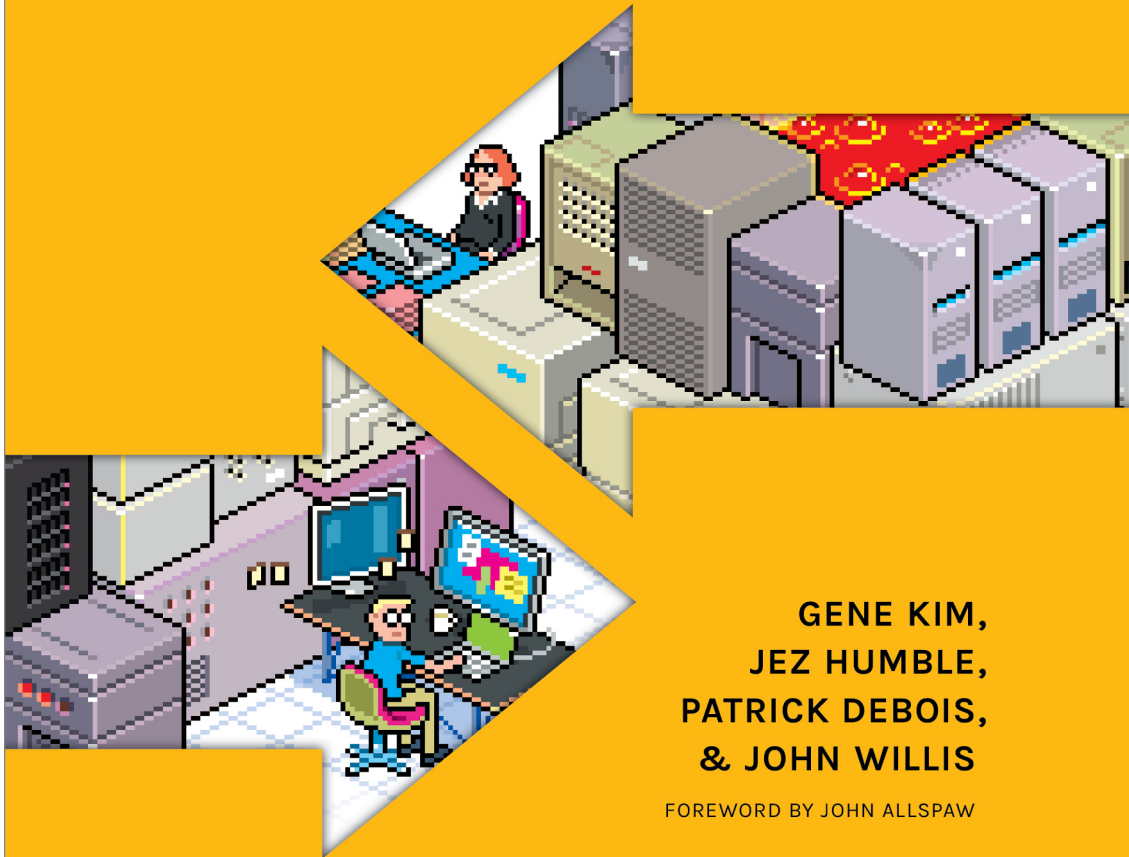


The
**DevOps
Handbook**

**HOW TO CREATE WORLD-CLASS
AGILITY, RELIABILITY, & SECURITY
IN TECHNOLOGY ORGANIZATIONS**



**GENE KIM,
JEZ HUMBLE,
PATRICK DEBOIS,
& JOHN WILLIS**

FOREWORD BY JOHN ALLSPAW

THE DEVOPS HANDBOOK

How to Create World-Class
Agility, Reliability, & Security in
Technology Organizations

By Gene Kim, Jez Humble, Patrick Debois, and John Willis





IT Revolution Press, LLC
25 NW 23rd Pl, Suite 6314
Portland, OR 97210

Copyright © 2016 by Gene Kim, Jez Humble, Patrick Debois, and John Willis

All rights reserved, for information about permission to reproduce selections from this book,
write to Permissions, IT Revolution Press, LLC, 25 NW 23rd Pl, Suite 6314, Portland, OR 97210

First Edition

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Cover design by Strauber Design Studio

Cover illustration by eboy

Book design by Mammoth Collective

ISBN: 978-1942788003

Publisher's note to readers: Many of the ideas, quotations, and paraphrases attributed to different thinkers and industry leaders herein are excerpted from informal conversations, correspondence, interviews, conference roundtables, and other forms of oral communication that took place over the last six years during the development and writing of this book. Although the authors and publisher have made every effort to ensure that the information in this book was correct at press time, the authors and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause.

The author of the 18F case study on page 325 has dedicated the work to the public domain by waiving all of his or her rights to the work worldwide under copyright law, including all related and neighboring rights, to the extent allowed by law. You can copy, modify, distribute, and perform case study 18F, even for commercial purposes, all without asking permission.

For information about special discounts for bulk purchases
or for information on booking authors for an event,
please visit ITRevolution.com.

THE DEVOPS HANDBOOK

Excerpt - Not for reproduction or use without permission

TABLE OF CONTENTS

Preface	xi
Foreword	xix
Imagine a World Where Dev and Ops Become DevOps: <i>An Introduction to The DevOps Handbook</i>	xxi
PART I—THE THREE WAYS	1
Part I Introduction	3
1 Agile, Continuous Delivery, and the Three Ways	7
2 The First Way: <i>The Principles of Flow</i>	15
3 The Second Way: <i>The Principles of Feedback</i>	27
4 The Third Way: <i>The Principles of Continual Learning and Experimentation</i>	37
PART II—WHERE TO START	47
Part II Introduction	49
5 Selecting Which Value Stream to Start With	51
6 Understanding the Work in Our Value Stream, Making it Visible, and Expanding it Across the Organization	61
7 How to Design Our Organization and Architecture with Conway's Law in Mind	77
8 How to Get Great Outcomes by Integrating Operations into the Daily Work of Development	95
PART III—THE FIRST WAY: THE TECHNICAL PRACTICES OF FLOW	107
Part III Introduction	109
9 Create the Foundations of Our Deployment Pipeline	111
10 Enable Fast and Reliable Automated Testing	123
11 Enable and Practice Continuous Integration	143
12 Automate and Enable Low-Risk Releases	153
13 Architect for Low-Risk Releases	179

PART IV—THE SECOND WAY:

THE TECHNICAL PRACTICES OF FEEDBACK

Part IV Introduction	191
14 Create Telemetry to Enable Seeing and Solving Problems	193
15 Analyze Telemetry to Better Anticipate Problems and Achieve Goals	195
16 Enable Feedback So Development and Operations Can Safely Deploy Code	215
17 Integrate Hypothesis-Driven Development and A/B Testing into Our Daily Work	227
18 Create Review and Coordination Processes to Increase Quality of Our Current Work	241
	249

PART V—THE THIRD WAY:

THE TECHNICAL PRACTICES OF CONTINUAL LEARNING AND EXPERIMENTATION

Part V Introduction	267
19 Enable and Inject Learning into Daily Work	269
20 Convert Local Discoveries into Global Improvements	271
21 Reserve Time to Create Organizational Learning and Improvement	287
	299

PART VI—THE TECHNOLOGICAL PRACTICES OF INTEGRATING INFORMATION SECURITY, CHANGE MANAGEMENT, AND COMPLIANCE

Part VI Introduction	309
22 Information Security as Everyone's Job, Every Day	311
23 Protecting the Deployment Pipeline, and Integrating into Change Management and Other Security and Compliance Controls	313
Conclusion to the DevOps Handbook: A Call to Action	333
	347

ADDITIONAL MATERIAL

Appendices	351
Additional Resources	353
Endnotes	366
Index	370
Acknowledgments	409
Author Biographies	435
	439

THE DEVOPS HANDBOOK

Excerpt - Not for reproduction or use without permission



Preface

Aha!

The journey to complete *The DevOps Handbook* has been a long one—it started with weekly working Skype calls between the co-authors in February of 2011, with the vision of creating a prescriptive guide that would serve as a companion to the as-yet unfinished book *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*.

More than five years later, with over two thousand hours of work, *The DevOps Handbook* is finally here. Completing this book has been an extremely long process, although one that has been highly rewarding and full of incredible learning, with a scope that is much broader than we originally envisioned. Throughout the project, all the co-authors shared a belief that DevOps is genuinely important, formed in a personal “aha” moment much earlier in each of our professional careers, which I suspect many of our readers will resonate with.

Gene Kim

I’ve had the privilege of studying high-performing technology organizations since 1999, and one of the earliest findings was that boundary-spanning between the different functional groups of IT Operations, Information Security, and Development was critical to success. But I still remember the first time I saw the magnitude of the downward spiral that would result when these functions worked toward opposing goals.

It was 2006, and I had the opportunity to spend a week with the group who managed the outsourced IT Operations of a large airline reservation service. They described the downstream consequences of their large, annual software releases: each release would cause immense chaos and disruption for the outsourcer, as well as customers; there would be SLA (service level agreement) penalties, because of the customer-impacting outages; there would be layoffs of the most

talented and experienced staff, because of the resulting profit shortfalls; there would be much unplanned work and firefighting so that the remaining staff couldn't work on the ever-growing service request backlogs coming from customers; the contract would be held together by the heroics of middle management; and everyone felt that the contract would be doomed to be put out for re-bid in three years.

The sense of hopelessness and futility that resulted created for me the beginnings of a moral crusade. Development seemed to always be viewed as strategic, but IT Operations was viewed as tactical, often delegated away or outsourced entirely, only to return in five years in worse shape than it was first handed over.

For many years, many of us knew that there must be a better way. I remember seeing the talks coming out of the 2009 Velocity Conference, describing amazing outcomes enabled by architecture, technical practices, and cultural norms that we now know as DevOps. I was so excited, because it clearly pointed to the better way that we had all been searching for. And helping spread that word was one of my personal motivations to co-author *The Phoenix Project*. You can imagine how incredibly rewarding it was to see the broader community react to that book, describing how it helped them achieve their own “aha” moments.

Jez Humble

My DevOps “aha” moment was at a start-up in 2000—my first job after graduating. For some time, I was one of two technical staff. I did everything: networking, programming, support, systems administration. We deployed software to production by FTP directly from our workstations.

Then in 2004 I got a job at ThoughtWorks, a consultancy where my first gig was working on a project involving about seventy people. I was on a team of eight engineers whose full-time job was to deploy our software into a production-like environment. In the beginning, it was really stressful. But over a few months we went from manual deployments that took two weeks to an automated deployment that took one hour, where we could roll forward and back in milliseconds using the blue-green deployment pattern during normal business hours.

That project inspired a lot of the ideas in both the *Continuous Delivery* (Addison-Wesley, 2000) book and this one. A lot of what drives me

and others working in this space is the knowledge that, whatever your constraints, we can always do better, and the desire to help people on their journey.

Patrick Debois

For me, it was a collection of moments. In 2007 I was working on a data center migration project with some Agile teams. I was jealous that they had such high productivity—able to get so much done in so little time.

For my next assignment, I started experimenting with Kanban in Operations and saw how the dynamic of the team changed. Later, at the Agile Toronto 2008 conference I presented my IEEE paper on this, but I felt it didn't resonate widely in the Agile community. We started an Agile system administration group, but I overlooked the human side of things.

After seeing the 2009 Velocity Conference presentation “10 Deploys per Day” by John Allspaw and Paul Hammond, I was convinced others were thinking in a similar way. So I decided to organize the first DevOpsDays, accidentally coining the term DevOps.

The energy at the event was unique and contagious. When people started to thank me because it changed their life for the better, I understood the impact. I haven't stopped promoting DevOps since.

John Willis

In 2008, I had just sold a consulting business that focused on large-scale, legacy IT operations practices around configuration management and monitoring (Tivoli) when I first met Luke Kanies (the founder of Puppet Labs). Luke was giving a presentation on Puppet at an O'Reilly open source conference on configuration management (CM).

At first I was just hanging out at the back of the room killing time and thinking, “What could this twenty-year-old tell me about configuration management?” After all, I had literally been working my entire life at some of the largest enterprises in the world, helping them architect CM and other operations management solutions. However, about five minutes into his session, I moved up to the first row and realized everything I had been doing for the last twenty years was wrong. Luke was describing what I now call second generation CM.

After his session I had an opportunity to sit down and have coffee with him. I was totally sold on what we now call infrastructure as code. However, while we met for coffee, Luke started going even further, explaining his ideas. He started telling me he believed that operations was going to have to start behaving like software developers. They were going to have to keep their configurations in source control and adopt CI/CD delivery patterns for their workflow. Being the old IT Operations person at the time, I think I replied to him with something like, “That idea is going to sink like Led Zeppelin with Ops folk.” (I was clearly wrong.)

Then about a year later in 2009 at another O’Reilly conference, Velocity, I saw Andrew Clay Shafer give a presentation on Agile Infrastructure. In his presentation, Andrew showed this iconic picture of a wall between developers and operations with a metaphorical depiction of work being thrown over the wall. He coined this “the wall of confusion.” The ideas he expressed in that presentation codified what Luke was trying to tell me a year earlier. That was the light bulb for me. Later that year, I was the only American invited to the original DevOpsDays in Ghent. By the time that event was over, this thing we call DevOps was clearly in my blood.

Clearly, the co-authors of this book all came to a similar epiphany, even if they came there from very different directions. But there is now an overwhelming weight of evidence that the problems described above happen almost everywhere, and that the solutions associated with DevOps are nearly universally applicable.

The goal of writing this book is to describe how to replicate the DevOps transformations we’ve been a part of or have observed, as well as dispel many of the myths of why DevOps won’t work in certain situations. Below are some of the most common myths we hear about DevOps.

Myth—DevOps is Only for Startups: While DevOps practices have been pioneered by the web-scale, Internet “unicorn” companies such as Google, Amazon, Netflix, and Etsy, each of these organizations has, at some point in their history, risked going out of business because of the problems associated with more traditional “horse” organizations: highly dangerous code releases that were prone to catastrophic failure, inability to release features fast enough to beat the competition, compliance concerns, an inability to scale, high levels of distrust between Development and Operations, and so forth.

