovation-accounting/`

14

Avoiding Pitfalls and Diving into the Future

The DevOps movement has evolved and changed in many ways since the Flickr talk at the 2009 Velocity Conference. New advances in technology, such as containers and Kubernetes, have changed many of the core practices. New ways of doing things such as **site reliability engineering** (**SRE**) have changed the roles people play.

In this chapter, we will examine how to start incorporating the principles and practices discussed throughout the book so that you can successfully move toward a DevOps transformation or move to SAFe®.

We will look at some emerging trends that may play a major role in the evolution of DevOps. We will look at XOps, which seeks to incorporate more parts of the organizations into the collaborative approach seen in DevOps and DevSecOps. We will also have a look at an alternative view to the standard linear value stream.

We will also examine new technologies that have emerged that promise to change how we work. With AI for IT Operations, also called AIOps, we will examine the growing role **Artificial Intelligence** (**AI**) and **Machine Learning** (**ML**) play in predicting problems and vulnerabilities. We will look at the merge between version control and **Continuous Integration/Continuous Deployment** (**CI/CD**) when we examine GitOps.

Finally, we will look at common questions that may come up with some answers.

Avoiding pitfalls

In the journey toward a DevOps approach, it only makes sense that not everyone will follow the same path. Different organizations are at different stages of readiness to accept the mindset change needed to allow for a successful DevOps journey.

That being said, we will look at a set of steps for starting or restarting a DevOps approach based on what we have learned in the previous chapters of this book. You will see that there is frequently no direct path from one step to the next.

Let's view the following steps for our DevOps journey:

- Understand the system
- Start small
- Start automating
- Measure, learn, and pivot

Lets start on our journey now.

Understand the system

Whether we are starting fresh or restarting after stumbling, we need to take a step from the Improvement Kata from Lean thinking and evaluate where we are. This evaluation should be a holistic look at the existing processes, the people behind the processes, and if tools are enabling processes.

One way of looking at all these facets is to do a value stream mapping, as first discussed in *Chapter 7*, *Mapping Your Value Streams*. As we originally spoke about in that chapter, performing a value stream mapping workshop allows you to discover the following things:

- The process steps in your workflow from beginning to end
- The people that perform those process steps
- Possible areas for improvement

The value stream mapping workshop also gives the perspective of looking at the scope of the entire effort of development in a broader perspective of a system. This systemic view is not just a series of process steps and improvements, but also how tools work in this process, or can work in the process if they're not incorporated.

A value stream mapping workshop produces two value streams: the current value stream and an *ideal* value stream that contains enhancements and improvements to lower cycle time and lead time and increases **Percent Complete and Accurate** (**%C&A**). Record these enhancements so that the people in the value stream will work on them as part of **continuous learning** (**CL**).

Start small

After the value stream mapping workshop, look at the list of improvements and enhancements from the *ideal* value stream. From the list of enhancements, the people in the value stream should select what they consider the most important enhancement to work on.

Allowing only one enhancement at a time promotes focus. We've seen this in *Chapter 4*, *Leveraging Lean Flow to Keep the Work Moving*, where working in small batch sizes avoids the problems that multitasking can bring in preventing work from completing.

One such enhancement may be to decrease cycle and lead time in developing your products by increasing flow. The practices in *Chapter 4*—such as small batch sizes, monitoring queues, and limiting WIP—can also help with this step. Following these practices may lead to increased agility that gets better with a DevOps approach.

Another area of improvement the people of the value stream can look at is the use of their tooling. The first foundational tool to consider is version control. The teams in the value stream should evaluate putting all their files in a version control system. If they are already using version control, they should ensure that they are using a common version control system and that everyone has the necessary access.

In addition to selecting a version control system, the people of the value stream should select a branching model that outlines how they will branch from the trunk and when to merge changes back to the main branch or trunk.

Start automating

A key step toward improving the value stream is establishing a CI/CD pipeline. If the organization has not established a CI/CD pipeline, it may establish one that comes included with many version control systems. Other organizations may want to have the flexibility that comes from installing a separate CI/CD pipeline tool. Either approach works, as the most important thing is the creation of such a pipeline.

The CI/CD pipeline can then be triggered from version control when a commit occurs. Add the necessary actions that occur after the commit, one action at a time. These actions may start with building or compiling the changes that come from the commit. A completed build can then allow the merge into the higher-level branch and possibly save the build artifacts into an artifact repository.

Start automating the testing

Once the endpoints of the CI/CD pipeline are defined, you can fill other intermediate steps of the pipeline with automated testing. You can create tests that will automatically run and find problems with functionality and security.

Start by creating, one by one, automated tests that comprise the following categories:

- Unit testing
- Functional testing
- System testing
- Security testing (**static application security testing** (**SAST**), **dynamic application security testing** (**DAST**), and other scans)

Development of the automated testing now encourages the creation of multiple environments for deployment, such as test, staging, and production. Unit test passage can allow for the deployment of a test environment for the other levels of testing.

Create environments, deployments, and monitoring

The establishment of testing allows us to establish the different pre-production environments, all of them equivalent as far as practicality will allow.

Introducing a new pre-production environment also allows us to introduce monitoring for these pre-production environments. Monitoring can start on the test environment until we have high enough confidence to deploy the monitoring on staging environments, and ultimately on production itself.

Constructing the pre-production environments allows us to establish the tooling to do the construction. Configuration management tooling and **Infrastructure-as-Code** (**IaC**) can be added to define and configure the environments. Establishing standard configurations allow capabilities such as rollback to occur.

Measure, learn, and pivot

At every step, we should evaluate whether each additional step we add is working to bring us closer to the ideal future state value stream. Make adjustments as needed. These tiny course corrections allow for CL.

There may be questions about whether the timing is right to pivot. The justification is always to avoid the *sunk cost* fallacy, where people hold on, avoiding change even in the face of overwhelming evidence that change should happen.

One final note: an apocryphal story holds that John F. Kennedy noted that the Chinese term for *crisis* was made up of the Chinese words for *danger* and *opportunity*. While it is true that the first part of the Chinese term for *crisis* on its own equals *danger*, the second part of the term on its own translates to *point of change* or *turning point*.

Adopting SAFe or adopting DevOps is a lot like watching the characters that make up a *crisis*. Watching the *danger* and watching the *turning points* will keep us away from the *crisis*.

Throughout this book, we've talked about the evolution of DevOps from its beginnings in 2009 to the present day. Let's now talk about trends that are starting to emerge.

Emerging trends in DevOps

One of the themes that contributed to the success of the DevOps movement was the collaboration between two separate functions of an organization, Development and Operations, to achieve the greater goal of releasing products frequently but also allowing stability in the production environment.

As the movement grew, its success encouraged more imaginative efforts at collaboration to encourage other aspects besides increased deployment frequency and stability. Some of the reimagined efforts we will examine include the following:

- XOps
- The Revolution model
- Platform engineering

Let's begin our exploration into the potential future of DevOps.

XOps

The success of the DevOps movement has encouraged the inclusion of other parts of the organization with Development and Operations to allow for speed of other qualities in addition to development frequency and stability. Notable movements include the following:

- **DevSecOps**: The incorporation of security throughout the stages of the pipeline to ensure that security is not left behind. Currently, many people consider the practices that are performed in a DevSecOps approach are fully absorbed in the DevOps movement.
- **BizDevOps**: Hailed as DevOps 2.0 this movement adds collaboration between business teams, developers, and operations personnel for the added goal of maximizing business revenue.

In addition, other engineering disciplines have found success in employing the same practices first promoted in DevOps. These DevOps-based movements for adjacent disciplines include **DataOps** and **ModelOps**.

Let's look at how a DevOps approach works in these engineering disciplines.

DataOps

DataOps is the incorporation of Agile development principles, DevOps practices, and Lean thinking into the field of data analysis and data analytics.

Data analysis follows a process flow similar to product development. The following steps are part of a typical process for data analysis:

1. Specification of the requirements for the data by those directing the analysis or by customers
2. Collection of the data
3. Processing data using statistical software or spreadsheets
4. Data cleaning to remove duplicates or errors
5. **Exploratory data analysis** (**EDA**)
6. Data modeling to find relationships

7. Creating outputs by using a data product application
8. Reporting

As the volume of data increases exponentially, new methods are called into service to allow insights to be quickly obtained. The following methods used resemble the steps seen in DevOps for product development:

- The establishment of a data pipeline that allows for the processing of data into reports and analytics using the data analysis process
- Verification of the data flowing through the data pipeline through **Statistical Process Controls** (**SPC**)
- Adding automated testing to ensure the data is correct in the data cleaning, data analysis, and data modeling stages
- Measurement of the data flow through the data pipeline
- Separation of data into development, test, and production environments to ensure the correct function of automation for the data pipeline
- Collaboration between data scientists, analysts, data engineers, IT, and quality assurance/governance

The ability to easily find relationships based on voluminous amounts of data provides a competitive advantage. Using DataOps practices and principles may allow those creating insights from data to achieve greater deployment rates of reliable and error-free data analytics.