However, each of these organizations was able to transform their architecture, technical practices, and culture to create the amazing outcomes that we associate with DevOps. As Dr. Branden Williams, an information security executive, quipped, “Let there be no more talk of DevOps unicorns or horses but only thoroughbreds and horses heading to the glue factory.”

Myth—*DevOps Replaces Agile*: DevOps principles and practices are compatible with Agile, with many observing that DevOps is a logical continuation of the Agile journey that started in 2001. Agile often serves as an effective enabler of DevOps, because of its focus on small teams continually delivering high quality code to customers.

Many DevOps practices emerge if we continue to manage our work beyond the goal of “potentially shippable code” at the end of each iteration, extending it to having our code always in a deployable state, with developers checking into trunk daily, and that we demonstrate our features in production-like environments.

Myth—*DevOps is incompatible with ITIL*: Many view DevOps as a backlash to ITIL or ITSM (IT Service Management), which was originally published in 1989. ITIL has broadly influenced multiple generations of Ops practitioners, including one of the co-authors, and is an ever-evolving library of practices intended to codify the processes and practices that underpin world-class IT Operations, spanning service strategy, design, and support.

DevOps practices can be made compatible with ITIL process. However, to support the shorter lead times and higher deployment frequencies associated with DevOps, many areas of the ITIL processes become fully automated, solving many problems associated with the configuration and release management processes (e.g., keeping the configuration management database and definitive software libraries up to date). And because DevOps requires fast detection and recovery when service incidents occur, the ITIL disciplines of service design, incident, and problem management remain as relevant as ever.

Myth—*DevOps is Incompatible with Information Security and Compliance*: The absence of traditional controls (e.g., segregation of duty, change approval processes, manual security reviews at the end of the project) may dismay information security and compliance professionals.

However, that doesn’t mean that DevOps organizations don’t have effective controls. Instead of security and compliance activities only being performed

at the end of the project, controls are integrated into every stage of daily work in the software development life cycle, resulting in better quality, security, and compliance outcomes.

Myth—*DevOps Means Eliminating IT Operations, or “NoOps:”* Many misinterpret DevOps as the complete elimination of the IT Operations function. However, this is rarely the case. While the nature of IT Operations work may change, it remains as important as ever. IT Operations collaborates far earlier in the software life cycle with Development, who continues to work with IT Operations long after the code has been deployed into production.

Instead of IT Operations doing manual work that comes from work tickets, it enables developer productivity through APIs and self-serviced platforms that create environments, test and deploy code, monitor and display production telemetry, and so forth. By doing this, IT Operations become more like Development (as do QA and Infosec), engaged in product development, where the product is the platform that developers use to safely, quickly, and securely test, deploy, and run their IT services in production.

Myth—*DevOps is Just “Infrastructure as Code” or Automation:* While many of the DevOps patterns shown in this book require automation, DevOps also requires cultural norms and an architecture that allows for the shared goals to be achieved throughout the IT value stream. This goes far beyond just automation. As Christopher Little, a technology executive and one of the earliest chroniclers of DevOps, wrote, “DevOps isn’t about automation, just as astronomy isn’t about telescopes.”

Myth—*DevOps is Only for Open Source Software:* Although many DevOps success stories take place in organizations using software such as the LAMP stack (Linux, Apache, MySQL, PHP), achieving DevOps outcomes is independent of the technology being used. Successes have been achieved with applications written in Microsoft.NET, COBOL, and mainframe assembly code, as well as with SAP and even embedded systems (e.g., HP LaserJet firmware).

SPREADING THE AHA! MOMENT

Each of the authors has been inspired by the amazing innovations happening in the DevOps community and the outcomes they are creating: they are creating safe systems of work, and enabling small teams to quickly and independently develop and validate code that can be safely deployed to customers. Given our belief that DevOps is a manifestation of creating dynamic, learning organi-

zations that continually reinforce high-trust cultural norms, it is inevitable that these organizations will continue to innovate and win in the marketplace.

It is our sincere hope that *The DevOps Handbook* will serve as a valuable resource for many people in different ways: a guide for planning and executing DevOps transformations, a set of case studies to research and learn from, a chronicle of the history of DevOps, a means to create a coalition that spans Product Owners, Architecture, Development, QA, IT Operations, and Information Security to achieve common goals, a way to get the highest levels of leadership support for DevOps initiatives, as well as a moral imperative to change the way we manage technology organizations to enable better effectiveness and efficiency, as well as enabling a happier and more humane work environment, helping everyone become lifelong learners—this not only helps everyone achieve their highest goals as human beings, but also helps their organizations win.

Excerpt - Not for reproduction or use without permission

Foreword

In the past, many fields of engineering have experienced a sort of notable evolution, continually “leveling-up” its understanding of its own work. While there are university curriculums and professional support organizations situated within specific disciplines of engineering (civil, mechanical, electrical, nuclear, etc.), the fact is, modern society needs all forms of engineering to recognize the benefits of and work in a multidisciplinary way.

Think about the design of a high-performance vehicle. Where does the work of a mechanical engineer end and the work of an electrical engineer begin? Where (and how, and when) should someone with domain knowledge of aerodynamics (who certainly would have well-formed opinions on the shape, size, and placement of windows) collaborate with an expert in passenger ergonomics? What about the chemical influences of fuel mixture and oil on the materials of the engine and transmission over the lifetime of the vehicle? There are other questions we can ask about the design of an automobile, but the end result is the same: success in modern technical endeavors absolutely requires multiple perspectives and expertise to collaborate.

In order for a field or discipline to progress and mature, it needs to reach a point where it can thoughtfully reflect on its origins, seek out a diverse set of perspectives on those reflections, and place that synthesis into a context that is useful for how the community pictures the future.

This book represents such a synthesis and should be seen as a seminal collection of perspectives on the (I will argue, still emerging and quickly evolving) field of software engineering and operations.

No matter what industry you are in, or what product or service your organization provides, this way of thinking is paramount and necessary for survival for every business and technology leader.

—John Allspaw, CTO, Etsy
Brooklyn, NY, August 2016

Imagine a World Where Dev and Ops Become DevOps

An Introduction to The DevOps Handbook

Imagine a world where product owners, Development, QA, IT Operations, and Infosec work together, not only to help each other, but also to ensure that the overall organization succeeds. By working toward a common goal, they enable the fast flow of planned work into production (e.g., performing tens, hundreds, or even thousands of code deploys per day), while achieving world-class stability, reliability, availability, and security.

In this world, cross-functional teams rigorously test their hypotheses of which features will most delight users and advance the organizational goals. They care not just about implementing user features, but also actively ensure their work flows smoothly and frequently through the entire value stream without causing chaos and disruption to IT Operations or any other internal or external customer.

Simultaneously, QA, IT Operations, and Infosec are always working on ways to reduce friction for the team, creating the work systems that enable developers to be more productive and get better outcomes. By adding the expertise of QA, IT Operations, and Infosec into delivery teams and automated self-service tools and platforms, teams are able to use that expertise in their daily work without being dependent on other teams.

This enables organizations to create a safe system of work, where small teams are able to quickly and independently develop, test, and deploy code and value quickly, safely, securely, and reliably to customers. This allows organizations to maximize developer productivity, enable organizational learning, create high employee satisfaction, and win in the marketplace.

These are the outcomes that result from DevOps. For most of us, this is not the world we live in. More often than not, the system we work in is broken, resulting in extremely poor outcomes that fall well short of our true potential. In our world, Development and IT Operations are adversaries; testing and Infosec activities happen only at the end of a project, too late to correct any problems found; and almost any critical activity requires too much manual effort and too many handoffs, leaving us to always be waiting. Not only does this contribute to extremely long lead times to get anything done, but the quality of our work, especially production deployments, is also problematic and chaotic, resulting in negative impacts to our customers and our business.

As a result, we fall far short of our goals, and the whole organization is dissatisfied with the performance of IT, resulting in budget reductions and frustrated, unhappy employees who feel powerless to change the process and its outcomes.[†] The solution? We need to change how we work; DevOps shows us the best way forward.

To better understand the potential of the DevOps revolution, let us look at the Manufacturing Revolution of the 1980s. By adopting Lean principles and practices, manufacturing organizations dramatically improved plant productivity, customer lead times, product quality, and customer satisfaction, enabling them to win in the marketplace.

Before the revolution, average manufacturing plant order lead times were six weeks, with fewer than 70% of orders being shipped on time. By 2005, with the widespread implementation of Lean practices, average product lead times had dropped to less than three weeks, and more than 95% of orders were being shipped on time. Organizations that did not implement Lean practices lost market share, and many went out of business entirely.

Similarly, the bar has been raised for delivering technology products and services—what was good enough in previous decades is not good enough now. For each of the last four decades, the cost and time required to develop and deploy strategic business capabilities and features has dropped by orders of magnitude. During the 1970s and 1980s, most new features required one to five years to develop and deploy, often costing tens of millions of dollars.

By the 2000's, because of advances in technology and the adoption of Agile principles and practices, the time required to develop new functionality had

[†] This is just a small sample of the problems found in typical IT organizations.

dropped to weeks or months, but deploying into production would still require weeks or months, often with catastrophic outcomes.

And by 2010, with the introduction of DevOps and the neverending commoditization of hardware, software, and now the cloud, features (and even entire startup companies) could be created in weeks, quickly being deployed into production in just hours or minutes—for these organizations, deployment finally became routine and low risk. These organizations are able to perform experiments to test business ideas, discovering which ideas create the most value for customers and the organization as a whole, which are then further developed into features that can be rapidly and safely deployed into production.

Table 1. *The ever accelerating trend toward faster, cheaper, low-risk delivery of software*

	1970s–1980s	1990s	2000s–Present
Era	Mainframes	Client/Server	Commoditization and Cloud
Representative technology of era	COBOL, DB2 on MVS, etc.	C++, Oracle, Solaris, etc.	Java, MySQL, Red Hat, Ruby on Rails, PHP, etc.
Cycle time	1–5 years	3–12 months	2–12 weeks
Cost	\$1M–\$100M	\$100k–\$10M	\$10k–\$1M
At risk	The whole company	A product line or division	A product feature
Cost of failure	Bankruptcy, sell the company, massive layoffs	Revenue miss, CIO's job	Negligible

(Source: Adrian Cockcroft, “Velocity and Volume (or Speed Wins),” presentation at FlowCon, San Francisco, CA, November 2013.)

Today, organizations adopting DevOps principles and practices often deploy changes hundreds or even thousands of times per day. In an age where competitive advantage requires fast time to market and relentless experimentation, organizations that are unable to replicate these outcomes are destined to lose in the marketplace to more nimble competitors and could potentially go out of business entirely, much like the manufacturing organizations that did not adopt Lean principles.

These days, regardless of what industry we are competing in, the way we acquire customers and deliver value to them is dependent on the technology value stream. Put even more succinctly, as Jeffrey Immelt, CEO of General Electric, stated, “Every industry and company that is not bringing software to the core of their business will be disrupted.” Or as Jeffrey Snover, Technical Fellow at Microsoft, said, “In previous economic eras, businesses created value by moving atoms. Now they create value by moving bits.”

It’s difficult to overstate the enormity of this problem—it affects every organization, independent of the industry we operate in, the size of our organization, whether we are profit or non-profit. Now more than ever, how technology work is managed and performed predicts whether our organizations will win in the marketplace, or even survive. In many cases, we will need to adopt principles and practices that look very different from those that have successfully guided us over the past decades. (See Appendix 1.)

Now that we have established the urgency of the problem that DevOps solves, let us take some time to explore in more detail the symptomatology of the problem, why it occurs, and why, without dramatic intervention, the problem worsens over time.

THE PROBLEM: SOMETHING IN YOUR ORGANIZATION MUST NEED IMPROVEMENT (OR YOU WOULDN’T BE READING THIS BOOK)

Most organizations are not able to deploy production changes in minutes or hours, instead requiring weeks or months. Nor are they able to deploy hundreds or thousands of changes into production per day; instead, they struggle to deploy monthly or even quarterly. Nor are production deployments routine, instead involving outages and chronic firefighting and heroics.

In an age where competitive advantage requires fast time to market, high service levels, and relentless experimentation, these organizations are at a significant competitive disadvantage. This is in large part due to their inability to resolve a core, chronic conflict within their technology organization.

THE CORE, CHRONIC CONFLICT

In almost every IT organization, there is an inherent conflict between Development and IT Operations which creates a downward spiral, resulting in

ever-slower time to market for new products and features, reduced quality, increased outages, and, worst of all, an ever-increasing amount of technical debt.

The term “technical debt” was first coined by Ward Cunningham. Analogous to financial debt, technical debt describes how decisions we make lead to problems that get increasingly more difficult to fix over time, continually reducing our available options in the future—even when taken on judiciously, we still incur interest.

One factor that contributes to this is the often competing goals of Development and IT Operations. IT organizations are responsible for many things. Among them are the two following goals, which must be pursued simultaneously:

- Respond to the rapidly changing competitive landscape
- Provide stable, reliable, and secure service to the customer

Frequently, Development will take responsibility for responding to changes in the market, deploying features and changes into production as quickly as possible. IT Operations will take responsibility for providing customers with IT service that is stable, reliable, and secure, making it difficult or even impossible for anyone to introduce production changes that could jeopardize production. Configured this way, Development and IT Operations have diametrically opposed goals and incentives.

Dr. Eliyahu M. Goldratt, one of the founders of the manufacturing management movement, called these types of configuration “the core, chronic conflict”—when organizational measurements and incentives across different silos prevent the achievement of global, organizational goals.[†]

This conflict creates a downward spiral so powerful it prevents the achievement of desired business outcomes, both inside and outside the IT organization. These chronic conflicts often put technology workers into situations that lead to poor software and service quality, and bad customer outcomes, as well as a daily need for workarounds, firefighting, and heroics, whether in Product

[†] In the manufacturing realm, a similar core, chronic conflict existed: the need to simultaneously ensure on-time shipments to customers and control costs. How this core, chronic conflict was broken is described in Appendix 2.

Management, Development, QA, IT Operations, or Information Security. (See Appendix 2.)

DOWNWARD SPIRAL IN THREE ACTS

The downward spiral in IT has three acts that are likely familiar to most IT practitioners.