ModelOps

One of the largest trends in technology today is AI. This growing trend sets up ML and decision models that leverage large amounts of data to progressively improve performance. The development of these ML models can give a competitive advantage to any organization looking to understand more about their customers and how their products can help.

Developing effective models now goes through a development cycle, similar to the **software development life cycle** (**SDLC**) process. In December 2018, Waldemar Hummer and Vinod Muthusamy of IBM Research AI proposed the initial concept of ModelOps, where AI workflows were made more reusable, platform-independent, and composable using techniques derived from the DevOps approach.

The model life cycle covers how to create, train, deploy, monitor, and, based on feedback from the data, retrain the AI model for an enterprise. The following diagram demonstrates the path of the model life cycle:

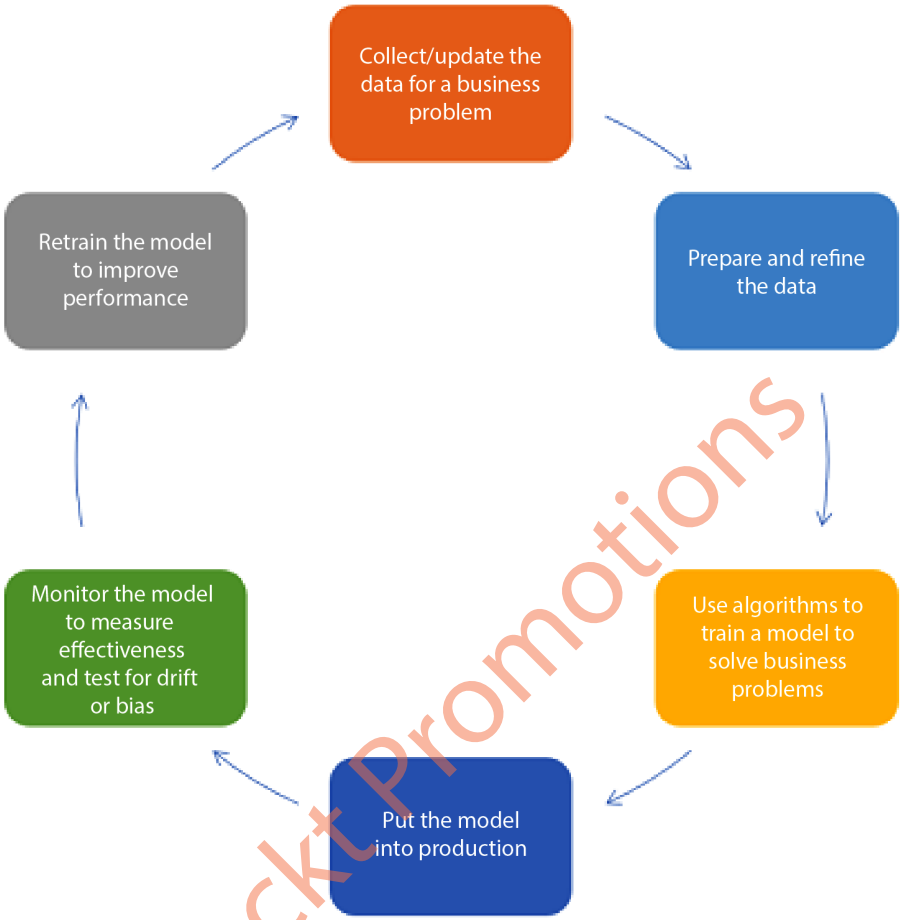


Figure 14.1 – ModelOps life cycle (By Jforgo1 – Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=99598780)

Because of the similarities between the model life cycle and the SDLC, the desired behaviors of deploying changes rapidly, monitoring their effects on the environment, and learning to improve allow for AI models developed using ModelOps to align with AI applications created using DevOps and have both rely on data analytics developed using DataOps.

The Revolution model

The concept of value streams for software development takes as a model the traditional SDLC process. This process, a remnant from Waterfall development, assumes that development happens in a step-by-step linear motion from one activity to the next. As products become more complex and DevOps expands the areas of responsibility, does a linear one-way progression of activities make sense?

Emily Freeman, Head of Community Engagement at **Amazon Web Services** (**AWS**) and the author of *DevOps for Dummies*, has proposed a different way of looking at the development process. Instead of a straight-line approach, she outlines a model where priorities move forward and back, along circles of activity. She calls this model **Revolution**. Let's take a closer look.

Freeman outlines the following five software development roles as concentric circles radiating out:

- Operating
- Deploying
- Automating
- Developing
- Architecting

Intersecting those circles are six qualities that engineers must consider in each activity. These qualities, listed here, are drawn as spokes against the concentric circles:

- Testability
- Securability
- Reliability
- Observability
- Flexibility
- Scalability

With the parts in place, let's present the following diagram of the model:

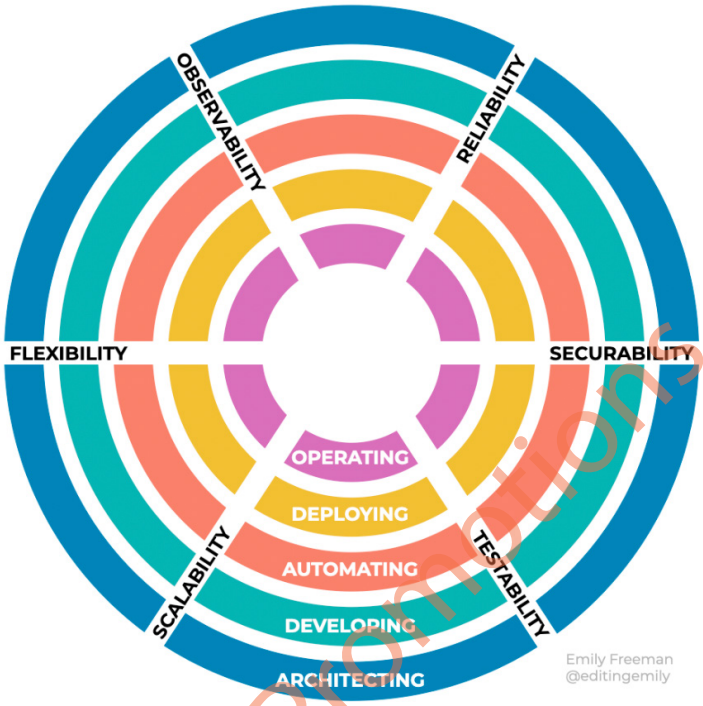


Figure 14.2 – Revolution model

To ensure the six qualities are adequately met, an engineer or team member would jump from one role (or circle) to the next, starting from the outside in. Roles are meant to be temporarily adopted by the engineer for some period of time until they are not needed at the moment. In an incident management scenario that Freeman illustrates in a keynote (`https://www.youtube.com/watch?v=rNBXfgWcy5Q`), the team has members assume roles in operating, developing, and deploying to investigate whether adequate reliability, observability, flexibility, scalability, and testability occur.

Platform engineering

The vast amount of tooling and technology involved in cloud-native environments has accelerated in the past few years. The level of knowledge needed to set up and maintain test, staging, and production environments on which products reside has overwhelmed those responsible for their operation.

Historically, DevOps started with the inclusion of Operations in an Agile development process so that stability and development speed are equally maintained. Over time, Operations has embraced using an Agile development mindset to solve reliability problems of large systems with SRE. Charity Majors, the CTO of honeycomb.io, a leading company in the development of a monitoring and observability platform, writes in a blog article (`https://www.honeycomb.io/blog/future-ops-platform-engineering`) that the next step is to look at the environments that are the responsibility of Operations as products themselves that can be developed using an Agile development mindset.

Platform engineering, as defined by Luca Galante of Humanitec (`https://platformengineering.org/blog/what-is-platform-engineering`), designs toolchains and workflows to allow self-service capabilities using this Agile development mindset. The product developed through platform engineering is an **Internal Developer Platform** (**IDP**), an abstraction of the technology used in operational environments for developers to build their products and solutions.