The first act begins in IT Operations, where our goal is to keep applications and infrastructure running so that our organization can deliver value to customers. In our daily work, many of our problems are due to applications and infrastructure that are complex, poorly documented, and incredibly fragile. This is the technical debt and daily workarounds that we live with constantly, always promising that we'll fix the mess when we have a little more time. But that time never comes.

Alarming, our most fragile artifacts support either our most important revenue-generating systems or our most critical projects. In other words, the systems most prone to failure are also our most important and are at the epicenter of our most urgent changes. When these changes fail, they jeopardize our most important organizational promises, such as availability to customers, revenue goals, security of customer data, accurate financial reporting, and so forth.

The second act begins when somebody has to compensate for the latest broken promise—it could be a product manager promising a bigger, bolder feature to dazzle customers with or a business executive setting an even larger revenue target. Then, oblivious to what technology can or can't do, or what factors led to missing our earlier commitment, they commit the technology organization to deliver upon this new promise.

As a result, Development is tasked with another urgent project that inevitably requires solving new technical challenges and cutting corners to meet the promised release date, further adding to our technical debt—made, of course, with the promise that we'll fix any resulting problems when we have a little more time.

This sets the stage for the third and final act, where everything becomes just a little more difficult, bit by bit—everybody gets a little busier, work takes a little more time, communications become a little slower, and work queues get a little longer. Our work becomes more tightly coupled, smaller actions cause bigger failures, and we become more fearful and less tolerant of making

changes. Work requires more communication, coordination, and approvals; teams must wait just a little longer for their dependent work to get done; and our quality keeps getting worse. The wheels begin grinding slower and require more effort to keep turning. (See Appendix 3.)

Although it's difficult to see in the moment, the downward spiral is obvious when one takes a step back. We notice that production code deployments are taking ever-longer to complete, moving from minutes to hours to days to weeks. And worse, the deployment outcomes have become even more problematic, that resulting in an ever-increasing number of customer-impacting outages that require more heroics and firefighting in Operations, further depriving them of their ability to pay down technical debt.

As a result, our product delivery cycles continue to move slower and slower, fewer projects are undertaken, and those that are, are less ambitious. Furthermore, the feedback on everyone's work becomes slower and weaker, especially the feedback signals from our customers. And, regardless of what we try, things seem to get worse—we are no longer able to respond quickly to our changing competitive landscape, nor are we able to provide stable, reliable service to our customers. As a result, we ultimately lose in the marketplace.

Time and time again, we learn that when IT fails, the entire organization fails. As Steven J. Spear noted in his book *The High-Velocity Edge*, whether the damages “unfold slowly like a wasting disease” or rapidly “like a fiery crash... the destruction can be just as complete.”

WHY DOES THIS DOWNWARD SPIRAL HAPPEN EVERYWHERE?

For over a decade, the authors of this book have observed this destructive spiral occur in countless organizations of all types and sizes. We understand better than ever why this downward spiral occurs and why it requires DevOps principles to mitigate. First, as described earlier, every IT organization has two opposing goals, and second, every company is a technology company, whether they know it or not.

As Christopher Little, a software executive and one of the earliest chroniclers of DevOps, said, “Every company is a technology company, regardless of what business they think they're in. A bank is just an IT company with a banking license.”[†]

[†] In 2013, the European bank HSBC employed more software developers than Google.

To convince ourselves that this is the case, consider that the vast majority of capital projects have some reliance upon IT. As the saying goes, “It is virtually impossible to make any business decision that doesn’t result in at least one IT change.”

In the business and finance context, projects are critical because they serve as the primary mechanism for change inside organizations. Projects are typically what management needs to approve, budget for, and be held accountable for; therefore, they are the mechanism that achieve the goals and aspirations of the organization, whether it is to grow or even shrink.[†]

Projects are typically funded through capital spending (i.e., factories, equipment, and major projects, and expenditures are capitalized when payback is expected to take years), of which 50% is now technology related. This is even true in “low tech” industry verticals with the lowest historical spending on technology, such as energy, metal, resource extraction, automotive, and construction. In other words, business leaders are far more reliant upon the effective management of IT in order to achieve their goals than they think.[‡]

THE COSTS: HUMAN AND ECONOMIC

When people are trapped in this downward spiral for years, especially those who are downstream of Development, they often feel stuck in a system that pre-ordains failure and leaves them powerless to change the outcomes. This powerlessness is often followed by burnout, with the associated feelings of fatigue, cynicism, and even hopelessness and despair.

Many psychologists assert that creating systems that cause feelings of powerlessness is one of the most damaging things we can do to fellow human beings—we deprive other people of their ability to control their own outcomes and even create a culture where people are afraid to do the right thing because of fear of punishment, failure, or jeopardizing their livelihood. This can create

† For now, let us suspend the discussion of whether software should be funded as a “project” or a “product.” This is discussed later in the book.

‡ For instance, Dr. Vernon Richardson and his colleagues published this astonishing finding. They studied the 10-K SEC filings of 184 public corporations and divided them into three groups: A) firms with material weaknesses with IT-related deficiencies, B) firms with material weaknesses with no IT-related deficiencies, and C) “clean firms” with no material weaknesses. Firms in Group A saw eight times higher CEO turnover than Group C, and there was four times higher CFO turnover in Group A than in Group C. Clearly, IT may matter far more than we typically think.

the conditions of *learned helplessness*, where people become unwilling or unable to act in a way that avoids the same problem in the future.

For our employees, it means long hours, working on weekends, and a decreased quality of life, not just for the employee, but for everyone who depends on them, including family and friends. It is not surprising that when this occurs, we lose our best people (except for those that feel like they can't leave, because of a sense of duty or obligation).

In addition to the human suffering that comes with the current way of working, the opportunity cost of the value that we could be creating is staggering—the authors believe that we are missing out on approximately \$2.6 trillion of value creation per year, which is, at the time of this writing, equivalent to the annual economic output of France, the sixth largest economy in the world.

Consider the following calculation—both IDC and Gartner estimated that in 2011, approximately 5% of the worldwide gross domestic product (\$3.1 trillion) was spent on IT (hardware, services, and telecom). If we estimate that 50% of that \$3.1 trillion was spent on operating costs and maintaining existing systems, and that one-third of that 50% was spent on urgent and unplanned work or rework, approximately \$520 billion was wasted.

If adopting DevOps could enable us, through better management and increased operational excellence, to halve that waste and redeploy that human potential into something that's five times the value (a modest proposal), we could create \$2.6 trillion of value per year.

THE ETHICS OF DEVOPS: THERE IS A BETTER WAY

In the previous sections, we described the problems and the negative consequences of the status quo due to the core, chronic conflict, from the inability to achieve organizational goals, to the damage we inflict on fellow human beings. By solving these problems, DevOps astonishingly enables us to simultaneously improve organizational performance, achieve the goals of all the various functional technology roles (e.g., Development, QA, IT Operations, Infosec), and improve the human condition.

This exciting and rare combination may explain why DevOps has generated so much excitement and enthusiasm in so many in such a short time, including technology leaders, engineers, and much of the software ecosystem we reside in.

BREAKING THE DOWNWARD SPIRAL WITH DEVOPS

Ideally, small teams of developers independently implement their features, validate their correctness in production-like environments, and have their code deployed into production quickly, safely and securely. Code deployments are routine and predictable. Instead of starting deployments at midnight on Friday and spending all weekend working to complete them, deployments occur throughout the business day when everyone is already in the office and without our customers even noticing—except when they see new features and bug fixes that delight them. And, by deploying code in the middle of the workday, for the first time in decades IT Operations is working during normal business hours like everyone else.

By creating fast feedback loops at every step of the process, everyone can immediately see the effects of their actions. Whenever changes are committed into version control, fast automated tests are run in production-like environments, giving continual assurance that the code and environments operate as designed and are always in a secure and deployable state.

Automated testing helps developers discover their mistakes quickly (usually within minutes), which enables faster fixes as well as genuine learning—learning that is impossible when mistakes are discovered six months later during integration testing, when memories and the link between cause and effect have long faded. Instead of accruing technical debt, problems are fixed as they are found, mobilizing the entire organization if needed, because global goals outweigh local goals.

Pervasive production telemetry in both our code and production environments ensure that problems are detected and corrected quickly, confirming that everything is working as intended and customers are getting value from the software we create.

In this scenario, everyone feels productive—the architecture allows small teams to work safely and architecturally decoupled from the work of other teams who use self-service platforms that leverage the collective experience of Operations and Information Security. Instead of everyone waiting all the time, with large amounts of late, urgent rework, teams work independently and productively in small batches, quickly and frequently delivering new value to customers.

Even high-profile product and feature releases become routine by using dark launch techniques. Long before the launch date, we put all the required code for the feature into production, invisible to everyone except internal employees

and small cohorts of real users, allowing us to test and evolve the feature until it achieves the desired business goal.

And, instead of firefighting for days or weeks to make the new functionality work, we merely change a feature toggle or configuration setting. This small change makes the new feature visible to ever-larger segments of customers, automatically rolling back if something goes wrong. As a result, our releases are controlled, predictable, reversible, and low stress.

It's not just feature releases that are calmer—all sorts of problems are being found and fixed early, when they are smaller, cheaper, and easier to correct. With every fix, we also generate organizational learnings, enabling us to prevent the problem from recurring and enabling us to detect and correct similar problems faster in the future.

Furthermore, everyone is constantly learning, fostering a hypothesis-driven culture where the scientific method is used to ensure nothing is taken for granted—we do nothing without measuring and treating product development and process improvement as experiments.

Because we value everyone's time, we don't spend years building features that our customers don't want, deploying code that doesn't work, or fixing something that isn't actually the cause of our problem.

Because we care about achieving goals, we create long-term teams that are responsible for meeting them. Instead of project teams where developers are reassigned and shuffled around after each release, never receiving feedback on their work, we keep teams intact so they can keep iterating and improving, using those learnings to better achieve their goals. This is equally true for the product teams who are solving problems for our external customers, as well as our internal platform teams who are helping other teams be more productive, safe, and secure.

Instead of a culture of fear, we have a high-trust, collaborative culture, where people are rewarded for taking risks. They are able to fearlessly talk about problems as opposed to hiding them or putting them on the backburner—after all, we must see problems in order to solve them.

And, because everyone fully owns the quality of their work, everyone builds automated testing into their daily work and uses peer reviews to gain confidence that problems are addressed long before they can impact a customer. These processes mitigate risk, as opposed to approvals from distant authorities,

allowing us to deliver value quickly, reliably, and securely—even proving to skeptical auditors that we have an effective system of internal controls.

And when something does go wrong, we conduct *blameless post-mortems*, not to punish anyone, but to better understand what caused the accident and how to prevent it. This ritual reinforces our culture of learning. We also hold internal technology conferences to elevate our skills and ensure that everyone is always teaching and learning.

Because we care about quality, we even inject faults into our production environment so we can learn how our system fails in a planned manner. We conduct planned exercises to practice large-scale failures, randomly kill processes and compute servers in production, and inject network latencies and other nefarious acts to ensure we grow ever more resilient. By doing this, we enable better resilience, as well as organizational learning and improvement.

In this world, everyone has ownership in their work, regardless of their role in the technology organization. They have confidence that their work matters and is meaningfully contributing to organizational goals, proven by their low-stress work environment and their organization's success in the marketplace. Their proof is that the organization is indeed winning in the marketplace.

THE BUSINESS VALUE OF DEVOPS

We have decisive evidence of the business value of DevOps. From 2013 through 2016, as part of Puppet Labs' *State Of DevOps Report*, to which authors Jez Humble and Gene Kim contributed, we collected data from over twenty-five thousand technology professionals, with the goal of better understanding the health and habits of organizations at all stages of DevOps adoption.

The first surprise this data revealed was how much high performing organizations using DevOps practices were outperforming their non-high performing peers in the following areas:

- Throughput metrics
 - Code and change deployments (thirty times more frequent)
 - Code and change deployment lead time (two hundred times faster)
- Reliability metrics

- Production deployments (sixty times higher change success rate)
- Mean time to restore service (168 times faster)
- Organizational performance metrics
- Productivity, market share, and profitability goals (two times more likely to exceed)
- Market capitalization growth (50% higher over three years)

In other words, high performers were both more agile and more reliable, providing empirical evidence that DevOps enables us to break the core, chronic conflict. High performers deployed code thirty times more frequently, and the time required to go from “code committed” to “successfully running in production” was two hundred times faster—high performers had lead times measured in minutes or hours, while low performers had lead times measured in weeks, months, or even quarters.

Furthermore, high performers were twice as likely to exceed profitability, market share, and productivity goals. And, for those organizations that provided a stock ticker symbol, we found that high performers had 50% higher market capitalization growth over three years. They also had higher employee job satisfaction, lower rates of employee burnout, and their employees were 2.2 times more likely to recommend their organization to friends as a great place to work.[†] High performers also had better information security outcomes. By integrating security objectives into all stages of the development and operations processes, they spent 50% less time remediating security issues.

DEVOPS HELPS SCALE DEVELOPER PRODUCTIVITY

When we increase the number of developers, individual developer productivity often significantly decreases due to communication, integration, and testing overhead. This is highlighted in the famous book by Frederick Brook, *The Mythical Man-Month*, where he explains that when projects are late, adding

[†] As measured by employee Net Promoter Score (eNPS). This is a significant finding, as research has shown that “companies with highly engaged workers grew revenues two and a half times as much as those with low engagement levels. And [publicly traded] stocks of companies with a high-trust work environment outperformed market indexes by a factor of three from 1997 through 2011.”

more developers not only decreases individual developer productivity but also decreases overall productivity.

On the other hand, DevOps shows us that when we have the right architecture, the right technical practices, and the right cultural norms, small teams of developers are able to quickly, safely, and independently develop, integrate, test, and deploy changes into production. As Randy Shoup, formerly a director of engineering at Google, observed, large organizations using DevOps “have thousands of developers, but their architecture and practices enable small teams to still be incredibly productive, as if they were a startup.”

The 2015 *State of DevOps Report* examined not only “deploys per day” but also “deploys per day per developer.” We hypothesized that high performers would be able to scale their number of deployments as team sizes grew.

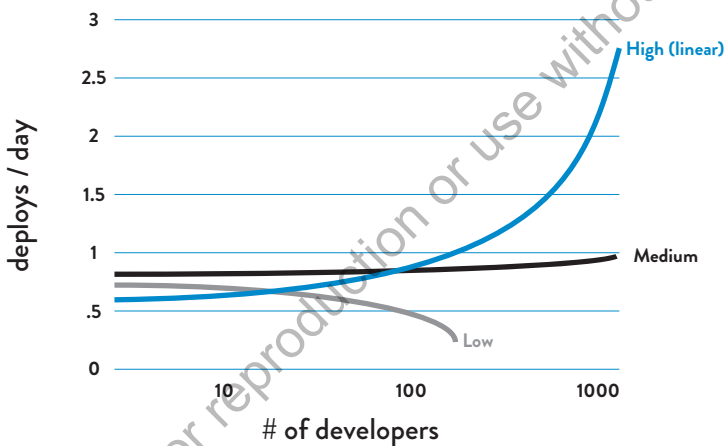


Figure 1. Deployments/day vs. number of developers

(Source: Puppet Labs, 2015 State Of DevOps Report.)[†]

Indeed, this is what we found. Figure 1 shows that in low performers, deploys per day per developer go down as team size increases, stays constant for medium performers, and increases linearly for high performers.

In other words, organizations adopting DevOps are able to linearly increase the number of deploys per day as they increase their number of developers, just as Google, Amazon, and Netflix have done.[‡]

[†] Only organizations that are deploying at least once per day are shown.

[‡] Another more extreme example is Amazon. In 2011, Amazon was performing approximately seven thousand deploys per day. By 2015, they were performing 130,000 deploys per day.

THE UNIVERSALITY OF THE SOLUTION

One of the most influential books in the Lean manufacturing movement is *The Goal: A Process of Ongoing Improvement* written by Dr. Eliyahu M. Goldratt in 1984. It influenced an entire generation of professional plant managers around the world. It was a novel about a plant manager who had to fix his cost and product due date issues in ninety days, otherwise his plant would be shut down.

Later in his career, Dr. Goldratt described the letters he received in response to *The Goal*. These letters would typically read, “You have obviously been hiding in our factory, because you’ve described my life [as a plant manager] exactly...” Most importantly, these letters showed people were able to replicate the breakthroughs in performance that were described in the book in their own work environments.

The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win, written by Gene Kim, Kevin Behr, and George Spafford in 2013, was closely modeled after *The Goal*. It is a novel that follows an IT leader who faces all the typical problems that are endemic in IT organizations: an over-budget, behind-schedule project that must get to market in order for the company to survive. He experiences catastrophic deployments; problems with availability, security, and compliance; and so forth. Ultimately, he and his team use DevOps principles and practices to overcome those challenges, helping their organization win in the marketplace. In addition, the novel shows how DevOps practices improved the workplace environment for the team, creating lower stress and higher satisfaction because of greater practitioner involvement throughout the process.

As with *The Goal*, there is tremendous evidence of the universality of the problems and solutions described in *The Phoenix Project*. Consider some of the statements found in the Amazon reviews: “I find myself relating to the characters in *The Phoenix Project*...I’ve probably met most of them over the course of my career,” “If you have ever worked in any aspect of IT, DevOps, or Infosec you will definitely be able to relate to this book,” or “There’s not a character in *The Phoenix Project* that I don’t identify with myself or someone I know in real life... not to mention the problems faced and overcome by those characters.”

In the remainder of this book, we will describe how to replicate the transformation described in *The Phoenix Project*, as well provide many case studies of how other organizations have used DevOps principles and practices to replicate those outcomes.

THE DEVOPS HANDBOOK: AN ESSENTIAL GUIDE

The purpose of the *DevOps Handbook* is to give you the theory, principles, and practices you need to successfully start your DevOps initiative and achieve your desired outcomes. This guidance is based on decades of sound management theory, study of high performing technology organizations, work we have done helping organizations transform, and research that validates the effectiveness of the prescribed DevOps practices. As well as interviews with relevant subject matter experts and analyses of nearly one hundred case studies presented at the DevOps Enterprise Summit.

Broken into six parts, this book covers DevOps theories and principles using the Three Ways, a specific view of the underpinning theory originally introduced in *The Phoenix Project*. *The DevOps Handbook* is for everyone who performs or influences work in the technology value stream (which typically includes Product Management, Development, QA, IT Operations, and Information Security), as well as for business and marketing leadership, where most technology initiatives originate.

The reader is not expected to have extensive knowledge of any of these domains, or of DevOps, Agile, ITIL, Lean, or process improvement. Each of these topics is introduced and explained in the book as it becomes necessary.

Our intent is to create a working knowledge of the critical concepts in each of these domains, both to serve as a primer and to introduce the language necessary to help practitioners work with all their peers across the entire IT value stream, and to frame shared goals.

This book will be of value to business leaders and stakeholders who are increasingly reliant upon the technology organization for the achievement of their goals.

Furthermore, this book is intended for readers whose organizations might not be experiencing all the problems described in the book (e.g., long deployment lead times or painful deployments). Even readers in this fortunate position will benefit from understanding DevOps principles, especially those relating to shared goals, feedback, and continual learning.

In Part I, we present a brief history of DevOps and introduce the underpinning theory and key themes from relevant bodies of knowledge that span over decades. We then present the high level principles of the Three Ways: Flow, Feedback, and Continual Learning and Experimentation.

Part II describes how and where to start, and presents concepts such as value streams, organizational design principles and patterns, organizational adoption patterns, and case studies.

Part III describes how to accelerate Flow by building the foundations of our deployment pipeline: enabling fast and effective automated testing, continuous integration, continuous delivery, and architecting for low-risk releases.

Part IV discusses how to accelerate and amplify Feedback by creating effective production telemetry to see and solve problems, better anticipate problems and achieve goals, enable feedback so that Dev and Ops can safely deploy changes, integrate A/B testing into our daily work, and create review and coordination processes to increase the quality of our work.

Part V describes how we accelerate Continual Learning by establishing a just culture, converting local discoveries into global improvements, and properly reserving time to create organizational learning and improvements.

Finally, in Part VI we describe how to properly integrate security and compliance into our daily work, by integrating preventative security controls into shared source code repositories and services, integrating security into our deployment pipeline, enhancing telemetry to better enable detection and recovery, protecting the deployment pipeline, and achieving change management objectives.

By codifying these practices, we hope to accelerate the adoption of DevOps practices, increase the success of DevOps initiatives, and lower the activation energy required for DevOps transformations.

Excerpt - Not for reproduction or use without permission

PART I

The Three Ways

Excerpt - Not for reproduction or use without permission



Part I

Introduction

In Part I of *The DevOps Handbook*, we will explore how the convergence of several important movements in management and technology set the stage for the DevOps movement. We describe value streams, how DevOps is the result of applying Lean principles to the technology value stream, and the Three Ways: Flow, Feedback, and Continual Learning and Experimentation.

Primary focuses within these chapters include:

- The principles of Flow, which accelerate the delivery of work from Development to Operations to our customers
- The principles of Feedback, which enable us to create ever safer systems of work
- The principles of Continual Learning and Experimentation foster a high-trust culture and a scientific approach to organizational improvement risk-taking as part of our daily work

A BRIEF HISTORY

DevOps and its resulting technical, architectural, and cultural practices represent a convergence of many philosophical and management movements. While many organizations have developed these principles independently, understanding that DevOps resulted from a broad stroke of movements, a phenomenon described by John Willis (one of the co-authors of this book) as the “convergence of DevOps,” shows an amazing progression of thinking and improbable connections. There are decades of lessons learned from manufacturing, high reliability organization, high-trust management models, and others that have brought us to the DevOps practices we know today.

DevOps is the outcome of applying the most trusted principles from the domain of physical manufacturing and leadership to the IT value stream. DevOps relies on bodies of knowledge from Lean, Theory of Constraints, the Toyota Production System, resilience engineering, learning organizations, safety culture, human factors, and many others. Other valuable contexts that DevOps draws from include high-trust management cultures, servant leadership, and organizational change management. The result is world-class quality, reliability, stability, and security at ever lower cost and effort; and accelerated flow and reliability throughout the technology value stream, including Product Management, Development, QA, IT Operations, and Infosec.

While the foundation of DevOps can be seen as being derived from Lean, the Theory of Constraints, and the Toyota Kata movement, many also view DevOps as the logical continuation of the Agile software journey that began in 2001.

THE LEAN MOVEMENT

Techniques such as Value Stream Mapping, Kanban Boards, and Total Productive Maintenance were codified for the Toyota Production System in the 1980s. In 1997, the Lean Enterprise Institute started researching applications of Lean to other value streams, such as the service industry and healthcare.

Two of Lean's major tenets include the deeply held belief that *manufacturing lead time* required to convert raw materials into finished goods was the best predictor of quality, customer satisfaction, and employee happiness, and that one of the best predictors of short lead times was small batch sizes of work.

Lean principles focus on how to create value for the customer through systems thinking by creating constancy of purpose, embracing scientific thinking, creating flow and pull (versus push), assuring quality at the source, leading with humility, and respecting every individual.

THE AGILE MANIFESTO

The Agile Manifesto was created in 2001 by seventeen of the leading thinkers in software development. They wanted to create a lightweight set of values and principles against heavyweight software development processes such as waterfall development, and methodologies such as the Rational Unified Process.

One key principle was to “deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale,” emphasizing the desire for small batch sizes, incremental releases instead of large, waterfall releases. Other principles emphasized the need for small, self-motivated teams, working in a high-trust management model.

Agile is credited for dramatically increasing the productivity of many development organizations. And interestingly, many of the key moments in DevOps history also occurred within the Agile community or at Agile conferences, as described below.

AGILE INFRASTRUCTURE AND VELOCITY MOVEMENT

At the 2008 Agile conference in Toronto, Canada, Patrick Debois and Andrew Schaffer held a “birds of a feather” session on applying Agile principles to infrastructure as opposed to application code. Although they were the only people who showed up, they rapidly gained a following of like-minded thinkers, including co-author John Willis.

Later, at the 2009 Velocity conference, John Allspaw and Paul Hammond gave the seminal “10 Deploys per Day: Dev and Ops Cooperation at Flickr” presentation, where they described how they created shared goals between Dev and Ops and used continuous integration practices to make deployment part of everyone’s daily work. According to first hand accounts, everyone attending the presentation immediately knew they were in the presence of something profound and of historic significance.

Patrick Debois was not there, but was so excited by Allspaw and Hammond’s idea that he created the first DevOpsDays in Ghent, Belgium, (where he lived) in 2009. There the term “DevOps” was coined.

THE CONTINUOUS DELIVERY MOVEMENT

Building upon the development discipline of continuous build, test, and integration, Jez Humble and David Farley extended the concept to *continuous delivery*, which defined the role of a “deployment pipeline” to ensure that code and infrastructure are always in a deployable state, and that all code checked in to trunk can be safely deployed into production. This idea was first presented at the 2006 Agile conference, and was also independently

developed in 2009 by Tim Fitz in a blog post on his website titled “Continuous Deployment.”†

TOYOTA KATA

In 2009, Mike Rother wrote *Toyota Kata: Managing People for Improvement, Adaptiveness and Superior Results*, which framed his twenty-year journey to understand and codify the Toyota Production System. He had been one of the graduate students who flew with GM executives to visit Toyota plants and helped develop the Lean toolkit, but he was puzzled when none of the companies adopting these practices replicated the level of performance observed at the Toyota plants.

He concluded that the Lean community missed the most important practice of all, which he called the *improvement kata*. He explains that every organization has work routines, and the improvement kata requires creating structure for the daily, habitual practice of improvement work, because daily practice is what improves outcomes. The constant cycle of establishing desired future states, setting weekly target outcomes, and the continual improvement of daily work is what guided improvement at Toyota.

The above describes the history of DevOps and relevant movements that it draws upon. Throughout the rest of Part I, we look at value streams, how Lean principles can be applied to the technology value stream, and the Three Ways of Flow, Feedback, and Continual Learning and Experimentation.

† DevOps also extends and builds upon the practices of *infrastructure as code*, which was pioneered by Dr. Mark Burgess, Luke Kanies, and Adam Jacob. In infrastructure as code, the work of Operations is automated and treated like application code, so that modern development practices can be applied to the entire development stream. This further enabled fast deployment flow, including continuous integration (pioneered by Grady Booch and integrated as one of the key 12 practices of Extreme Programming), continuous delivery (pioneered by Jez Humble and David Farley), and continuous deployment (pioneered by Etsy, Wealthfront, and Eric Ries’s work at IMVU).

1 Agile, Continuous Delivery, and the Three Ways

In this chapter, an introduction to the underpinning theory of Lean Manufacturing is presented, as well as the Three Ways, the principles from which all of the observed DevOps behaviors can be derived.

Our focus here is primarily on theory and principles, describing many decades of lessons learned from manufacturing, high-reliability organizations, high-trust management models, and others, from which DevOps practices have been derived. The resulting concrete principles and patterns, and their practical application to the technology value stream, are presented in the remaining chapters of the book.

THE MANUFACTURING VALUE STREAM

One of the fundamental concepts in Lean is the *value stream*. We will define it first in the context of manufacturing and then extrapolate how it applies to DevOps and the technology value stream.

Karen Martin and Mike Osterling define value stream in their book *Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation* as “the sequence of activities an organization undertakes to deliver upon a customer request,” or “the sequence of activities required to design, produce, and deliver a good or service to a customer, including the dual flows of information and material.”

In manufacturing operations, the value stream is often easy to see and observe: it starts when a customer order is received and the raw materials are released onto the plant floor. To enable fast and predictable lead times in any value stream, there is usually a relentless focus on creating a smooth and even flow of work, using techniques such as small batch sizes, reducing work in process

(WIP), preventing rework to ensure we don't pass defects to downstream work centers, and constantly optimizing our system toward our global goals.

THE TECHNOLOGY VALUE STREAM

The same principles and patterns that enable the fast flow of work in physical processes are equally applicable to technology work (and, for that matter, for all knowledge work). In DevOps, we typically define our technology value stream as the process required to convert a business hypothesis into a technology-enabled service that delivers value to the customer.