Developers will use the following five components of an IDP to establish the capabilities required by their product or solution:

- **Application Configuration Management (ACM)**: This includes the automatic creation of manifest files used by configuration management tools to deploy application changes
- **Infrastructure Orchestration**: This establishes integration between the CI pipeline with the environments for deployment, including possible cluster creation/updates, IAC, and image registries
- **Environment Management**: This integrates the ACM of the IDP with the underlying infrastructure of the environments to allow developers to create fully provisioned environments when needed
- **Deployment Management**: This sets up the CD pipeline
- **Role-Based Access Control (RBAC)**: This allows granular access to the environment and its resources

We have seen trends emerge as people look at applying changes to processes, but DevOps will continue to change as technology changes. Let's look at trends in DevOps that are driven by advances in technology.

New technologies in DevOps

DevOps today has expanded beyond what was originally envisioned in 2009, in part due to advances in technology that encouraged deployment into cloud-native environments. Advances today promise to add new capabilities never imagined.

The technologies we will examine for changing DevOps come from both refinements in deploying to cloud-native environments and applications on exciting new technology. We will take a look at these technology-based DevOps trends: **AIOps** and **GitOps**.

Let's start out by looking to the future with AIOps.

AIOps

We discussed in the previous section the rise of the use of AI to create applications that can learn from the large amounts of data presented to AI models. One area that can benefit from improved insights based on large amounts of data may be product development that uses a DevOps approach and collects the data through full-stack telemetry. This is different from our previous discussions of DataOps or ModelOps because this time, we are applying ML and data visualization to our actual development process.

AI for IT Operations (or AIOps) looks to enhance traditional practices we previously identified in the CI, CD, and Release on Demand stages of the CD pipeline to incorporate tools that are based on ML to process the amount of data that comes from our full-stack telemetry. Incorporation of AIOps tools may assist in making sense of the following areas of the IT environment:

- **Systems**: Testing, staging, and production environments may be a complex mix of cloud-native and on-premise resources. Cloud resources may come from one or more vendors. The resources may be a mix of physical computing servers ("bare-metal") or use a form of virtualization such as **virtual machines** (**VMs**) or containers. This large mix of resources ensures the environment is stable and reliable.
- **Data**: Full-stack telemetry creates mountains of data. Some of this data may be key to making crucial decisions, while some may not. How can we determine when the amount of data is voluminous?
- **Tools**: To acquire full-stack telemetry, a wide variety of tools may be employed to collect the data and manage the systems. These tools may not operate with one another or may be limited in functionality, creating silos of data.

To answer this challenge, tools incorporating ML may adopt the following types of AI algorithms:

- **Data selection**: This analyzes data to remove redundant and non-relevant information
- **Pattern discovery**: This examines the data to determine relationships
- **Inference**: This looks at the insights to identify root causes
- **Collaboration**: This enables automation of notifications to teams of problems discovered
- **Automation**: This looks to automate a response to recurring problems

These tools have applications in terms of identifying security vulnerabilities, finding root causes of production failures, or even identifying trends that may predict impending problems in your production environments. As the technology improves, the promise of the viability of AIOps to be a more standard part of DevOps and IT operations grows.

GitOps

As acceptance of CI/CD pipelines grew, many wondered how to establish these pipelines to deploy features into cloud-native testing, staging, and production environments. In 2017, Weaveworks coined the term *GitOps*, suggesting a CI/CD approach that is triggered from a commit in Git, a popular version control program.

GitOps starts with version control. There are separate repositories for the application and the environment configuration. The application repository will contain the code for the product, including how to build the product as a container in a Dockerfile. The environment repository will contain configuration files and scripts for a CI/CD pipeline tool, as well as deployment records for the environments.

Deployments in GitOps can either be push-based or pull-based. A push-based deployment uses the customary CI/CD pipeline tools to move from CI to CD. Pull-based deployment employs an operator that observes changes to the environment repository. When those changes occur, the operator deploys the changes to the environment according to the environment repository changes.