The input to our process is the formulation of a business objective, concept, idea, or hypothesis, and starts when we accept the work in Development, adding it to our committed backlog of work.

From there, Development teams that follow a typical Agile or iterative process will likely transform that idea into user stories and some sort of feature specification, which is then implemented in code into the application or service being built. The code is then checked in to the version control repository, where each change is integrated and tested with the rest of the software system.

Because value is created only when our services are running in production, we must ensure that we are not only delivering fast flow, but that our deployments can also be performed without causing chaos and disruptions such as service outages, service impairments, or security or compliance failures.

FOCUS ON DEPLOYMENT LEAD TIME

For the remainder of this book, our attention will be on deployment lead time, a subset of the value stream described above. This value stream begins when any engineer[†] in our value stream (which includes Development, QA, IT Operations, and Infosec) checks a change into version control and ends when that change is successfully running in production, providing value to the customer and generating useful feedback and telemetry.

The first phase of work that includes Design and Development is akin to Lean Product Development and is highly variable and highly uncertain, often requiring high degrees of creativity and work that may never be performed again, resulting in high variability of process times. In contrast, the second

[†] Going forward, *engineer* refers to anyone working in our value stream, not just developers.

phase of work, which includes Testing and Operations, is akin to Lean Manufacturing. It requires creativity and expertise, and strives to be predictable and mechanistic, with the goal of achieving work outputs with minimized variability (e.g., short and predictable lead times, near zero defects).

Instead of large batches of work being processed sequentially through the design/development value stream and then through the test/operations value stream (such as when we have a large batch waterfall process or long-lived feature branches), our goal is to have testing and operations happening simultaneously with design/development, enabling fast flow and high quality. This method succeeds when we work in small batches and build quality into every part of our value stream.[‡]

Defining Lead Time vs. Processing Time

In the Lean community, lead time is one of two measures commonly used to measure performance in value streams, with the other being processing time (sometimes known as touch time or task time).[§]

Whereas the lead time clock starts when the request is made and ends when it is fulfilled, the process time clock starts only when we begin work on the customer request—specifically, it omits the time that the work is in queue, waiting to be processed (figure 2).

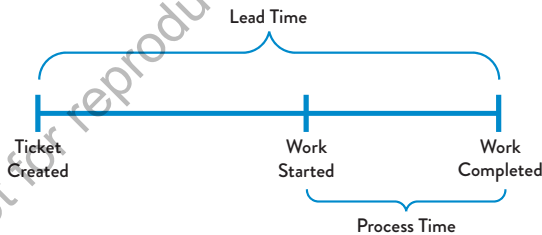


Figure 2. Lead time vs. process time of a deployment operation

Because lead time is what the customer experiences, we typically focus our process improvement attention there instead of on process time. However, the proportion of process time to lead time serves as an important measure

‡ In fact, with techniques such as test-driven development, testing occurs even before the first line of code is written.

§ In this book, the term *process time* will be favored for the same reason Karen Martin and Mike Osterling cite: “To minimize confusion, we avoid using the term cycle time as it has several definitions synonymous with processing time and pace or frequency of output, to name a few.”

of efficiency—achieving fast flow and short lead times almost always requires reducing the time our work is waiting in queues.

The Common Scenario: Deployment Lead Times Requiring Months

In business as usual, we often find ourselves in situations where our deployment lead times require months. This is especially common in large, complex organizations that are working with tightly-coupled, monolithic applications, often with scarce integration test environments, long test and production environment lead times, high reliance on manual testing, and multiple required approval processes. When this occurs, our value stream may look like figure 3:

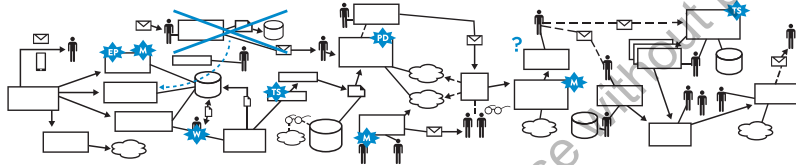


Figure 3: A technology value stream with a deployment lead time of three months
(Source: Damon Edwards, “DevOps Kaizen,” 2015.)

When we have long deployment lead times, heroics are required at almost every stage of the value stream. We may discover that nothing works at the end of the project when we merge all the development team’s changes together, resulting in code that no longer builds correctly or passes any of our tests. Fixing each problem requires days or weeks of investigation to determine who broke the code and how it can be fixed, and still results in poor customer outcomes.

Our DevOps Ideal: Deployment Lead Times of Minutes

In the DevOps ideal, developers receive fast, constant feedback on their work, which enables them to quickly and independently implement, integrate, and validate their code, and have the code deployed into the production environment (either by deploying the code themselves or by others).

We achieve this by continually checking small code changes into our version control repository, performing automated and exploratory testing against it, and deploying it into production. This enables us to have a high degree of confidence that our changes will operate as designed in production and that any problems can be quickly detected and corrected.

This is most easily achieved when we have architecture that is modular, well encapsulated, and loosely-coupled so that small teams are able to work with high degrees of autonomy, with failures being small and contained, and without causing global disruptions.

In this scenario, our deployment lead time is measured in minutes, or, in the worst case, hours. Our resulting value stream map should look something like figure 4:

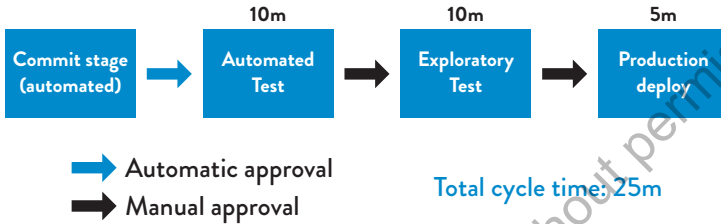


Figure 4: A technology value stream with a lead time of minutes

OBSERVING “%C/A” AS A MEASURE OF REWORK

In addition to lead times and process times, the third key metric in the technology value stream is percent complete and accurate (%C/A). This metric reflects the quality of the output of each step in our value stream. Karen Martin and Mike Osterling state that “the %C/A can be obtained by asking downstream customers what percentage of the time they receive work that is ‘usable as is,’ meaning that they can do their work without having to correct the information that was provided, add missing information that should have been supplied, or clarify information that should have and could have been clearer.”

THE THREE WAYS: THE PRINCIPLES UNDERPINNING DEVOPS

The *Phoenix Project* presents the Three Ways as the set of underpinning principles from which all the observed DevOps behaviors and patterns are derived (figure 5).

The First Way enables fast left-to-right flow of work from Development to Operations to the customer. In order to maximize flow, we need to make work visible, reduce our batch sizes and intervals of work, build in quality by preventing defects from being passed to downstream work centers, and constantly optimize for the global goals.

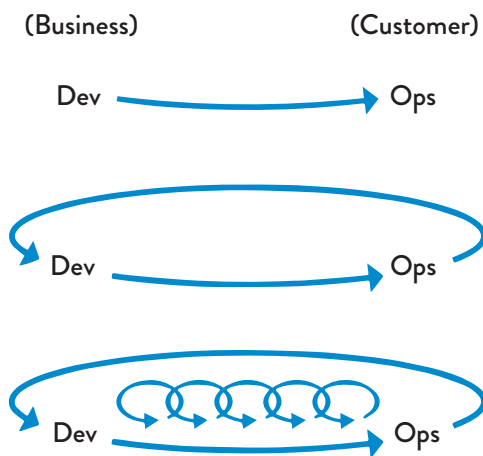


Figure 5: *The Three Ways* (Source: Gene Kim, “The Three Ways: The Principles Underpinning DevOps,” IT Revolution Press blog, accessed August 9, 2016, <http://itrevolution.com/the-three-ways-principles-underpinning-devops/>.)

By speeding up flow through the technology value stream, we reduce the lead time required to fulfill internal or customer requests, especially the time required to deploy code into the production environment. By doing this, we increase the quality of work as well as our throughput, and boost our ability to out-experiment the competition.

The resulting practices include continuous build, integration, test, and deployment processes; creating environments on demand; limiting work in process (WIP); and building systems and organizations that are safe to change.

The Second Way enables the fast and constant flow of feedback from right to left at all stages of our value stream. It requires that we amplify feedback to prevent problems from happening again, or enable faster detection and recovery. By doing this, we create quality at the source and generate or embed knowledge where it is needed—this allows us to create ever-safer systems of work where problems are found and fixed long before a catastrophic failure occurs.

By seeing problems as they occur and swarming them until effective countermeasures are in place, we continually shorten and amplify our feedback loops, a core tenet of virtually all modern process improvement methodologies. This maximizes the opportunities for our organization to learn and improve.

The Third Way enables the creation of a generative, high-trust culture that supports a dynamic, disciplined, and scientific approach to experimentation

and risk-taking, facilitating the creation of organizational learning, both from our successes and failures. Furthermore, by continually shortening and amplifying our feedback loops, we create ever-safer systems of work and are better able to take risks and perform experiments that help us learn faster than our competition and win in the marketplace.

As part of the Third Way, we also design our system of work so that we can multiply the effects of new knowledge, transforming local discoveries into global improvements. Regardless of where someone performs work, they do so with the cumulative and collective experience of everyone in the organization.

CONCLUSION

In this chapter, we described the concepts of value streams, lead time as one of the key measures of the effectiveness for both manufacturing and technology value streams, and the high-level concepts behind each of the Three Ways, the principles that underpin DevOps.

In the following chapters, the principles for each of the Three Ways are described in greater detail. The first of these principles is Flow, which is focused on how we create the fast flow of work in any value stream, whether it's in manufacturing or technology work. The practices that enable fast flow are described in Part III.

2

The First Way: *The Principles of Flow*

In the technology value stream, work typically flows from Development to Operations, the functional areas between our business and our customers. The First Way requires the fast and smooth flow of work from Development to Operations, to deliver value to customers quickly. We optimize for this global goal instead of local goals, such as Development feature completion rates, test find/fix ratios, or Ops availability measures.

We increase flow by making work visible, by reducing batch sizes and intervals of work, and by building quality in, preventing defects from being passed to downstream work centers. By speeding up the flow through the technology value stream, we reduce the lead time required to fulfill internal and external customer requests, further increasing the quality of our work while making us more agile and able to out-experiment the competition.

Our goal is to decrease the amount of time required for changes to be deployed into production and to increase the reliability and quality of those services. Clues on how we do this in the technology value stream can be gleaned from how the Lean principles were applied to the manufacturing value stream.

MAKE OUR WORK VISIBLE

A significant difference between technology and manufacturing value streams is that our work is invisible. Unlike physical processes, in the technology value stream we cannot easily see where flow is being impeded or when work is piling up in front of constrained work centers. Transferring work between work centers is usually highly visible and slow because inventory must be physically moved.

However, in technology work the move can be done with a click of a button, such as by re-assigning a work ticket to another team. Because it is so easy,

work can bounce between teams endlessly due to incomplete information, or work can be passed onto downstream work centers with problems that remain completely invisible until we are late delivering what we promised to the customer or our application fails in the production environment.

To help us see where work is flowing well and where work is queued or stalled, we need to make our work as visible as possible. One of the best methods of doing this is using visual work boards, such as kanban boards or sprint planning boards, where we can represent work on physical or electronic cards. Work originates on the left (often being pulled from a backlog), is pulled from work center to work center (represented in columns), and finishes when it reaches the right side of the board, usually in a column labeled “done” or “in production.”

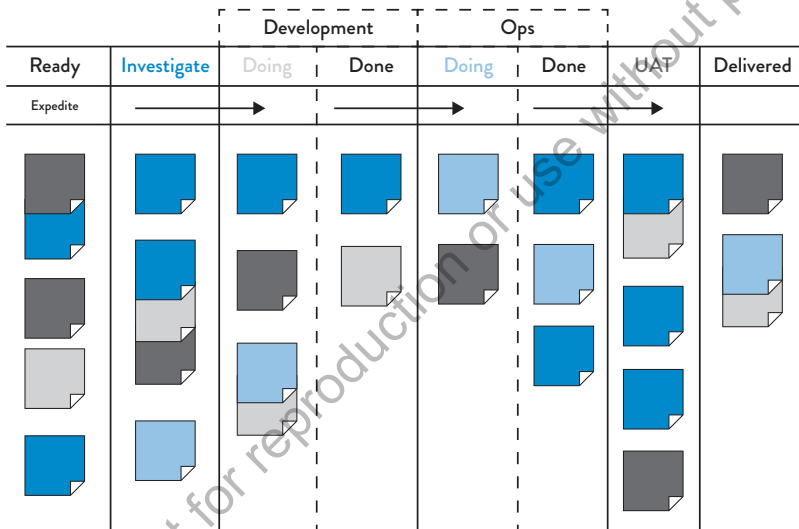


Figure 6: An example kanban board, spanning Requirements, Dev, Test, Staging, and In Production (Source: David J. Andersen and Dominica DeGrandis, Kanban for ITOps, training materials for workshop, 2012.)

Not only does our work become visible, we can also manage our work so that it flows from left to right as quickly as possible. Furthermore, we can measure lead time from when a card is placed on the board to when it is moved into the “Done” column.

Ideally, our kanban board will span the entire value stream, defining work as completed only when it reaches the right side of the board (figure 6). Work is not done when Development completes the implementation of a feature—

rather, it is only done when our application is running successfully in production, delivering value to the customer.

By putting all work for each work center in queues and making it visible, all stakeholders can more easily prioritize work in the context of global goals. Doing this enables each work center to single-task on the highest priority work until it is completed, increasing throughput.

LIMIT WORK IN PROCESS (WIP)

In manufacturing, daily work is typically dictated by a production schedule that is generated regularly (e.g., daily, weekly), establishing which jobs must be run based on customer orders, order due dates, parts available, and so forth.

In technology, our work is usually far more dynamic—this is especially the case in shared services, where teams must satisfy the demands of many different stakeholders. As a result, daily work becomes dominated by the priority *du jour*, often with requests for urgent work coming in through every communication mechanism possible, including ticketing systems, outage calls, emails, phone calls, chat rooms, and management escalations.

Disruptions in manufacturing are also highly visible and costly, often requiring breaking the current job and scrapping any incomplete work in process to start the new job. This high level of effort discourages frequent disruptions.