The following diagram shows the progression from the Git repository to the CI pipeline to the operator and deployment pipeline:

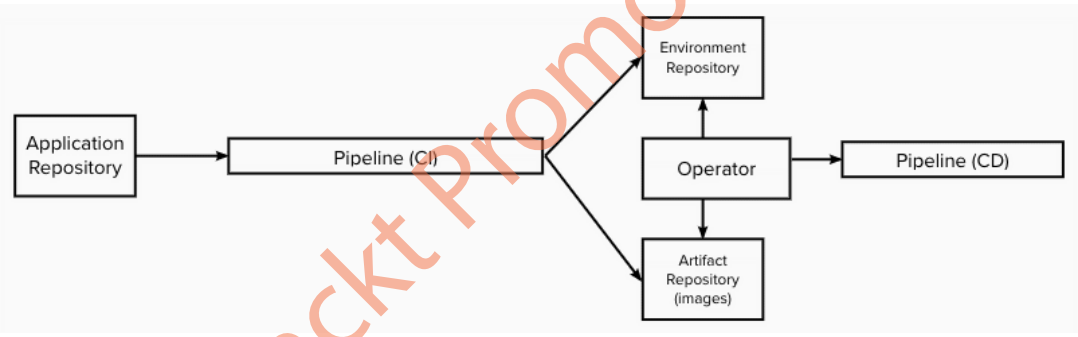


Figure 14.3 – GitOps pull-based deployment

GitOps has had a tremendous following with those looking to set up deployment using Kubernetes, a technology that packages Docker containers to set up clusters of microservices. As Kubernetes becomes more popular, GitOps has become an established practice to continuously deploy Kubernetes clusters to cloud-based environments.

Frequently asked questions

We now take a look at some questions that may come up, even after reading the earlier parts of this book.

What is DevOps?

DevOps is a technical movement for developing and maintaining products. It incorporates Lean thinking and Agile development principles and practices to extend focus beyond development to the deployment, release, and maintenance of new products and their features. DevOps promotes the use of automation and tooling to allow frequent testing for proper functionality and security and allow for consistent deployments and releases. As products move to release, DevOps models operational practices done by developers and operations personnel that allow for rapid recovery should problems appear in production environments in a mission-based generative culture.

What is the Scaled Agile Framework®?

The Scaled Agile Framework, or SAFe, is a set of values, principles, and practices developed by Scaled Agile, Inc. SAFe is primarily adopted by medium and large-size organizations working to introduce Lean and Agile ways of working for teams of teams and portfolios in enterprises. As Scaled Agile, Inc. defines it:

> *SAFe for Lean Enterprises is the world's leading framework for business agility. SAFe integrates the power of Lean, Agile, and DevOps into a comprehensive operating system that helps enterprises thrive in the digital age by delivering innovative products and services faster, more predictably, and with higher quality.*
> *— © Scaled Agile, Inc.*

Do I need to adopt SAFe to move to DevOps?

No. We look at SAFe while talking about DevOps because a key perspective of working in SAFe is the focus on creating and organizing value streams for its team of teams, known as an **Agile Release Train** (**ART**). SAFe also focuses on process and automation through its CD pipeline. Both of these are in line with successful DevOps approaches.

Adopting the SAFe is best suited when your value stream can be accomplished by a team of 5-12 regular-sized teams and this *team of teams* is working together on a single product or solution. This is known as the *Essential SAFe* configuration.

Is DevOps only for those companies that develop for cloud environments?

While many of the recent technical strides in DevOps were developed to help those develop, test, deploy, release, and maintain on cloud environments, DevOps as a technical movement is agnostic to whichever technology your product is based on.

DevOps principles and practices have been successful in organizations that develop and maintain products on cloud, embedded hardware, mainframe, and physical server environments. Practices such as the adoption of Lean thinking, value stream management, and SRE do not rely on the technology used by the end product.

What's the best tool for doing _____?

A common question is to ask what the most preferred tool is to perform a specific function such as CI/CD, automated testing, configuration management, or security testing. I always refrain from picking an individual tool for several reasons.

The primary reason is that every industry and organization is different. A tool that works for one industry or organization may not work for a different industry or organization.

Another reason is that technology moves onward, releasing new tools that may now be considered "best."

How does DevSecOps fit into DevOps?

Earlier in this chapter, we saw how DevSecOps was one of the first models of XOps where what eventually happened was the incorporation of a security mindset and practices into the broader definition of DevOps. With DevSecOps, we are adding security as a focus in addition to the desired foci of development speed and operational stability in *vanilla* DevOps. In this book, we outlined several places where including security practices move us from DevOps to DevSecOps.