However, interrupting technology workers is easy, because the consequences are invisible to almost everyone, even though the negative impact to productivity may be far greater than in manufacturing. For instance, an engineer assigned to multiple projects must switch between tasks, incurring all the costs of having to re-establish context, as well as cognitive rules and goals.

Studies have shown that the time to complete even simple tasks, such as sorting geometric shapes, significantly degrades when multitasking. Of course, because our work in the technology value stream is far more cognitively complex than sorting geometric shapes, the effects of multitasking on process time is much worse.

We can limit multitasking when we use a kanban board to manage our work, such as by codifying and enforcing WIP (work in progress) limits for each

column or work center that puts an upper limit on the number of cards that can be in a column.

For example, we may set a WIP limit of three cards for testing. When there are already three cards in the test lane, no new cards can be added to the lane unless a card is completed or removed from the “in work” column and put back into queue (i.e., putting the card back to the column to the left). Nothing can be worked on until it is represented first in a work card, reinforcing that all work must be made visible.

Dominica DeGrandis, one of the leading experts on using kanbans in DevOps value streams, notes that “controlling queue size [WIP] is an extremely powerful management tool, as it is one of the few leading indicators of lead time—with most work items, we don’t know how long it will take until it’s actually completed.”

Limiting WIP also makes it easier to see problems that prevent the completion of work.[†] For instance, when we limit WIP, we find that we may have nothing to do because we are waiting on someone else. Although it may be tempting to start new work (i.e., “It’s better to be doing something than nothing”), a far better action would be to find out what is causing the delay and help fix that problem. Bad multitasking often occurs when people are assigned to multiple projects, resulting in many prioritization problems.

In other words, as David J. Andersen, author of *Kanban: Successful Evolutionary Change for Your Technology Business*, quipped, “Stop starting. Start finishing.”

REDUCE BATCH SIZES

Another key component to creating smooth and fast flow is performing work in small batch sizes. Prior to the Lean manufacturing revolution, it was common practice to manufacture in large batch sizes (or lot sizes), especially for operations where job setup or switching between jobs was time-consuming or costly. For example, producing large car body panels requires setting large and heavy dies onto metal stamping machines, a process that could take days. When changeover cost is so expensive, we would often stamp as many panels at a time as possible, creating large batches in order to reduce the number of changeovers.

[†] Taiichi Ohno compared enforcing WIP limits to draining water from the river of inventory in order to reveal all the problems that obstruct fast flow.

However, large batch sizes result in skyrocketing levels of WIP and high levels of variability in flow that cascade through the entire manufacturing plant. The result is long lead times and poor quality—if a problem is found in one body panel, the entire batch has to be scrapped.

One of the key lessons in Lean is that in order to shrink lead times and increase quality, we must strive to continually shrink batch sizes. The theoretical lower limit for batch size is *single-piece flow*, where each operation is performed one unit at a time.[‡]

The dramatic differences between large and small batch sizes can be seen in the simple newsletter mailing simulation described in *Lean Thinking: Banish Waste and Create Wealth in Your Corporation* by James P. Womack and Daniel T. Jones.

Suppose in our own example we have ten brochures to send and mailing each brochure requires four steps: fold the paper, insert the paper into the envelope, seal the envelope, and stamp the envelope.

The large batch strategy (i.e., “mass production”) would be to sequentially perform one operation on each of the ten brochures. In other words, we would first fold all ten sheets of paper, then insert each of them into envelopes, then seal all ten envelopes, and then stamp them.

On the other hand, in the small batch strategy (i.e., “single-piece flow”), all the steps required to complete each brochure are performed sequentially before starting on the next brochure. In other words, we fold one sheet of paper, insert it into the envelope, seal it, and stamp it—only then do we start the process over with the next sheet of paper.

The difference between using large and small batch sizes is dramatic (see figure 7). Suppose each of the four operations takes ten seconds for each of the ten envelopes. With the large batch size strategy, the first completed and stamped envelope is produced only after 310 seconds.

Worse, suppose we discover during the envelope sealing operation that we made an error in the first step of folding—in this case, the earliest we would discover the error is at two hundred seconds, and we have to refold and reinsert all ten brochures in our batch again.

‡ Also known as “batch size of one” or “1x1 flow,” terms that refer to batch size and a WIP limit of one.

Large Batches



Single-Piece Flow



Figure 7: Simulation of “envelope game” (fold, insert, seal, and stamp the envelope)

(Source: Stefan Luyten, “Single Piece Flow: Why mass production isn’t the most efficient way of doing ‘stuff,’” Medium.com, August 8, 2014, <https://medium.com/@stefanluyten/single-piece-flow-5d2c2bec845b#907517415c>.)

In contrast, in the small batch strategy the first completed stamped envelope is produced in only forty seconds, eight times faster than the large batch strategy. And, if we made an error in the first step, we only have to redo the one brochure in our batch. Small batch sizes result in less WIP, faster lead times, faster detection of errors, and less rework.

The negative outcomes associated with large batch sizes are just as relevant to the technology value stream as in manufacturing. Consider when we have an annual schedule for software releases, where an entire year’s worth of code that Development has worked on is released to production deployment.

Like in manufacturing, this large batch release creates sudden, high levels of WIP and massive disruptions to all downstream work centers, resulting in poor flow and poor quality outcomes. This validates our common experience that the larger the change going into production, the more difficult the production errors are to diagnose and fix, and the longer they take to remediate.

In a post on *Startup Lessons Learned*, Eric Ries states, “The batch size is the unit at which work-products move between stages in a development [or DevOps] process. For software, the easiest batch to see is code. Every time an engineer checks in code, they are batching up a certain amount of work. There are many techniques for controlling these batches, ranging from the tiny batches needed for continuous deployment to more traditional branch-based development, where all of the code from multiple developers working for weeks or months is batched up and integrated together.”

The equivalent to single piece flow in the technology value stream is realized with continuous deployment, where each change committed to version control is integrated, tested, and deployed into production. The practices that enable this are described in Part IV.

REDUCE THE NUMBER OF HANDOFFS

In the technology value stream, whenever we have long deployment lead times measured in months, it is often because there are hundreds (or even thousands) of operations required to move our code from version control into the production environment. To transmit code through the value stream requires multiple departments to work on a variety of tasks, including functional testing, integration testing, environment creation, server administration, storage administration, networking, load balancing, and information security.

Each time the work passes from team to team, we require all sorts of communication: requesting, specifying, signaling, coordinating, and often prioritizing, scheduling, deconflicting, testing, and verifying. This may require using different ticketing or project management systems; writing technical specification documents; communicating via meetings, emails, or phone calls; and using file system shares, FTP servers, and Wiki pages.

Each of these steps is a potential queue where work will wait when we rely on resources that are shared between different value streams (e.g., centralized operations). The lead times for these requests are often so long that there is constant escalation to have work performed within the needed timelines.

Even under the best circumstances, some knowledge is inevitably lost with each handoff. With enough handoffs, the work can completely lose the context of the problem being solved or the organizational goal being supported. For instance, a server administrator may see a newly created ticket requesting that user accounts be created, without knowing what application or service it's for, why it needs to be created, what all the dependencies are, or whether it's actually recurring work.

To mitigate these types of problems, we strive to reduce the number of handoffs, either by automating significant portions of the work or by reorganizing teams so they can deliver value to the customer themselves, instead of having to be constantly dependent on others. As a result, we increase flow by reducing the amount of time that our work spends waiting in queue, as well as the amount of non-value-added time. (See Appendix 4.)

CONTINUALLY IDENTIFY AND ELEVATE OUR CONSTRAINTS

To reduce lead times and increase throughput, we need to continually identify our system's constraints and improve its work capacity. In *Beyond the Goal*,

Dr. Goldratt states, “In any value stream, there is always a direction of flow, and there is always one and only constraint; any improvement not made at that constraint is an illusion.” If we improve a work center that is positioned before the constraint, work will merely pile up at the bottleneck even faster, waiting for work to be performed by the bottlenecked work center.

On the other hand, if we improve a work center positioned *after* the bottleneck, it remains starved, waiting for work to clear the bottleneck. As a solution, Dr. Goldratt defined the “five focusing steps:”

- Identify the system’s constraint.
- Decide how to exploit the system’s constraint.
- Subordinate everything else to the above decisions.
- Elevate the system’s constraint.
- If in the previous steps a constraint has been broken, go back to step one, but do not allow inertia to cause a system constraint.

In typical DevOps transformations, as we progress from deployment lead times measured in months or quarters to lead times measured in minutes, the constraint usually follows this progression:

- **Environment creation:** We cannot achieve deployments on-demand if we always have to wait weeks or months for production or test environments. The countermeasure is to create environments that are on demand and completely self-serviced, so that they are always available when we need them.
- **Code deployment:** We cannot achieve deployments on demand if each of our production code deployments take weeks or months to perform (i.e., each deployment requires 1,300 manual, error-prone steps, involving up to three hundred engineers). The countermeasure is to automate our deployments as much as possible, with the goal of being completely automated so they can be done self-service by any developer.
- **Test setup and run:** We cannot achieve deployments on demand if every code deployment requires two weeks to set up our test environments and data sets, and another four weeks to manually

execute all our regression tests. The countermeasure is to automate our tests so we can execute deployments safely and to parallelize them so the test rate can keep up with our code development rate.

- **Overly tight architecture:** We cannot achieve deployments on demand if overly tight architecture means that every time we want to make a code change we have to send our engineers to scores of committee meetings in order to get permission to make our changes. Our countermeasure is to create more loosely coupled architecture so that changes can be made safely and with more autonomy, increasing developer productivity.

After all these constraints have been broken, our constraint will likely be Development or the product owners. Because our goal is to enable small teams of developers to independently develop, test, and deploy value to customers quickly and reliably, this is where we want our constraint to be. High performers, regardless of whether an engineer is in Development, QA, Ops, or Infosec, state that their goal is to help maximize developer productivity.

When the constraint is here, we are limited only by the number of good business hypotheses we create and our ability to develop the code necessary to test these hypotheses with real customers.

The progression of constraints listed above are generalizations of typical transformations—techniques to identify the constraint in actual value streams, such as through value stream mapping and measurements, are described later in this book.

ELIMINATE HARDSHIPS AND WASTE IN THE VALUE STREAM

Shigeo Shingo, one of the pioneers of the Toyota Production System, believed that waste constituted the largest threat to business viability—the commonly used definition in Lean is “the use of any material or resource beyond what the customer requires and is willing to pay for.” He defined seven major types of manufacturing waste: inventory, overproduction, extra processing, transportation, waiting, motion, and defects.

More modern interpretations of Lean have noted that “eliminating waste” can have a demeaning and dehumanizing context; instead, the goal is reframed

to reduce hardship and drudgery in our daily work through continual learning in order to achieve the organization's goals. For the remainder of this book, the term *waste* will imply this more modern definition, as it more closely matches the DevOps ideals and desired outcomes.

In the book *Implementing Lean Software Development: From Concept to Cash*, Mary and Tom Poppendieck describe waste and hardship in the software development stream as anything that causes delay for the customer, such as activities that can be bypassed without affecting the result.

The following categories of waste and hardship come from *Implementing Lean Software Development* unless otherwise noted:

- **Partially done work:** This includes any work in the value stream that has not been completed (e.g., requirement documents or change orders not yet reviewed) and work that is sitting in queue (e.g., waiting for QA review or server admin ticket). Partially done work becomes obsolete and loses value as time progresses.
- **Extra processes:** Any additional work that is being performed in a process that does not add value to the customer. This may include documentation not used in a downstream work center, or reviews or approvals that do not add value to the output. Extra processes add effort and increase lead times.
- **Extra features:** Features built into the service that are not needed by the organization or the customer (e.g., “gold plating”). Extra features add complexity and effort to testing and managing functionality.
- **Task switching:** When people are assigned to multiple projects and value streams, requiring them to context switch and manage dependencies between work, adding additional effort and time into the value stream.
- **Waiting:** Any delays between work requiring resources to wait until they can complete the current work. Delays increase cycle time and prevent the customer from getting value.
- **Motion:** The amount of effort to move information or materials from one work center to another. Motion waste can be created when people who need to communicate frequently are not

colocated. Handoffs also create motion waste and often require additional communication to resolve ambiguities.

- **Defects:** Incorrect, missing, or unclear information, materials, or products create waste, as effort is needed to resolve these issues. The longer the time between defect creation and defect detection, the more difficult it is to resolve the defect.
- **Nonstandard or manual work:** Reliance on nonstandard or manual work from others, such as using non-rebuilding servers, test environments, and configurations. Ideally, any dependencies on Operations should be automated, self-serviced, and available on demand.
- **Heroics:** In order for an organization to achieve goals, individuals and teams are put in a position where they must perform unreasonable acts, which may even become a part of their daily work (e.g., nightly 2:00 a.m. problems in production, creating hundreds of work tickets as part of every software release).[†]

Our goal is to make these wastes and hardships—anywhere heroics become necessary—visible, and to systematically do what is needed to alleviate or eliminate these burdens and hardships to achieve our goal of fast flow.

CONCLUSION

Improving flow through the technology value stream is essential to achieving DevOps outcomes. We do this by making work visible, limiting WIP, reducing batch sizes and the number of handoffs, continually identifying and evaluating our constraints, and eliminating hardships in our daily work.

The specific practices that enable fast flow in the DevOps value stream are presented in Part IV. In the next chapter, we present The Second Way: The Principles of Feedback.

[†] Although heroics is not included in the Poppendieck categories of waste, it is included here because of how often it occurs, especially in Operation shared services.

3

The Second Way: *The Principles of Feedback*

While the First Way describes the principles that enable the fast flow of work from left to right, the Second Way describes the principles that enable the reciprocal fast and constant feedback from right to left at all stages of the value stream. Our goal is to create an ever safer and more resilient system of work.

This is especially important when working in complex systems, when the earliest opportunity to detect and correct errors is typically when a catastrophic event is underway, such as a manufacturing worker being hurt on the job or a nuclear reactor meltdown in progress.

In technology, our work happens almost entirely within complex systems with a high risk of catastrophic consequences. As in manufacturing, we often discover problems only when large failures are underway, such as a massive production outage or a security breach resulting in the theft of customer data.

We make our system of work safer by creating fast, frequent, high quality information flow throughout our value stream and our organization, which includes feedback and feedforward loops. This allows us to detect and remediate problems while they are smaller, cheaper, and easier to fix; avert problems before they cause catastrophe; and create organizational learning that we integrate into future work. When failures and accidents occur, we treat them as opportunities for learning, as opposed to a cause for punishment and blame. To achieve all of the above, let us first explore the nature of complex systems and how they can be made safer.

WORKING SAFELY WITHIN COMPLEX SYSTEMS

One of the defining characteristics of a complex system is that it defies any single person's ability to see the system as a whole and understand how all

the pieces fit together. Complex systems typically have a high degree of interconnectedness of tightly coupled components, and system-level behavior cannot be explained merely in terms of the behavior of the system components.

Dr. Charles Perrow studied the Three Mile Island crisis and observed that it was impossible for anyone to understand how the reactor would behave in all circumstances and how it might fail. When a problem was underway in one component, it was difficult to isolate from the other components, quickly flowing through the paths of least resistance in unpredictable ways.

Dr. Sidney Dekker, who also codified some of the key elements of safety culture, observed another characteristic of complex systems: doing the same thing twice will not predictably or necessarily lead to the same result. It is this characteristic that makes static checklists and best practices, while valuable, insufficient to prevent catastrophes from occurring. (See Appendix 5.)

Therefore, because failure is inherent and inevitable in complex systems, we must design a safe system of work, whether in manufacturing or technology, where we can perform work without fear, confident that any errors will be detected quickly, long before they cause catastrophic outcomes, such as worker injury, product defects, or negative customer impact.

After he decoded the causal mechanism behind the Toyota Product System as part of his doctoral thesis at Harvard Business School, Dr. Steven Spear stated that designing perfectly safe systems is likely beyond our abilities, but we can make it safer to work in complex systems when the four following conditions are met:[†]

- Complex work is managed so that problems in design and operations are revealed
- Problems are swarmed and solved, resulting in quick construction of new knowledge
- New local knowledge is exploited globally throughout the organization
- Leaders create other leaders who continually grow these types of capabilities

[†] Dr. Spear extended his work to explain the long-lasting successes of other organizations, such as the Toyota supplier network, Alcoa, and the US Navy's Nuclear Power Propulsion Program.

Each of these capabilities are required to work safely in a complex system. In the next sections, the first two capabilities and their importance are described, as well as how they have been created in other domains and what practices enable them in the technology value stream. (The third and fourth capabilities are described in chapter 4.)

SEE PROBLEMS AS THEY OCCUR

In a safe system of work, we must constantly test our design and operating assumptions. Our goal is to increase information flow in our system from as many areas as possible, sooner, faster, cheaper, and with as much clarity between cause and effect as possible. The more assumptions we can invalidate, the faster we can find and fix problems, increasing our resilience, agility, and ability to learn and innovate.

We do this by creating feedback and feedforward loops into our system of work. Dr. Peter Senge in his book *The Fifth Discipline: The Art & Practice of the Learning Organization* described feedback loops as a critical part of learning organizations and systems thinking. Feedback and feedforward loops cause components within a system to reinforce or counteract each other.

In manufacturing, the absence of effective feedback often contribute to major quality and safety problems. In one well-documented case at the General Motors Fremont manufacturing plant, there were no effective procedures in place to detect problems during the assembly process, nor were there explicit procedures on what to do when problems were found. As a result, there were instances of engines being put in backward, cars missing steering wheels or tires, and cars even having to be towed off the assembly line because they wouldn't start.

In contrast, in high performing manufacturing operations there is fast, frequent, and high quality information flow throughout the entire value stream—every work operation is measured and monitored, and any defects or significant deviations are quickly found and acted upon. These are the foundation of what enables quality, safety, and continual learning and improvement.

In the technology value stream, we often get poor outcomes because of the absence of fast feedback. For instance, in a waterfall software project, we may develop code for an entire year and get no feedback on quality until we begin the testing phase—or worse, when we release our software to customers.

When feedback is this delayed and infrequent, it is too slow to enable us to prevent undesirable outcomes.

In contrast, our goal is to create fast feedback and fastforward loops wherever work is performed, at all stages of the technology value stream, encompassing Product Management, Development, QA, Infosec, and Operations. This includes the creation of automated build, integration, and test processes, so that we can immediately detect when a change has been introduced that takes us out of a correctly functioning and deployable state.

We also create pervasive telemetry so we can see how all our system components are operating in the production environment, so that we can quickly detect when they are not operating as expected. Telemetry also allows us to measure whether we are achieving our intended goals and, ideally, is radiated to the entire value stream so we can see how our actions affect other portions of the system as a whole.

Feedback loops not only enable quick detection and recovery of problems, but they also inform us on how to prevent these problems from occurring again in the future. Doing this increases the quality and safety of our system of work, and creates organizational learning.

As Elisabeth Hendrickson, VP of Engineering at Pivotal Software, Inc. and author of *Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing*, said, “When I headed up quality engineering, I described my job as ‘creating feedback cycles.’ Feedback is critical because it is what allows us to steer. We must constantly validate between customer needs, our intentions and our implementations. Testing is merely one sort of feedback.”

SWARM AND SOLVE PROBLEMS TO BUILD NEW KNOWLEDGE

Obviously, it is not sufficient to merely detect when the unexpected occurs. When problems occur, we must swarm them, mobilizing whoever is required to solve the problem.

According to Dr. Spear, the goal of swarming is to contain problems before they have a chance to spread, and to diagnose and treat the problem so that it cannot recur. “In doing so,” he says, “they build ever-deeper knowledge about how to manage the systems for doing our work, converting inevitable up-front ignorance into knowledge.”

The paragon of this principle is the Toyota *Andon cord*. In a Toyota manufacturing plant, above every work center is a cord that every worker and manager is trained to pull when something goes wrong; for example, when a part is defective, when a required part is not available, or even when work takes longer than documented.[†]

When the *Andon cord* is pulled, the team leader is alerted and immediately works to resolve the problem. If the problem cannot be resolved within a specified time (e.g., fifty-five seconds), the production line is halted so that the entire organization can be mobilized to assist with problem resolution until a successful countermeasure has been developed.

Instead of working around the problem or scheduling a fix “when we have more time,” we swarm to fix it immediately—this is nearly the opposite of the behavior at the GM Fremont plant described earlier. Swarming is necessary for the following reasons:

- It prevents the problem from progressing downstream, where the cost and effort to repair it increases exponentially and technical debt is allowed to accumulate.
- It prevents the work center from starting new work, which will likely introduce new errors into the system.
- If the problem is not addressed, the work center could potentially have the same problem in the next operation (e.g., fifty-five seconds later), requiring more fixes and work. (See Appendix 6.)

This practice of swarming seems contrary to common management practice, as we are deliberately allowing a local problem to disrupt operations globally. However, swarming enables learning. It prevents the loss of critical information due to fading memories or changing circumstances. This is especially critical in complex systems, where many problems occur because of some unexpected, idiosyncratic interaction of people, processes, products, places, and circumstances—as time passes, it becomes impossible to reconstruct exactly what was going on when the problem occurred.

As Dr. Spear notes, swarming is part of the “disciplined cycle of real-time problem recognition, diagnosis,...and treatment (countermeasures or corrective measures in manufacturing vernacular). It [is] the discipline of the

† In some of its plants, Toyota has moved to using an *Andon* button.

Shewhart cycle—plan, do, check, act—popularized by W. Edwards Deming, but accelerated to warp speed.”

It is only through the swarming of ever smaller problems discovered ever earlier in the life cycle that we can deflect problems before a catastrophe occurs. In other words, when the nuclear reactor melts down, it is already too late to avert worst outcomes.

To enable fast feedback in the technology value stream, we must create the equivalent of an Andon cord and the related swarming response. This requires that we also create the culture that makes it safe, and even encouraged, to pull the Andon cord when something goes wrong, whether it is when a production incident occurs or when errors occur earlier in the value stream, such as when someone introduces a change that breaks our continuous build or test processes.

When conditions trigger an Andon cord pull, we swarm to solve the problem and prevent the introduction of new work until the issue has been resolved.[†] This provides fast feedback for everyone in the value stream (especially the person who caused the system to fail), enables us to quickly isolate and diagnose the problem, and prevents further complicating factors that can obscure cause and effect.

Preventing the introduction of new work enables continuous integration and deployment, which is single-piece flow in the technology value stream. All changes that pass our continuous build and integration tests are deployed into production, and any changes that cause any tests to fail trigger our Andon cord and are swarmed until resolved.

KEEP PUSHING QUALITY CLOSER TO THE SOURCE

We may inadvertently perpetuate unsafe systems of work due to the way we respond to accidents and incidents. In complex systems, adding more inspection steps and approval processes actually increases the likelihood of future failures. The effectiveness of approval processes decreases as we push decision-making further away from where the work is performed. Doing so not only lowers the quality of decisions but also increases our cycle time, thus decreasing

[†] Astonishingly, when the number of Andon cord pulls drop, plant managers will actually decrease the tolerances to get an increase in the number of Andon cord pulls in order to continue to enable more learnings and improvements and to detect ever-weaker failure signals.

the strength of the feedback between cause and effect, and reducing our ability to learn from successes and failures.[‡]

This can be seen even in smaller and less complex systems. When top-down, bureaucratic command and control systems become ineffective, it is usually because the variance between “who should do something” and “who is actually doing something” is too large, due to insufficient clarity and timeliness.

Examples of ineffective quality controls include:

- Requiring another team to complete tedious, error-prone, and manual tasks that could be easily automated and run as needed by the team who needs the work performed
- Requiring approvals from busy people who are distant from the work, forcing them to make decisions without an adequate knowledge of the work or the potential implications, or to merely rubber stamp their approvals
- Creating large volumes of documentation of questionable detail which become obsolete shortly after they are written
- Pushing large batches of work to teams and special committees for approval and processing and then waiting for responses

Instead, we need everyone in our value stream to find and fix problems in their area of control as part of our daily work. By doing this, we push quality and safety responsibilities and decision-making to where the work is performed, instead of relying on approvals from distant executives.

We use peer reviews of our proposed changes to gain whatever assurance is needed that our changes will operate as designed. We automate as much of the quality checking typically performed by a QA or Information Security department as possible. Instead of developers needing to request or schedule

‡ In the 1700s, the British government engaged in a spectacular example of top-down, bureaucratic command and control, which proved remarkably ineffective. At the time, Georgia was still a colony, and despite the fact that the British government was three thousand miles away and lacked firsthand knowledge of local land chemistry, rockiness, topography, accessibility to water, and other conditions, it tried to plan Georgia's entire agricultural economy. The results of the attempt were dismal and left Georgia with the lowest levels of prosperity and population in the thirteen colonies.

a test to be run, these tests can be performed on demand, enabling developers to quickly test their own code and even deploy those changes into production themselves.

By doing this, we truly make quality everyone's responsibility as opposed to it being the sole responsibility of a separate department. Information security is not just Information Security's job, just as availability isn't merely the job of Operations.

Having developers share responsibility for the quality of the systems they build not only improves outcomes but also accelerates learning. This is especially important for developers as they are typically the team that is furthest removed from the customer. As Gary Gruver observes, "It's impossible for a developer to learn anything when someone yells at them for something they broke six months ago—that's why we need to provide feedback to everyone as quickly as possible, in minutes, not months."

ENABLE OPTIMIZING FOR DOWNSTREAM WORK CENTERS

In the 1980s, Designing for Manufacturability principles sought to design parts and processes so that finished goods could be created with the lowest cost, highest quality, and fastest flow. Examples include designing parts that are wildly asymmetrical to prevent them from being put on backwards, and designing screw fasteners so that they are impossible to over-tighten.

This was a departure from how design was typically done, which focused on the external customers but overlooked internal stakeholders, such as the people performing the manufacturing.

Lean defines two types of customers that we must design for: the external customer (who most likely pays for the service we are delivering) and the internal customer (who receives and processes the work immediately after us). According to Lean, our most important customer is our next step downstream. Optimizing our work for them requires that we have empathy for their problems in order to better identify the design problems that prevent fast and smooth flow.

In the technology value stream, we optimize for downstream work centers by designing for operations, where operational non-functional requirements (e.g., architecture, performance, stability, testability, configurability, and security) are prioritized as highly as user features.

By doing this, we create quality at the source, likely resulting in a set of codified non-functional requirements that we can proactively integrate into every service we build.

CONCLUSION

Creating fast feedback is critical to achieving quality, reliability, and safety in the technology value stream. We do this by seeing problems as they occur, swarming and solving problems to build new knowledge, pushing quality closer to the source, and continually optimizing for downstream work centers.

The specific practices that enable fast flow in the DevOps value stream are presented in Part IV. In the next chapter, we present the Third Way, the Principles of Feedback

4

The Third Way: *The Principles of Continual Learning and Experimentation*

While the First Way addresses work flow from left to right and the Second Way addresses the reciprocal fast and constant feedback from right to left, the Third Way focuses on creating a culture of continual learning and experimentation. These are the principles that enable constant creation of individual knowledge, which is then turned into team and organizational knowledge.

In manufacturing operations with systemic quality and safety problems, work is typically rigidly defined and enforced. For instance, in the GM Fremont plant described in the previous chapter, workers had little ability to integrate improvements and learnings into their daily work, with suggestions for improvement “apt to meet a brick wall of indifference.”

In these environments, there is also often a culture of fear and low trust, where workers who make mistakes are punished, and those who make suggestions or point out problems are viewed as whistle-blowers and troublemakers. When this occurs, leadership is actively suppressing, even punishing, learning and improvement, perpetuating quality and safety problems.

In contrast, high-performing manufacturing operations require and actively promote learning—instead of work being rigidly defined, the system of work is dynamic, with line workers performing experiments in their daily work to generate new improvements, enabled by rigorous standardization of work procedures and documentation of the results.

In the technology value stream, our goal is to create a high-trust culture, reinforcing that we are all lifelong learners who must take risks in our daily work. By applying a scientific approach to both process improvement and product development, we learn from our successes and failures, identifying

which ideas don't work and reinforcing those that do. Moreover, any local learnings are rapidly turned into global improvements, so that new techniques and practices can be used by the entire organization.

We reserve time for the improvement of daily work and to further accelerate and ensure learning. We consistently introduce stress into our systems to force continual improvement. We even simulate and inject failures in our production services under controlled conditions to increase our resilience.

By creating this continual and dynamic system of learning, we enable teams to rapidly and automatically adapt to an ever-changing environment, which ultimately helps us win in the marketplace.

ENABLING ORGANIZATIONAL LEARNING AND A SAFETY CULTURE

When we work within a complex system, by definition it is impossible for us to perfectly predict all the outcomes for any action we take. This is what contributes to unexpected, or even catastrophic, outcomes and accidents in our daily work, even when we take precautions and work carefully.

When these accidents affect our customers, we seek to understand why it happened. The root cause is often deemed to be human error, and the all too common management response is to “name, blame, and shame” the person who caused the problem.[†] And, either subtly or explicitly, management hints that the person guilty of committing the error will be punished. They then create more processes and approvals to prevent the error from happening again.

Dr. Sidney Dekker, who codified some of the key elements of safety culture and coined the term *just culture*, wrote, “Responses to incidents and accidents that are seen as unjust can impede safety investigations, promote fear rather than mindfulness in people who do safety-critical work, make organizations more bureaucratic rather than more careful, and cultivate professional secrecy, evasion, and self-protection.”

These issues are especially problematic in the technology value stream—our work is almost always performed within a complex system, and how management chooses to react to failures and accidents leads to a culture of fear,

[†] The “name, blame, shame” pattern is part of the Bad Apple Theory criticized by Dr. Sydney Dekker and extensively discussed in his book *The Field Guide to Understanding Human Error*.

which then makes it unlikely that problems and failure signals are ever reported. The result is that problems remain hidden until a catastrophe occurs.

Dr. Ron Westrum was one of the first to observe the importance of organizational culture on safety and performance. He observed that in healthcare organizations, the presence of “generative” cultures was one of the top predictors of patient safety. Dr. Westrum defined three types of culture:

- Pathological organizations are characterized by large amounts of fear and threat. People often hoard information, withhold it for political reasons, or distort it to make themselves look better. Failure is often hidden.
- Bureaucratic organizations are characterized by rules and processes, often to help individual departments maintain their “turf.” Failure is processed through a system of judgment, resulting in either punishment or justice and mercy.
- Generative organizations are characterized by actively seeking and sharing information to better enable the organization to achieve its mission. Responsibilities are shared throughout the value stream, and failure results in reflection and genuine inquiry.

Pathological	Bureaucratic	Generative
Information is hidden	Information may be ignored	Information is actively sought
Messengers are “shot”	Messengers are tolerated	Messengers are trained
Responsibilities are shirked	Responsibilities are compartmented	Responsibilities are shared
Bridging between teams is discouraged	Bridging between teams is allowed but discouraged	Bridging between teams is rewarded
Failure is covered up	Organizatoin is just and merciful	Failure causes inquiry
New ideas are crushed	New ideas create problems	New ideas are weclomed

Figure 8: The Westrum organizational typology model: how organizations process information (Source: Ron Westrum, “A typology of organisation culture,” *BMJ Quality & Safety* 13, no. 2 (2004), doi:10.1136/qshc.2003.009522.)

Just as Dr. Westrum found in healthcare organizations, a high-trust, generative culture also predicted IT and organizational performance in technology value streams.

In the technology value stream, we establish the foundations of a generative culture by striving to create a safe system of work. When accidents and failures occur, instead of looking for human error, we look for how we can redesign the system to prevent the accident from happening again.

For instance, we may conduct a blameless post-mortem after every incident to gain the best understanding of how the accident occurred and agree upon what the best countermeasures are to improve the system, ideally preventing the problem from occurring again and enabling faster detection and recovery.

By doing this, we create organizational learning. As Bethany Macri, an engineer at Etsy who led the creation of the Morgue tool to help with recording of post-mortems, stated, “By removing blame, you remove fear; by removing fear, you enable honesty; and honesty enables prevention.”

Dr. Spear observes that the result of removing blame and putting organizational learning in its place is that “organizations become ever more self-diagnosing and self-improving, skilled at detecting problems [and] solving them.”

Many of these attributes were also described by Dr. Senge as attributes of learning organizations. In *The Fifth Discipline*, he wrote that these characteristics help customers, ensure quality, create competitive advantage and an energized and committed workforce, and uncover the truth.

INSTITUTIONALIZE THE IMPROVEMENT OF DAILY WORK

Teams are often not able or not willing to improve the processes they operate within. The result is not only that they continue to suffer from their current problems, but their suffering also grows worse over time. Mike Rother observed in *Toyota Kata* that in the absence of improvements, processes don't stay the same—due to chaos and entropy, processes actually degrade over time.

In the technology value stream, when we avoid fixing our problems, relying on daily workarounds, our problems and technical debt accumulates until all we are doing is performing workarounds, trying to avoid disaster, with no cycles leftover for doing productive work. This is why Mike Orzen, author of

Lean IT, observed, “Even more important than daily work is the improvement of daily work.”

We improve daily work by explicitly reserving time to pay down technical debt, fix defects, and refactor and improve problematic areas of our code and environments—we do this by reserving cycles in each development interval, or by scheduling *kaizen blitzes*, which are periods when engineers self-organize into teams to work on fixing any problem they want.

The result of these practices is that everyone finds and fixes problems in their area of control, all the time, as part of their daily work. When we finally fix the daily problems that we’ve worked around for months (or years), we can eradicate from our system the less obvious problems. By detecting and responding to these ever-weaker failure signals, we fix problems when it is not only easier and cheaper but also when the consequences are smaller.

Consider the following example that improved workplace safety at Alcoa, an aluminum manufacturer with \$7.8 billion in revenue in 1987. Aluminum manufacturing requires extremely high heat, high pressures, and corrosive chemicals. In 1987, Alcoa had a frightening safety record, with 2% of the ninety thousand employee workforce being injured each year—that’s seven injuries per day. When Paul O’Neill started as CEO, his first goal was to have zero injuries to employees, contractors, and visitors.

O’Neill wanted to be notified within twenty-four hours of anyone being injured on the job—not to punish, but to ensure and promote that learnings were being generated and incorporated to create a safer workplace. Over the course of ten years, Alcoa reduced their injury rate by 95%.

The reduction in injury rates allowed Alcoa to focus on smaller problems and weaker failure signals—instead of notifying O’Neill only when injuries occurred, they started reporting any close calls as well.[†] By doing this, they improved workplace safety over the subsequent twenty years and have one of the most enviable safety records in the industry.

As Dr. Spear writes, “Alcoans gradually stopped working around the difficulties, inconveniences, and impediments they experienced. Coping, fire fighting, and making do were gradually replaced throughout the organization by a

[†] It is astonishing, instructional, and truly moving to see the level of conviction and passion that Paul O’Neill has about the moral responsibility leaders have to create workplace safety.

dynamic of identifying opportunities for process and product improvement. As those opportunities were identified and the problems were investigated, the pockets of ignorance that they reflected were converted into nuggets of knowledge.” This helped give the company a greater competitive advantage in the market.

Similarly, in the technology value stream, as we make our system of work safer, we find and fix problems from ever weaker failure signals. For example, we may initially perform blameless post-mortems only for customer-impacting incidents. Over time, we may perform them for lesser team-impacting incidents and near misses as well.

TRANSFORM LOCAL DISCOVERIES INTO GLOBAL IMPROVEMENTS

When new learnings are discovered locally, there must also be some mechanism to enable the rest of the organization to use and benefit from that knowledge. In other words, when teams or individuals have experiences that create expertise, our goal is to convert that tacit knowledge (i.e., knowledge that is difficult to transfer to another person by means of writing it down or verbalizing) into explicit, codified knowledge, which becomes someone else’s expertise through practice.

This ensures that when anyone else does similar work, they do so with the cumulative and collective experience of everyone in the organization who has ever done the same work. A remarkable example of turning local knowledge into global knowledge is the US Navy’s Nuclear Power Propulsion Program (also known as “NR” for “Naval Reactors”), which has over 5,700 reactor-years of operation without a single reactor-related casualty or escape of radiation.

The NR is known for their intense commitment to scripted procedures and standardized work, and the need for incident reports for any departure from procedure or normal operations to accumulate learnings, no matter how minor the failure signal—they constantly update procedures and system designs based on these learnings.

The result is that when a new crew sets out to sea on their first deployment, they and their officers benefit from the collective knowledge of 5,700 accident-free reactor-years. Equally impressive is that their own experiences at

sea will be added to this collective knowledge, helping future crews safely achieve their own missions.

In the technology value stream, we must create similar mechanisms to create global knowledge, such as making all our blameless post-mortem reports searchable by teams trying to solve similar problems, and by creating shared source code repositories that span the entire organization, where shared code, libraries, and configurations that embody the best collective knowledge of the entire organization can be easily utilized. All these mechanisms help convert individual expertise into artifacts that the rest of the organization can use.

INJECT RESILIENCE PATTERNS INTO OUR DAILY WORK

Lower performing manufacturing organizations buffer themselves from disruptions in many ways—in other words, they bulk up or add flab. For instance, to reduce the risk of a work center being idle (due to inventory arriving late, inventory that had to be scrapped, etc.), managers may choose to stockpile more inventory at each work center. However, that inventory buffer also increases WIP, which has all sorts of undesired outcomes, as previously discussed.

Similarly, to reduce the risk of a work center going down due to machinery failure, managers may increase capacity by buying more capital equipment, hiring more people, or even increasing floor space. All these options increase costs.

In contrast, high performers achieve the same results (or better) by improving daily operations, continually introducing tension to elevate performance, as well as engineering more resilience into their system.

Consider a typical experiment at one of Aisin Seiki Global's mattress factories, one of Toyota's top suppliers. Suppose they had two production lines, each capable of producing one hundred units per day. On slow days, they would send all production onto one line, experimenting with ways to increase capacity and identify vulnerabilities in their process, knowing that if overloading the line caused it to fail, they could send all production to the second line.

By relentless and constant experimentation in their daily work, they were able to continually increase capacity, often without adding any new equipment

or hiring more people. The emergent pattern that results from these types of improvement rituals not only improves performance but also improves resilience, because the organization is always in a state of tension and change. This process of applying stress to increase resilience was named *antifragility* by author and risk analyst Nassim Nicholas Taleb.

In the technology value stream, we can introduce the same type of tension into our systems by seeking to always reduce deployment lead times, increase test coverage, decrease test execution times, and even by re-architecting if necessary to increase developer productivity or increase reliability.

We may also perform *game day* exercises, where we rehearse large scale failures, such as turning off entire data centers. Or we may inject ever-larger scale faults into the production environment (such as the famous Netflix “Chaos Monkey” which randomly kills processes and compute servers in production) to ensure that we’re as resilient as we want to be.

LEADERS REINFORCE A LEARNING CULTURE

Traditionally, leaders were expected to be responsible for setting objectives, allocating resources for achieving those objectives, and establishing the right combination of incentives. Leaders also establish the emotional tone for the organizations they lead. In other words, leaders lead by “making all the right decisions.”

However, there is significant evidence that shows greatness is not achieved by leaders making all the right decisions—instead, the leader’s role is to create the conditions so their team can discover greatness in their daily work. In other words, creating greatness requires both leaders and workers, each of whom are mutually dependent upon each other.

Jim Womack, author of *Gemba Walks*, described the complementary working relationship and mutual respect that must occur between leaders and frontline workers. According to Womack, this relationship is necessary because neither can solve problems alone—leaders are not close enough to the work, which is required to solve any problem, and frontline workers do not have the broader organizational context or the authority to make changes outside of their area of work.[†]

[†] Leaders are responsible for the design and operation of processes at a higher level of aggregation where others have less perspective and authority.

Leaders must elevate the value of learning and disciplined problem solving. Mike Rother formalized these methods in what he calls the *coaching kata*. The result is one that mirrors the scientific method, where we explicitly state our True North goals, such as “sustain zero accidents” in the case of Alcoa, or “double throughput within a year” in the case of Aisin.

These strategic goals then inform the creation of iterative, shorter term goals, which are cascaded and then executed by establishing target conditions at the value stream or work center level (e.g., “reduce lead time by 10% within the next two weeks”).

These target conditions frame the scientific experiment: we explicitly state the problem we are seeking to solve, our hypothesis of how our proposed countermeasure will solve it, our methods for testing that hypothesis, our interpretation of the results, and our use of learnings to inform the next iteration.

The leader helps coach the person conducting the experiment with questions that may include:

- What was your last step and what happened?
- What did you learn?
- What is your condition now?
- What is your next target condition?
- What obstacle are you working on now?
- What is your next step?
- What is your expected outcome?
- When can we check?

This problem-solving approach in which leaders help workers see and solve problems in their daily work is at the core of the Toyota Production System, of learning organizations, the Improvement Kata, and high-reliability organizations. Mike Rother observes that he sees Toyota “as an organization defined primarily by the unique behavior routines it continually teaches to all its members.”

In the technology value stream, this scientific approach and iterative method guides all of our internal improvement processes, but also how we perform experiments to ensure that the products we build actually help our internal and external customers achieve their goals.

CONCLUSION

The principles of the Third Way address the need for valuing organizational learning, enabling high trust and boundary-spanning between functions, accepting that failures will always occur in complex systems, and making it acceptable to talk about problems so we can create a safe system of work. It also requires institutionalizing the improvement of daily work, converting local learnings into global learnings that can be used by the entire organization, as well as continually injecting tension into our daily work.

Although fostering a culture of continual learning and experimentation is the principle of the Third Way, it is also interwoven into the First and Second Ways. In other words, improving flow and feedback requires an iterative and scientific approach that includes framing of a target condition, stating a hypothesis of what will help us get there, designing and conducting experiments, and evaluating the results.

The results are not only better performance but also increased resilience, higher job satisfaction, and improved organization adaptability.

PART I CONCLUSION

In Part I of *The DevOps Handbook* we looked back at several movements in history that helped lead to the development of DevOps. We also looked at the three main principles that form the foundation for successful DevOps organizations: the principles of Flow, Feedback, and Continual Learning and Experimentation. In Part II, we will begin to look at how to start a DevOps movement in your organization.