In *Chapter 6*, *Recovering from Production Failures*, we saw that Chaos Engineering could be used to simulate disasters such as a security vulnerability and evaluate potential responses to such security failures.

In *Chapter 10*, *Continuous Exploration and Finding New Features*, we discovered that security was one of the primary considerations when trying to create new features for products. This consideration is handled by the system architect, often consulting with the organization's security team.

We extended security further in *Chapter 11*, *Continuous Integration of Solution Development*, and *Chapter 12*, *Continuous Deployment to Production*, where we looked at adding security testing in various forms to detect potential vulnerabilities. During CI, we performed threat modeling to identify any potential attack vectors.

Finally, in *Chapter 13*, *Releasing on Demand to Realize Value*, we looked at continuous security monitoring and continuous security practices in the production environment to remain vigilant and fix environments when vulnerabilities do get exploited.

Summary

In this final chapter, we looked to the future. We first saw the emerging trends happening to DevOps from the product development side. New advances incorporating AI and data engineering have added new components to products under development. Development of artifacts such as AI models and data pipelines shows great benefits using a DevOps approach in new fields such as DataOps and ModelOps. We also examined a new look at software development and maintenance that may influence DevOps and value stream management.

Changes in technology have changed DevOps as well. We looked at incorporating AIOps products that include AI for analyzing the data collected in testing and maintenance to find vulnerabilities and potential failures. We explored the GitOps movement that unites version control with CD.

We then moved from the future of DevOps to making DevOps part of your organization's future. We looked at adopting DevOps practices, including mapping your value stream and iteratively employing automation in different areas of your development process. We then finished with some questions you may have and their answers.

We have explored the different aspects of DevOps, from looking at the motivations of people to understanding the processes at play and examining the tools that can help achieve higher levels of performance. All three aspects are adopted in the SAFe so that teams of teams can perform together as a value stream to develop, deploy, and maintain products that provide value to the customer. It is our hope that this exploration helps guide your thoughts as you continue your DevOps journey.

Further reading

For more information, refer to the following resources:

- Blog article talking about XOps and its various components: `https://www.expressanalytics.com/blog/everything-you-need-to-know-about-xops/`
- Description of XOps, its main components, and its popularity: `https://datakitchen.io/gartner-top-trends-in-data-and-analytics-for-2021-xops/`
- Guidance article from Scaled Agile, Inc. that details how to work with DataOps in your value stream: `https://www.scaledagileframework.com/an-agile-approach-to-big-data-in-safe/`

- An initial whitepaper from Waldemar Hummer and Vinod Muthusamy of IBM Research AI detailing their ModelOps process: https://s3.us.cloud-object-storage.appdomain.cloud/res-files/3842-plday18-hummer.pdf
- An essential guide to ModelOps, created by ModelOp.com: https://www.modelop.com/wp-content/uploads/2020/05/ModelOps_Essential_Guide.pdf
- Guidance article from Scaled Agile, Inc. on working with ModelOps on your value stream: https://www.scaledagileframework.com/succeeding-with-ai-in-safe/
- The official documentation for Emily Freeman's Revolution model: https://github.com/revolution-model
- Keynote address that features Emily Freeman discussing the Revolution model: https://www.youtube.com/watch?v=rNBXfgWcy5Q
- Blog article from Charity Majors, CTO of honeycomb.io, introducing platform engineering: https://www.honeycomb.io/blog/future-ops-platform-engineering
- Article from Luca Galante of Humanitec, describing platform engineering: https://platformengineering.org/blog/what-is-platform-engineering
- Definition of an IDP and its components: https://internaldeveloperplatform.org
- Blog article from Moogsoft, a DevOps tool vendor, detailing the uses of AIOps: https://www.moogsoft.com/resources/aiops/guide/everything-aiops/
- A primer on GitOps: https://www.gitops.tech
- A blog article from Warren Marusiak of Atlassian, outlining an approach to adopting DevOps: https://community.atlassian.com/t5/DevOps-articles/How-to-do-DevOps/ba-p/2137695
- Article from Scaled Agile, Inc. defining the Scaled Agile Framework: https://www.scaledagileframework.com/safe-for-lean-enterprises/

Assessment Answers

This section contains answers to questions from all chapters.

Chapter 1 – Introducing SAFe® and DevOps

1. C
2. C, D
3. B
4. B
5. C

Chapter 2 – Culture of Shared Responsibility

1. D
2. B
3. C
4. A, E
5. C
6. C

Chapter 3 – Automation for Efficiency and Quality

1. B, D
2. C
3. C

Chapter 4 – Leveraging Lean Flow to Keep the Work Moving

1. A, D
2. B

3. A
4. C
5. B, E
6. B
7. C

Chapter 5 – Measuring the Process and Solution

1. C
2. A
3. B
4. B, D
5. A

Chapter 6 – Recovering from Production Failures

1. B
2. C
3. C
4. A
5. C, D
6. B

Chapter 7 – Mapping your Value Streams

1. C
2. B
3. C
4. D
5. D
6. D
7. C

Chapter 8 – Measuring Value Stream Performance

1. B
2. D
3. A
4. D, E
5. A
6. C

Chapter 9 – Moving to the Future with Continuous Learning

1. B
2. B, D
3. A
4. C

Chapter 10 – Continuous Exploration and Finding New Features

1. C
2. B, C, E
3. B
4. D
5. A, C, E

Chapter 11 – Continuous Integration of Solution Development

1. B, D
2. A
3. C
4. A
5. E

6. B, C
7. D

Chapter 12 – Continuous Deployment to Production

1. B, C, E
2. B
3. A, D, E
4. B, D, E
5. E
6. B
7. C
8. B, C

Chapter 13 – Releasing on Demand to Realize Value

1. B, D, E
2. A, B, E
3. A
4. A, C
5. B, E

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

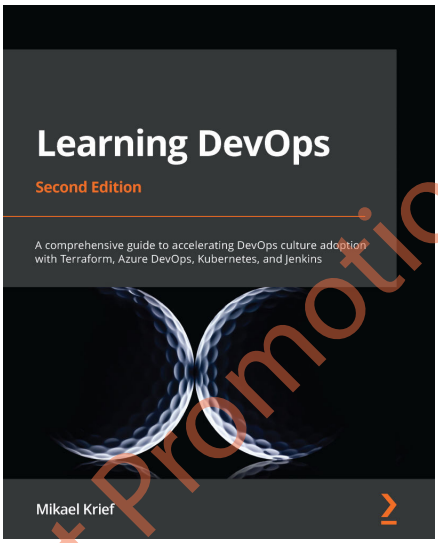
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

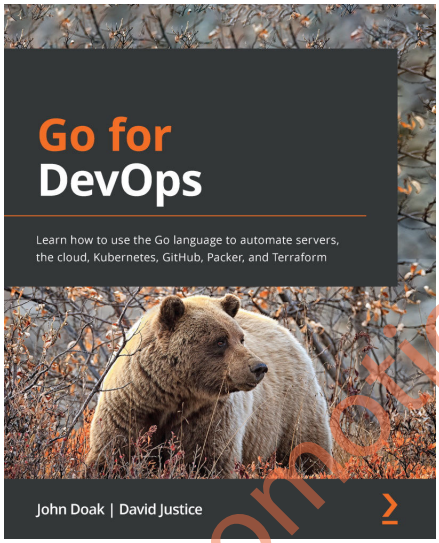


Learning DevOps - Second Edition

Mikael Krief

ISBN: 9781801818964

- Understand the basics of infrastructure as code patterns and practices
- Get an overview of Git command and Git flow
- Install and write Packer, Terraform, and Ansible code for provisioning and configuring cloud infrastructure based on Azure examples
- Use Vagrant to create a local development environment
- Containerize applications with Docker and Kubernetes
- Apply DevSecOps for testing compliance and securing DevOps infrastructure
- Build DevOps CI/CD pipelines with Jenkins, Azure Pipelines, and GitLab CI
- Explore blue/green deployment and DevOps practices for open sources projects



Go for DevOps

John Doak, David Justice

ISBN: 9781801818896

- Understand the basic structure of the Go language to begin your DevOps journey
- Interact with filesystems to read or stream data
- Communicate with remote services via REST and gRPC
- Explore writing tools that can be used in the DevOps environment
- Develop command-line operational software in Go
- Work with popular frameworks to deploy production software
- Create GitHub actions that streamline your CI/CD process
- Write a ChatOps application with Slack to simplify production visibility

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *SAFe® for DevOps Practitioners*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781803231426`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly