

O'REILLY®



Effective DevOps

BUILDING A CULTURE OF COLLABORATION,
AFFINITY, AND TOOLING AT SCALE

Jennifer Davis & Ryn Daniels

Praise for *Effective DevOps*

“Excellent reading for anyone interested in understanding devops and how to foster a devops culture.”

—James Turnbull, CTO at Kickstarter

“Devops is not another “tech movement” that can be deferred and discarded; it has tentacles throughout the organization from concept to cash. Anyone who has watched their cost of building new features increase, or has lived through the frustrations of unplanned downtime, will understand the intrinsic values espoused by the devops movement.

Effective DevOps is the most comprehensive book I’ve seen on the topic; and one that is completely digestible and applicable for those within technology *and* the lines of business.

I particularly enjoyed the sections debunking common myths and memes regarding devops; such as it being cost prohibitive, only applicable to startups, or that it’s a position to be filled. Such resounding rebuttals help preserve the spirit of devops and keeps commoditization of the movement at bay. This book will become recommended reading for any organization that conceives, develops, and deploys software.”

—Nivia S. Henry, Sr. Agile Coach at Summa

“High performance organizations see technology as a strategic capability. However, they understand that technology is often not the most challenging dimension. Culture matters.

How well your organization collaborates, experiments and seeks to learn and share knowledge dictates the level it not only accepts but will achieve. In this book, Jennifer and Ryn have outlined the necessary conditions to create a system of work for your business to develop its own path towards continuous improvement, growth and transformation.

Read it, reflect on it, then deploy it.”

—Barry O’Reilly, Founder and CEO, ExecCamp and coauthor of *Lean Enterprise: How High Performance Organizations Innovate At Scale* (O’Reilly)

“Careful attention to the history and traditions of the devops movement meets detailed advice and empathy for today’s practitioners.”

—*Bridget Kromhout, Principal Technologist at Pivotal*

"*Effective DevOps* is a flat-out impressive collection of the bits and pieces that make up what is essentially the largest shift in technical work since the Agile Manifesto was published. It brings together stories and resources from diverse resources that feel worlds apart and presents everything in an approachable manner, and covers not just tools but the culture, behavior, and work pattern characteristics that are so essential for successful technology teams.”

—*Mandi Walls, Technical Community Manager at Chef and author of Building a DevOps Culture (O'Reilly)*

“Everyone knows culture is important in the context of devops, but we’ve lacked a book-length reference into this vital topic. *Effective DevOps* is a broad, deep inquiry into human factors that should be studied by every manager who wants to build high performing technical teams and organizations.”

—*Jez Humble, coauthor of Continuous Delivery (Addison-Wesley) and Lean Enterprise (O'Reilly)*

Effective DevOps

*Building a Culture of Collaboration,
Affinity, and Tooling at Scale*

This excerpt contains Chapters 1–6 of the book *Effective DevOps*. The complete book is available on Safari and through other retailers.

Jennifer Davis and Ryn Daniels

Effective DevOps

by Jennifer Davis and Ryn Daniels

Copyright © 2016 Jennifer Davis and Ryn Daniels. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com/safari>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Brian Anderson

Production Editor: Colleen Lobner

Copyeditor: Rachel Monaghan

Proofreader: Jasmine Kwitny

Indexer: WordCo Indexing Services

Interior Designer: David Futato

Cover Designer: Randy Comer

Illustrator: Rebecca Demarest

June 2016:

First Edition

Revision History for the First Edition

2016-05-23: First Release

2018-01-19: Second Release

2018-04-06: Third Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491926307> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Effective DevOps*, the cover image of a wild yak, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Microsoft. See our [statement of editorial independence](#).

978-1-492-04527-4

[LSI]

Table of Contents

Foreword by John Allspaw.....	ix
Foreword by Nicole Forsgren.....	xi
1. The Big Picture.....	13
A Snapshot of Devops Culture	13
The Evolution of Culture	15
The Value of the Story	17
Ryn's Story	18
Jennifer's Story	19
Illustrating Devops with Stories	21
2. What Is Devops?.....	23
A Prescription for Culture	23
The Devops Equation	23
Devops as Folk Model	24
The Old View and the New View	24
The Devops Compact	25
3. A History of Devops.....	29
Developer as Operator	29
The Advent of Software Engineering	30
The Advent of Proprietary Software and Standardization	31
The Age of the Network	32
The Beginnings of a Global Community	33
The Age of Applications and the Web	34
The Growth of Software Development Methodologies	35
Open Source Software, Proprietary Services	36

Agile Infrastructure	37
The Beginning of devopsdays	38
The Current State of Devops	39
Summary	39
4. Foundational Terminology and Concepts.....	41
Software Development Methodologies	41
Waterfall	42
Agile	43
Scrum	44
Operations Methodologies	44
ITIL	44
COBIT	45
Systems Methodologies	45
Lean	45
Development, Release, and Deployment Concepts	47
Version Control	47
Test-Driven Development	47
Application Deployment	48
Continuous Integration	48
Continuous Delivery	49
Continuous Deployment	49
Minimum Viable Product	49
Infrastructure Concepts	50
Configuration Management	50
Cloud Computing	51
Infrastructure Automation	51
Artifact Management	51
Containers	52
Cultural Concepts	52
Retrospective	52
Postmortem	53
Blamelessness	53
Organizational Learning	54
Summary	54
5. Devops Misconceptions and Anti-Patterns.....	55
Common Devops Misconceptions	55
Devops Only Involves Developers and System Administrators	55
Devops Is a Team	56
Devops Is a Job Title	56
Devops Is Relevant Only to Web Startups	57

You Need a Devops Certification	57
Devops Means Doing All the Work with Half the People	58
There Is One “Right Way” (or “Wrong Way”) to Do Devops	59
It Will Take X Weeks/Months to Implement Devops	59
Devops Is All About the Tools	60
Devops Is About Automation	60
Devops Is a Fad	62
Devops Anti-Patterns	63
Blame Culture	63
Silos	64
Root Cause Analysis	64
Human Error	65
Summary	65
6. The Four Pillars of Effective Devops.....	67
Collaboration	68
Affinity	68
Tools	68
Scaling	68

Foreword by John Allspaw

There is a sea change happening in software development and operations, and it is not simply the introduction of a new word into our lexicon—it’s much more than that. It is a fundamental shift of perspective in the design, construction, and operation of software in a world where almost every successful organization recognizes that software is not something you simply build and launch—it is something you operate.

What makes this shift unique is that it’s more encompassing, more holistic, and more reflective of the reality that engineering teams face on a daily basis. Long gone are the days that manufacturing and assembly-line metaphors could be used in software development and operations. Long gone are the days that products are things that are designed, planned, and then finally launched. There is no “finally” anymore. There is only an endless cycle of adaptation, change, and learning.

What Ryn and Jennifer have laid out in this volume is a myriad of threads to pull on as engineers in teams and organizations cope with the complexity that comes with trying to make their work “simple.”

Ryn and Jennifer do not paint a picture of one-size-fits-all or deterministic solutionism. Instead, they describe a landscape of topic areas, practices, and observations of teams and organizations that understand the idea that at the heart of good products, good user experiences, and good software is the elegantly messy world of human cooperation, thoughtful critique, effective collaboration, and judgment.

In 2009, my friend Paul Hammond and I gave a presentation at O’Reilly’s Velocity Conference entitled “10+ Deploys a Day: Dev and Ops Cooperation at Flickr.” While some of the material was about perspectives on continuous deployment, many chose to focus on the “10+ Deploys” part rather than the “Cooperation” part. I believe it is a mistake to think that the technology or “hard parts” can be seen as somehow distinct or separate from the social and cultural “soft parts”—they cannot. They are inextricably linked and equally critical to success. In other words, people and process influence tools and software in more ways than most are willing to admit or even know.

My strong advice to readers: do not make the mistake of thinking that technology is not about people and process, first and foremost. Your competition will eat you alive the second you start thinking this.

These topics of concern are not found in typical computer science curriculums, nor are they found in typical professional leadership and development courses. These topics are born out of the pragmatic maturity that only hard-won work in the field can produce.

Ryn and Jennifer have given you an in-depth set of signposts in this book. My genuine request to you, dear reader, is to use these signposts to think with, in your own context and environment.

—*John Allspaw*
Chief Technology Officer, Etsy
Brooklyn, NY

Foreword by Nicole Forsgren

In 2003, Nicholas Carr declared to the world that IT didn't matter—and because he said it in *Harvard Business Review*, organizations (and the executives that run them) believed him. Well, times have changed, and so has IT. Since 2009, the most innovative teams and organizations have shown that technology can play a key role in delivering real value and competitive advantage. This technology revolution is known as DevOps, and this book can show you how to join those innovative companies to deliver value with technology as well.

Here, Jennifer and Ryn draw on their experiences at innovative companies as well as their backgrounds as prominent experts in the community to highlight what it really takes to do DevOps—or as they call it, devops—effectively. This vantage point brings unique insights that can be applicable and useful to any reader, because they combine knowledge from several companies and across industries. This gives readers something to take away regardless of where you are in your journey, no matter how big (or small) your organization is.

The tales and advice found in *Effective DevOps* parallel those I've seen in my own work for the past decade. As a leading researcher in the field and the lead investigator on the *State of DevOps Reports*, I know that a strong organizational culture that prioritizes information flow and trust is a key component of any DevOps transformation, and the factor that sets the DevOps movement apart from traditional IT. Data I've collected from over 20,000 DevOps professionals also shows that this culture drives IT and organizational performance, helping the best IT organizations see double productivity, profitability, and market share when compared to their peers. It's no mistake that Jennifer and Ryn start the book with a discussion on aspects of culture, communication, and trust, and devote a good portion of their time addressing the importance of these factors to any transformative work. As technologists, we love to start with the tools and maybe even the processes, but time and again, the data shows that culture is essential for tooling and technology success in addition to the aforementioned IT and organizational performance.

We live and work in exciting times, and the integration of technology into our core businesses has made every firm a software firm. Technology now provides opportunities to deliver features to customers in new ways and at speeds not possible before. Organizations often find themselves struggling to keep up. Old IT and waterfall methods just don't allow organizations to deliver value fast enough—I've seen it in the data, and I've seen it among the customers and companies I work with to create solutions for their own DevOps journeys. Jennifer and Ryn have also seen the challenges to technology transformation in the old way, and the exciting opportunities that are possible with DevOps, and they've responded by writing this book to guide us all through our own journeys. So read through, and choose your own adventure! Iterate, learn, grow, and choose your own adventure again!

—Nicole Forsgren, PhD
Director, Chef Software
Seattle, WA

The Big Picture

Devops is a way of thinking and a way of working. It is a framework for sharing stories and developing empathy, enabling people and teams to practice their crafts in effective and lasting ways. It is part of the cultural weave that shapes how we work and why. Many people think about devops as specific tools like Chef or Docker, but tools alone are not devops. What makes tools “devops” is the manner of their use, not fundamental characteristics of the tools themselves.

In addition to the tools we use to practice our crafts, an equally important part of our culture is our values, norms, and knowledge. Examining how people work, the technologies we use, how technology influences the way we work, and how people influence technology can help us make intentional decisions about the landscape of our organizations, and of our industry.

Devops is not just another software development methodology. Although it is related to and even influenced by software development methodologies like Agile or XP, and its practices can include software development methods, or features like infrastructure automation and continuous delivery, it is much more than just the sum of these parts. While these concepts are related and may be frequently seen in devops environments, focusing solely on them misses the bigger picture—the cultural and interpersonal aspects that give devops its power.

A Snapshot of Devops Culture

What does a successful devops culture look like? To demonstrate this, we will examine the intersection of people, process, and tools at Etsy, an online global marketplace for handmade and vintage goods. We chose Etsy as an example not only because they are well-known in the industry for their technical and cultural practices, but also

because Ryn’s experiences working there allow us a more detailed look at what this culture looks like from the inside.

A new engineer at Etsy starts her first day with a laptop and a development virtual machine (VM) already set up with the appropriate accounts for access and authorization, the most common GitHub repositories cloned, aliases and shortcuts to relevant tools precreated, and a guide with new hire information and links to other company resources on her desktop. Standardization of tools and practices between teams makes it easier for new people to get up to speed regardless of what team they are joining, but every team also has the flexibility to customize as they see fit.

A current employee will pair with the new employee to walk through what testing and development processes she will use in her day-to-day work. She starts by writing code on her local development VM, which is set up with configuration management to be nearly identical to the live production environment. This development VM is set up to be able to run and test code locally, so she can quickly work and make changes without affecting anyone else’s development work during these early stages.

Etsy engineers can achieve a good degree of confidence that a change is working locally by running a suite of local unit and functional tests. From there, they test changes against the **try server**, a Jenkins cluster nearly identical to the organization’s production continuous integration (CI) cluster but with the added bonus of not requiring any code to be committed to the master branch yet. When try passes, engineers have an even higher degree of confidence that their changes will not break any tests.

Depending on the size and complexity of the change, this new engineer might choose to open a pull request or more informally ask for a code review from one of her colleagues. This is not mandated for every change, and is often left up to individual discretion—in Etsy’s high-trust, blameless environment, people are given the trust and authority to decide whether a code review is necessary. Newer or more junior employees are given the guidance to help them figure out what changes merit a code review and who should be involved. As a new employee, she has a teammate glance over her changes before she starts deploying them.

When local and try tests have passed, the developer then joins what Etsy calls a *push queue* to deploy her changes to production. This queue system uses Internet Relay Chat (IRC) and an IRC bot to coordinate deploys when multiple developers are pushing changes at once. When it is her turn, she pushes her commits to the master branch of the repository she is working in and uses **Deployinator** to deploy these changes to QA. This automatically starts builds on the QA server as well as running the full CI test suite.

After the builds and tests have completed successfully, she does a quick manual check of the QA version of the site and its logs to look for any problems that weren’t caught

by the automated tests. From there, she uses the same Deployinator process to deploy her code to production and make sure that the tests and logs look good there as well. If something does break that the tests don't catch, there are plenty of dashboards of graphs to watch, and Nagios monitoring checks to alert people to problems. In addition, many teams have their own Nagios checks that go to their own on-call rotations, encouraging everyone to share the responsibilities of keeping everything running smoothly. And if something doesn't run so smoothly, people work together to help fix any issues, and blameless postmortems mean that people learn from their mistakes without being chastised for making them.

This process is so streamlined that it takes only around 10 minutes on average from start to finish, and Etsy engineering as a whole deploys around 60 times a day. Documentation is available for anyone interested to peruse, and every engineer pushes code to production on their first day, guided by a current team member to help familiarize them with the process. Even nonengineers are encouraged to participate by way of the *First Push Program*, pairing with engineers to deploy a small change such as adding their photo to the staff page on the website. In addition to being used for regular software development, the try and Deployinator process works so well that it is used for nearly everything that can be deployed, from the tools that developers use to build virtual machines to Kibana dashboards for searching logs, from Nagios monitoring checks to the Deployinator tool itself.

The Evolution of Culture

This story of Etsy today is in stark contrast to how things were several years ago with a less transparent and more error-prone deployment process that took close to four hours. Developers had their own blade servers to work on, rather than virtual machines, but the blade servers weren't powerful enough to run the automated test suites to completion. Tests that were run in the staging environment took a couple of hours to complete, and even then they were flaky enough to make their results less than useful.

Teams within the engineering organization were siloed. There were a lot of developers throwing code over the metaphorical wall to ops, who were solely responsible for deployments and monitoring and thus tended to be incredibly change-averse. Developers wrote code, executing handcrafted shell scripts to create a new SVN branch which would be deployed using `svn merge`—not known for being the easiest merging tool to work with—to merge in all the developers' changes to this deploy branch. Developers would tell the one operations engineer who had permissions to deploy which branch to use. This would begin the painstaking, multihour deployment process (see [Figure 1-1](#)). Because the process was so painful, it happened only once every two or three weeks.

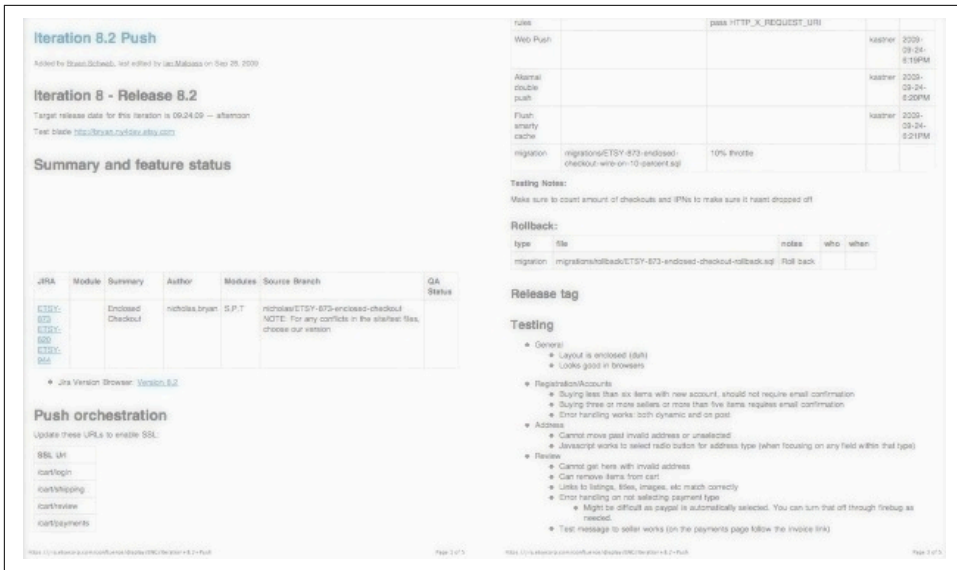


Figure 1-1. Before Deployinator, a complex and error-prone deploy process

People became fed up with this process. They realized something had to change. It would be hard for the deployment situation to get any worse. The organization was made up of smart and talented people with the motivation and now frustration to solve this problem. They had buy-in from the CEO and CTO down, which was key to having resources available to make change.

The keys to the deployment kingdom were shared from the one operations engineer to two developers, who were given the time and go-ahead to hack on this process as much as they needed. They say when you have a hammer, everything looks like a nail, and when you have a web application developer, everything looks like it needs a web app—and so the first Deployinator was born (Figure 1-2). At first, it was a web wrapper around the existing shell scripts, but over time, more people began to work and improve it. While the underlying mechanisms doing the heavy lifting changed, the overall interface stayed largely the same.

It became evident to everyone that empowering these employees to create tools to improve their jobs made it a lot easier for people to do said jobs. Deploys went from being in people's way to helping them accomplish the goal of getting features in front of users. Tests went from being flaky wastes of time to helping catch bugs. Logs, graphs, and alerts made it possible for everyone, not just a select few individuals, to see the impact of their work.

The key takeaway from the stories of all these tools isn't so much the specifics of the tools themselves, but the fact that someone realized that they needed to be built, and was also given the time and resources to build them.



Figure 1-2. After Deployinator, a simple web interface that anyone can use



Buy-in from management, freedom to experiment and work on non-customer-facing code, and trust across various teams all allowed these tools to be developed and along with them, a culture of tooling, sharing, and collaboration.

These factors made Etsy the so-called devops unicorn (they prefer to call themselves just a sparkly horse) they are known as today, and maintaining that culture is prioritized at all levels of the company. This example embodies our definition of effective devops—an organization that embraces cultural change to affect how individuals think about work, value all the different roles that individuals have, accelerate business value, and measure the effects of the change. It was these principles of devops that helped take Etsy from a state of frustration and silos to industry-renowned collaboration and tool builders. While the details may differ, we have seen these guiding principles in success stories throughout the industry, and in sharing them we provide a guide for organizations looking to make a similar transformation.

The Value of the Story

Each of us thinks our own thoughts. Our concepts we share.

—Stephen Toulmin, *Human Understanding*

Effective DevOps contains case studies and stories from both teams and individuals. When looking at the devops books that already existed, we found that there were fewer direct real-world experiences for people to draw from; so many stories focus on a specific tool or an abstract cultural practice. It is one thing to talk about how something should work in theory, but there is often a great deal of difference between how something should work in theory and how things turn out in practice. We want to share the actual stories of these practices, what did and didn't end up working, and the thought processes behind the decisions that were made, so that people have as much information as possible as they undertake their own devops journey.

Ryn's Story

My devops story begins around the same time as the devops movement itself. I made an accidental career change into operations shortly after the idea of devops was first formalized and the first devopsdays conference events had been held. As luck would have it, I found myself working as a one-person operations team at a small startup in the ecommerce space, and discovered I loved operations work. Even though I was a one-person team for so long, the idea of devops immediately made sense to me—it seemed to be a common sense, no-bullshit approach to working more effectively with other parts of an organization outside of your own team. At the time, I was the grumpy system administrator who spent their days hiding in the data center. I was the only person on-call, and was so busy fighting the fires that I had inherited that I had little to no idea what the developers (or anyone else, for that matter) were working on. So the ideas of sharing responsibilities and information and breaking down barriers between teams really spoke to me.

Some organizations are much more open to change and new ideas than others, and in addition to being rather change-resistant, this startup wasn't keen on listening to anything a very junior system administrator had to say. "You're not even a real sysadmin," they said as a way of dismissing my ideas. There also wasn't enough budget to even buy me a couple of books (I bought Tom Limoncelli's *The Practice of System and Network Administration* and *Time Management for System Administrators* on my own dime, and they were worth every penny), let alone send me to LISA or Velocity, and devopsdays New York wouldn't be started for another couple of years.

Luckily, I started to find the devops community online, and being able to talk to and learn from people who shared my passions for operations and learning and working together revitalized me. James Turnbull, now CTO of Kickstarter but who was at the time working at Puppet, found me on Twitter, struck up a conversation with me, and sent me a copy of his excellent *Pro Puppet* book at a time when I was struggling with having inherited 200 snowflake servers with not even a bash script to manage them. That small action introduced me to a growing and thriving community of people, and gave me hope that I could one day be a part of that community.

As Jennifer will note in her story as well, it's hard to effect change when you're burnt out, and after a year or so of trying to help change and improve my current company and being shut down for my lack of experience (which was growing by the day) every time, I also made the choice to move on. I kept learning and growing my skills, but still didn't feel totally aligned with the places I ended up working. I still seemed to be fighting against my coworkers and organizations, rather than working with them.

In January 2013, I went to the first devopsdays New York, soaking up all the talks and listening to as much of the hallway track as I could, even if I didn't feel like I was experienced enough to add anything to the conversations. I lived vicariously through

following #VelocityConf on Twitter. In October of that year, I gave a lightning talk at the second devopsdays New York, and through that met Mike Rembetsy, one of the co-organizers of that conference. He told me I should come work for Etsy, but after years of feeling like an impostor in my own field, I thought he was joking. I'd followed Code as Craft and the Etsy operations team online since I first discovered the operations and devops communities, but I didn't think I was good enough to join them.

I've never been more happy to be proven so wrong. My career in operations has taken me on a journey through several different organizational structures and ways that development, operations, and sometimes even “devops” teams worked together. Having worked at companies as small as a 25-person startup and as large as a decades-old enterprise organization with hundreds of thousands of employees, I've seen many ways of developing and delivering software and systems—some more effective than others.

Having spent time being the one person on-call 24/7 for the better part of a year and in other less-than-ideal workplace scenarios, I want to share the techniques and methods that have worked best for both me and my teams over the years to help reduce the number of people who might also find themselves being the single point of failure for part of their organization. A large part of my motivation for writing this book was being able to tell these stories—both ones that I've lived personally and ones told to me by others—so that we can share, learn, and grow as a community. Community helped me get to where I am today, and this book is just one way I have of giving back.

Jennifer's Story

In 2007, I was contacted by Yahoo management about a position that was “a little dev” and “a little ops”—a Senior Service Engineer position building out a multitenanted, hosted, distributed, and geographically replicated key/value data store called Sherpa.

As a service engineer at Yahoo, I honed my skills in programming, operations, and project management. I worked alongside the development and quality assurance teams building Sherpa, coordinating efforts with the data center, network, security, and storage teams. In 2009, as the murmurings of devops trickled into Yahoo, I discounted its value, as I already was a devops!

Fast-forward to the summer of 2011, and Jeff Park took over the leadership of my team. He helped grow the team so that we had multiple people in Service Engineering in the United States as well as in India. It wasn't enough. Jeff had concerns about me as an employee working nonstop and almost single-handedly keeping the service up. He was also concerned about the business and wanted to build resiliency into the

support model through redundant staff support. In December, he told me to take a real vacation—to not read my email or take phone calls but to disconnect.

I told him that I felt like something wasn't quite right, something wasn't working as expected. He told me that he'd fire me if I didn't take the time off. He reassured me that everything would be fine. I set up a simple visualization of the concerning metrics with JavaScript and a Perl script running on a cron job the evening before my vacation, confident that it would be enough to provide warning.

I came back from vacation to a degraded service. So many small issues that I had found over the years impacted the overall event, making it more difficult to debug. I felt like a complete failure, even though my last-minute visualization had been critical to identifying and monitoring the issue.

Jeff took me aside and said that he knew there was a high risk that things would break while I was out and additional risk of issues resulting from the team's historical complete reliance on me. My *heroics* were disguising the failures inherent in the system.

He felt that sometimes short-term setbacks are an OK thing, if you later use them as a lesson to right things over the long term. If things broke, it would help prioritize the criticality of further sharing, documenting, and distributing my knowledge and expertise to the business. Ultimately, that would lead to more stability and a better overall outcome for the organization and individuals on the team.

That event united the Sherpa team as we tried to repair the service and understand what happened. We divided into cross-functional teams to address the different components of the problem: failure handlers, communication group, tools, monitoring, and cleanup crew. Key individuals from management were available at all times and prepared to make hard decisions. These hard decisions helped us limit the overall length of the outage.

Failure sucks but instructs.

—Bob Sutton, *Stanford Management Instructor*

A key takeaway from this event for me was the value of failure. We couldn't be afraid to fail, and we needed to learn from the failure. We had ongoing meetings to resolve the operational issues that had been highlighted in the event. We continued to remediate outages as a cross-discipline team rather than limiting those activities to the Service Engineering team. We promoted discussions between our consumers and providers to better understand the weak points in the system.

Having spent over 10 years building work practices based on operations' tribal cultures of long hours, isolation, and avoiding system failures, how was I going to evoke the change I needed?

I was ready for devops. For me, the value of devops wasn't the mantra of "devs do X and ops do Y, or dev versus ops", but the shared stories, solving problems in a collaborative manner across the industry, and strengthening community. From the open spaces to the collaborative hacking, a new support system was emerging that strengthened the foundations of sustainable workplace practices and cultivated relationships between people.

Collaborating with Ryn on this book has strengthened my understanding of devops. Being able to share the working strategies and techniques from around the world to help improve and create sustainable work practices has been an incredible journey. It doesn't end with the final words in this book.

We are all gaining so much experience each and every day based on our different, diverse perspectives. Whether you are at the beginning of your career, knee-deep in cultural transformation, or about to change roles and responsibilities, your experiences can inform and educate others. I look forward to hearing and amplifying your stories so that together we grow as a community and collectively learn from our failures and successes.

Illustrating Devops with Stories

We have selected a variety of case studies to help illustrate the different ways in which a culture of effective devops can manifest. The goal of these stories is not to provide templates that can be followed exactly; blindly copying another organization or individual discounts all of the circumstances and reasoning that went into the choices they made.

These stories are illustrations or guides. Our hope is that you read these stories and see reflections of your experiences, perhaps as they are today, but maybe as how they could be in the future. We included stories from a variety of sources, both formal case studies and informal personal stories. While some stories are from organizations that are more well known, we deliberately chose to include stories from lesser-known sources as well, to showcase the variety of devops narratives that exist.



When you read these studies, consider not only the choices that were made and their outcomes, but also their circumstances and situations. What similarities can you see between their circumstances and your own, and what are the key differences? If you made the same choice in your own organization, what factors that are unique to your workplace would change the outcome? By reading and understanding these stories, we hope that you will be able to see their underlying themes, and start applying them to your own devops narrative.

Learning shouldn't stop at these shared stories. Experiment with new processes, tools, techniques, and ideas. Measure your progress, and most importantly, understand your reasons why. Once you start realizing what does and doesn't work with the things you try, you can begin to do more sophisticated experiments.

What Is Devops?

Devops is a cultural movement that changes how individuals think about their work, values the diversity of work done, supports intentional processes that accelerate the rate by which businesses realize value, and measures the effect of social and technical change. It is a way of thinking and a way of working that enables individuals and organizations to develop and maintain sustainable work practices. It is a cultural framework for sharing stories and developing empathy, enabling people and teams to practice their crafts in effective and lasting ways.

A Prescription for Culture

Devops is a prescription for culture. No cultural movement exists in a vacuum; social structure and culture are inherently intertwined. The hierarchies within organizations, industry connections, and globalization influence culture, as well as the values, norms, beliefs, and artifacts that reflect these areas. Software we create does not exist separately from the people who use it and the people who create it. Devops is about finding ways to adapt and innovate social structure, culture, and technology together in order to work more effectively.

The Devops Equation

The danger for a movement that regards itself as new is that it may try to embrace everything that is not old.

—Lee Roy Beach et al., *Naturalistic Decision Making and Related Research Lines*

This book is not a prescription for the One True Way of doing devops. While we will describe commonly seen misconceptions and anti-patterns, we are more interested in describing what a successful devops culture looks and acts like and how these principles can be applied across a variety of organizations and environments.

While the term *devops* itself is a portmanteau of “development” and “operations,” the core concepts of devops can and should be applied throughout the entire organization. A sustainable, successful business is more than the development and operations teams. Limiting our thinking to just those teams who write software or deploy it into production does the entire business a disservice.

Devops as Folk Model

In many ways, devops has become a *folk model*, a term used with different intent that leads to miscommunication and misunderstanding. In the field of cognitive science, a folk model is used as an abstraction for more concrete ideas and often substituted, being easier to understand than the concept ultimately being discussed. An example of this is the term *situational awareness*, which is often used as a stand-in for more specific ideas like perception and short-term memory. Folk models are not necessarily bad. They become problematic when different groups use the same term with different intent.

People will often spend more time arguing over what “devops” means—what folk model they are using for it—than they spend focusing on the ideas that they really want to discuss.¹ Sometimes, in order to get around the issue of defining devops and get people talking about concepts and principles, individuals will exaggerate “bad” behaviors as a way of focusing on the “good” behaviors that are seen as being “devops.” To talk about effective interteam collaboration, someone might use a cartoonish example of a company that creates a devops team that serves only to act as go-betweens for the development and operations teams, as we did in the Preface. It is an extreme example, but it gets people talking about something more meaningful and practical than a definition.

The Old View and the New View

In an environment where humans are blamed and punished for errors, a culture of fear can build up walls that prevent clear communication and transparency. Contrast this with a blameless environment, where issues are addressed cooperatively and viewed as learning opportunities for individuals and the organization. Professor Sidney Dekker described these two environments as the “old view” and the “new view” of human error in his book, *The Field Guide to Understanding Human Error*.²

The first environment views “human error as the cause of trouble.” This “old view” is described as a mindset in which the focus is on elimination of human error. Mistakes

1 Sidney Dekker and Erik Hollnagel, “Human Factors and Folk Models.” *Cognition, Technology & Work* 6, no. 2 (2004): 79–86.

2 Sidney Dekker, *The Field Guide to Understanding Human Error* (Farnham, UK: Ashgate Publishing Ltd, 2014).

are made by “bad apples” who need to be thrown out. This view is found in blameful cultures, as it assumes that errors are often caused by malice or incompetence. Individuals responsible for failure must be blamed and shamed (or simply fired).

The second environment views “human error as a symptom of trouble deeper in the system.” This “new view” is a mindset that sees human errors as structural rather than personal. People make choices and take the actions based on their context and what makes most sense to them, not intentional malice or incompetence. Organizations should consider systems holistically when looking to minimize or respond to issues.

Understanding and embracing the “new view” is key to understanding the devops movement. This view encourages us to share stories, as everything is a learning opportunity.

Shared stories:

- lead to increased transparency and trust within a team;
- instruct our coworkers in how to avoid a costly mistake without having to directly experience it; and
- increase the time spent on solving new problems, allowing for more innovation.

When these stories are shared throughout the industry, we impact the industry as a whole, creating new opportunities, knowledge, and shared understanding.

The Devops Compact

The heart of devops starts with people working not only as groups but as teams with a desire for mutual understanding. This can be described as a compact that teams will work together, communicate their intentions and the issues that they run into, and dynamically adjust in order to work toward their shared organizational goals.

Example of a compact

We can visualize this critical compact by examining the communication, clarification, and mutual trust of two rock climbers. Rock climbing is an activity that involves participants climbing up, down, or across natural rock formations or synthetic walls. The shared goal is to reach the top or end point of a specific route, usually without falling. It is a combination of the physical endurance required to navigate the problem as well as the mental acuity to understand and prepare for the next steps.

In some forms of rock climbing, one individual, the *climber*, will use a rope and a harness as protection against falls. The second individual, the *belay*, monitors the tension in the rope, giving the climber enough tension to prevent a long fall while also providing enough slack to give her room to maneuver as she climbs.

Belaying properly and safely requires both a shared understanding of the tools and process as well as ongoing communication. The climber will securely knot into her harness. The belayer will make sure his belay device is properly attached to his climbing harness. Each will trust but verify the status of the other's work before starting the climb.

The climb itself has a set of verbal cues to indicate readiness prior to approaching the climb, with the climber asking “on belay?” and the belayer communicating “belay on.” The climber responds with “climbing” to indicate her readiness. Finally, the belayer acknowledges with “climb on.”

The principles of this compact that make it work include:

- Shared, clearly defined goals
- Ongoing communication
- Dynamic adjustment and repairs of understanding

As we will illustrate next, these principles bring just as much to devops in the workplace as they do to climbers on the wall.

Example of the devops compact

Two employees work on separate teams at Sparkle Corp. The General is a senior developer with a number of different experiences in her background and has worked at Sparkle Corp for two years. George is an operations engineer with some experience and is relatively new to Sparkle Corp.

Their two teams support the global community of people that depend on the Sparkle Corp website for their creative endeavors. Their shared goal is to implement a new feature that will increase the value to end users, hopefully without impacting the site.

As the one with more experience at the company, the General will be really clear with George about the expectations, values, and processes in place at Sparkle Corp. In turn, George will be really clear with the General about when he needs help or doesn't understand part of the process. Both the General and George will check in with each other's work before proceeding to next steps—an example of the *trust-but-verify* model, as described with the climbing process.

The General and George have a shared understanding of their goals:

- Implementing a new feature that increases the value to Sparkle Corp customers
- Maintaining safety and trust in their communication with each other

In a siloed, nondevops environment, the lack of a shared understanding would be like the General trying to start coding without making sure George understands the

requirements—it might end up working, but without communication of intentions, the odds are stacked against it.

An organization will certainly run into unexpected issues or roadblocks along the way, but with the shared understanding that everyone is still a part of the compact, actions turn into repairs. We repair our misunderstandings about who would be working on a particular feature or when something would get done. We repair bugs that affect our understanding of how the software is supposed to behave. We repair processes and their documentation when things don't go the way we expect in production.

Throughout the book, we're going to take this idea of a devops compact and show how both the technological and cultural aspects of devops are ways of developing and maintaining this shared mutual understanding.

A History of Devops

Examining the history of the industry and the recurring patterns and ideas within it helps us to understand what shaped the devops movement. With that understanding we can make sense of where we are today and understand how, through effective devops, we can break the cycle of increasing specialization that creates silos and devaluation of specific roles.

Developer as Operator

In the beginning, the developer was the operator. When World War II broke out, the United States government put out a call for math majors to become “computers,” a job in which they were responsible for calculating ballistics firing tables for the war effort. Jean Bartik was one of the many women that responded. Her college advisor encouraged her to ignore the call, worrying that the repetitious tasks were not as admirable as continuing her family’s tradition of education.

While her advisor was right about the repetitious nature of calculating numbers, the job put Bartik in the right place at the right time to become one of the first programmers of the Electronic Numeric Integrator and Computer, or **ENIAC**, the first all-electronic, programmable computer system.

With no documentation and no plans, Bartik and the other five women working on the ENIAC figured out how to program it by reviewing the device’s hardware and logic diagrams. Programming the machine and its 18,000 vacuum tubes meant setting dials and changing out cable connections across 40 control panels.

At the time, the industry focused on hardware engineering and not on the programming required to make the system work. When problems arose, hardware engineers would come in and proclaim, “It’s not the machine; it’s the operators.” The program-

mers felt the pain of managing and operating these systems as they had to replace fuses and cables and remove literal bugs in the system.

The Advent of Software Engineering

In 1961, President John F. Kennedy set the challenge that within the decade the United States would land a person on the moon, and return them safely to Earth. Faced with this deadline but lacking employees with the necessary skills, the National Aeronautics and Space Administration (NASA) needed to find someone to write the onboard flight software required to accomplish this task. NASA enlisted Margaret Hamilton, a mathematician at the Massachusetts Institute of Technology (MIT) to lead the effort.¹

Hamilton recalls:

Coming up with new ideas was an adventure. Dedication and commitment were a given. Mutual respect was across the board. Because software was a mystery, a black box, upper management gave us total freedom and trust. We had to find a way and we did. Looking back, we were the luckiest people in the world; there was no choice but to be pioneers; no time to be beginners.²

In her pursuit of writing this complex software, Hamilton is credited with coining the term *software engineering*. She also created the concept of *priority displays*, software that alerts astronauts to information that requires their attention in real time. She instituted a set of requirement gathering that added quality assurance to the list of software engineering concerns, which included:

- debugging all individual components;
- testing individual components prior to assembly; and
- integration testing.

In 1969, during the Apollo 11 mission, the lunar module guidance computer software was tasked with too many calculations for its limited capacity. Hamilton's team had programmed the software such that it could be manually overridden, allowing Neil Armstrong to step in and pilot the lunar module using manual controls.

The freedom and trust that the management team afforded the team of engineers working on the onboard flight software, as well as the mutual respect between team members, led to software that facilitated one of humankind's great leaps in technol-

1 Robert McMillan, "Her Code Got Humans on the Moon—And Invented Software Itself," *WIRED*, October 13, 2015.

2 A. S. J. Rayl, "NASA Engineers and Scientists—Transforming Dreams Into Reality," 2008, http://www.nasa.gov/50th/50th_magazine/scientists.html.

ogy as Neil Armstrong stepped on the moon. Without this high-trust environment, this manual override ability (something that turned out to be of critical importance) might not have been present, and the moon landing story might have had a very different outcome.

The Problems of Software

Space flight was not the only area in which software was becoming critical in the 1960s. As hardware became more readily available, people became more concerned about the impending complexity of software that did not follow standards across other engineering disciplines. The growth rate of systems and the emerging dependence upon them were alarming.

In 1967, the NATO Science Committee, comprising scientists across countries and industries, held discussions to assess the state of software engineering. A Study Group on Computer Science was formed in the fall of 1967, whose goal was to focus attention on the problems of software. They invited 50 experts from all areas of industry, with three working groups focusing on the design of software, production of software, and service of software, in an effort to define, describe, and begin solving the problems of software engineering.

At the NATO Software Engineering Conference of 1968, key problems with software engineering were identified, including:

- defining and measuring success;
- building complex systems requiring large investment and unknown feasibility;
- producing systems on schedule and to specification, and;
- putting economic pressures on manufacturers to build specific products.

The identification of these problems would help to define and shape areas of focus for the industry in years to come, and still impacts us to this day.

The Advent of Proprietary Software and Standardization

Until 1964, the practice was to build computers that were specific and targeted to customer requirements. Software and hardware were not standardized or interchangeable. In 1964, International Business Machines (IBM) announced a family of computers known as the System/360—computers designed to support a wide range of utility from small to large and for commercial and scientific purposes.

The goal was to reduce the cost of product development, manufacturing, service, and support while also facilitating the ability for customers to upgrade as needed. The System/360 became the dominant mainframe computer, providing customers the

flexibility to start small and grow computing resources as needed. It also enabled job flexibility, as individuals could learn the software and hardware, which at the time were still tightly coupled, and then have the requisite skills needed for a similar job in another location.

Up until the late 1960s, computers were leased rather than bought outright. The cost of the hardware was high and incorporated the cost of software and services. Source code for the software was generally provided. In 1969, faced with a US antitrust lawsuit, IBM again impacted the industry by decoupling the software and hardware of their product, charging separately for the software associated with their mainframe hardware. This changed how software was viewed; software had suddenly acquired significant monetary value in and of itself and was not provided openly.

The Age of the Network

In 1979, a worldwide distributed discussion platform called Usenet was started by Tom Truscott and Jim Ellis, then students at Duke University. Usenet started out as a simple shell script that would automatically call different computers, search for changes in files on those computers, and copy changes from one computer to another using UUCP (Unix-to-Unix copy, a suite of programs allowing for file transfer and remote command execution between computers). Ellis gave a talk on the “Invitation to a General Access UNIX Network”³ at a Unix users group known as USENIX. This was one of the first ways to communicate and share knowledge across organizations with computers, and its use grew rapidly.

While this tool started to facilitate the sharing of knowledge across universities and corporations, this was also a time when the details of how companies were run were considered part of their “secret sauce.” Talking about solving problems outside of the company was not done, because such knowledge was viewed as a competitive advantage. There was an intentional cultural drive for competitors to work inefficiently. This stymied a great deal of collaboration and limited the effectiveness of the communication channels that were available. This cultural siloization led to companies growing in complexity.

Increasingly complex systems in turn led to the need for specialization of skills and role proliferation. Such roles included system administrators, specializing in systems management and minimizing system costs, and software engineers, specializing in creating of new products and features to address the new needs. Other more specialized groups were siloed off as well, with the NOC (network operations center), QA, security, databases, and storage all becoming separate areas of concern.

3 Ronda Hauben and Michael Hauben, *Netizens: On the History and Impact of Usenet and the internet* (Los Alamitos, CA: IEEE, 1997).

This situation created the institutional Tower of Babel, with the different silos all speaking different languages due to differing concerns. Along with this siloization, the specific pains of software and the hardware on which it ran were also separated. No longer were developers exposed to the late-night pages of down systems, or the anger expressed by unsatisfied users. Additionally, programming's trend toward higher-level languages meant that development became more abstracted, moving further and further away from the hardware and the systems engineers of the past.

In an effort to be proactive and prevent service outages, system administrators would document the set of steps required to do regular operations manually. System administrators borrowed the idea of “root cause analysis” from total quality management (TQM). This led in part to additional attention and effort toward minimizing risk. The lack of transparency and change management created growing amounts of entropy that engineers had to deal with more and more.

The Beginnings of a Global Community

As interconnected networks allowed programmers and IT practitioners to share their ideas online, people began looking for ways to share their ideas in person as well. User groups, where practitioners and users of various technologies could meet to discuss their fields, began to grow in number and popularity. One of the biggest worldwide user groups was DECUS, the Digital Equipment Computer Users' Society, which was founded in 1961 with members consisting mostly of programmers who wrote code for or maintained DEC computer equipment.

The US chapter of DECUS ran a variety of technical conferences and local user groups (LUGs) throughout the United States, while chapters in other countries were doing the same globally. These conferences and events began to publish their papers and ideas in the form of DECUS proceedings, which were made available to members as a way of sharing information and growing both the total knowledge of the community and the interconnectedness of its members.

A similar community specifically for system administrators was found with USENIX and its special interest group, the System Administrators Group. Known later as SAGE, today the group is known as the special interest group LISA (Large Installation System Administration), and it runs an annual conference with the same name.⁴ Separately, NSFNET “Regional-Tech” meetings evolved into the North American Network Operators' Group (NANOG), a community specifically for network administrators to increase collaboration to make the internet better.

⁴ USENIX has announced that LISA will be retired at the end of 2016. For details, see <https://www.usenix.org/blog/refocusing-lisa-community>.

Contrary to the focus on knowledge sharing that was a primary feature of these local and global user groups, at the same time there was a great deal of secrecy within technology companies regarding their practices. Companies, in their quests for their own financial and material successes, kept their processes as closely guarded secrets, because if their competitors had inefficient practices, that improved the likelihood of their own relative success. Employees were strongly discouraged or even explicitly forbidden from sharing knowledge at industry conferences to try to maintain this sort of competitive advantage. This is in stark contrast to more recent developments, where communities and conferences are growing around knowledge sharing and cross-collaboration between companies.



Trade Secrets and Proprietary Information

Information that is not generally known to the public that is sufficiently secret to confer economic or business advantage is considered a trade secret. Information a company possesses, owns, or holds exclusive rights to is considered proprietary. Software, processes, methods, salary structure, organizational structure and customer lists are examples of items that can be considered a company's proprietary information. For example, proprietary software is software for which the source code is generally not available to end users. All trade secrets are proprietary; not all proprietary information is secret.

In addition to the changes in culture in the industry, commoditization and the costs of knowledge and technology impact what companies keep secret within their organizations.

The Age of Applications and the Web

An early example of successful cooperation across organizational boundaries, the very popular **Apache HTTP Server** was released in 1995. Based on the public domain NCSA HTTP daemon developed by Robert McCool, an undergraduate at the University of Illinois at Urbana-Champaign, the modular Apache software enabled anyone to quickly deploy a web server with minimal configuration. This marked the beginning of a trend toward this and other open source solutions. Open source software, with licenses that allow users to read, modify, and distribute its source code, began to compete with proprietary, closed source solutions.

Combined with the availability of various distributions of the Linux operating system and the growth in popularity of scripting languages such as PHP and Perl, the open source movement led to the proliferation of the LAMP stack (most commonly Linux, Apache, MySQL, and PHP) as a solution for building web applications. MySQL, a relational database first released in 1995, combined with the server-side scripting capabilities of PHP, allowed developers to create dynamic websites and applications,

with more rapidly updated or dynamically generated content than before. Given the ease with which these new web applications could be created, people and organizations in the late 1990s had to begin working with more speed and flexibility in order to stay competitive.

It was a time of angst and frustration for both system administrators and computer programmers. Endemic in system administration, there was a long-standing culture of saying “no” and “it’s critical to preserve stability.” In 1992, Simon Travaglia started posting a series on Usenet called **The Bastard Operator From Hell (BOFH)** that described a rogue sysadmin who would take out his frustration and anger on the users of the system. Toxic operations environments led some individuals in this field to view that rogue sysadmin as a hero and emulate his behaviors, often to the detriment of others around them.

In development, there was a culture of “it’s critical to get these changes out” and “I don’t want to know how to do that because I’ll get stuck doing it.” In some environments, this prompted developers to risk system stability by finding unofficial ways to work around the established processes in order to meet their own goals. This in turn led to additional massive cleanups, further solidifying the idea that change is extremely risky. The singletons in either group who tried to make changes to the overall processes often found themselves stuck in the mire of becoming the subject matter expert, locked into the positions of support that became critical to maintain.

The Growth of Software Development Methodologies

In 2001, an invitation to discuss software development went out to people interested and active in the Extreme Programming (XP) community and others within the field. XP was a form of Agile development that was designed to be more responsive to changing requirements than previous development software methodologies, known for short release cycles, extensive testing, and pair programming. In response to this invitation, 17 software engineers got together in Snowbird, Utah. They summarized their shared common values to capture the adaptiveness and response to change that they wanted to see in development with an explicit emphasis on human factors. This Agile Manifesto was the rallying cry that started the Agile movement.

In 2004, Alistair Cockburn, a software developer who was one of the coauthors of the Agile Manifesto, described Crystal Clear,⁵ a software development methodology for small teams based on his 10 years of research with successful teams. It described three common properties:

5 Alistair Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams: A Human-Powered Methodology for Small Teams* (Boston: Addison Wesley, 2004).

- frequent delivery of usable code, moving toward smaller, more frequent deployments rather than large, infrequent ones;
- reflective improvement, or using reflections on what worked well and what worked poorly in previous work to help guide future work; and
- osmotic communication between developers—the idea that if developers are in the same room, information will drift through the background to be picked up informally, as by osmosis.

This movement continued in software development for several years, and later expanded its influence. Around that time, a system administrator named Marcel Wegermann wrote an essay on how to take the principles of Crystal Clear, Scrum, and Agile and apply them to the field of system administration. In addition to giving a lightning talk on the subject where he suggested ideas such as version control for the Linux operating system's */etc* directory, pair system administration, and operational retrospectives, he also started the Agile System Administration mailing list in 2008.

Open Source Software, Proprietary Services

As open source software proliferated and software in general became more modular and interoperable, engineers found themselves with more and more choices as they worked. Instead of being restricted to one hardware vendor and whatever operating system and proprietary software would run on that hardware, developers were now able to pick and choose which tools and technologies they wanted to use. As software, especially web software, became more commoditized, it became at once both more and less valuable, being less exclusive and more commonplace, but with software developers being highly paid and widely sought after.

In 2006, Amazon.com, Inc., an ecommerce company that until then had mostly been known as a website selling books and other goods to consumers, launched two services, Amazon Elastic Compute Cloud (EC2) and Amazon Simple Storage Service (S3), a first foray into providing virtualized compute instances and storage through a proprietary service. This allowed individuals to spin up compute resources quickly without a large upfront expenditure of hardware, and request more as needed. As with the introduction of the System/360, the service was quickly adopted, becoming the de facto standard due to its ease of use, low entry cost, and flexibility.

As web technology continued to grow and evolve, the ways that people communicated and collaborated online did too. Twitter, an online social networking service, was introduced to the world in 2006. At first it seemed very much like a tool for people wanting to share information in an abbreviated format, for short attention spans or for celebrities to reach out to fans. In 2007, however, its usage skyrocketed thanks to

the South by Southwest Interactive (SXSW) conference streaming live tweets about the conference on screens in the hallways.

Twitter quickly became a way for ad hoc communities to form across the globe. For conferences, it was a way to get additional value out of the multitrack systems and connect with like-minded individuals. The hallway track, a phrase often used to describe the interactions and conversations that take place in the hallways of conferences, had expanded from the physical world to the web, where anyone could discover and participate in these ad hoc interactions.

Agile Infrastructure

At the Agile 2008 conference in Toronto, system administrator and IT consultant Patrick Debois spoke about incorporating Scrum into operations with his talk, “Agile Operations and Infrastructure: How Infra-gile are You?” Patrick worked with development and operations teams on a project to test data center migration. In his work he might be doing development one day and fighting fires with the operations team the next; this context switching began to take a toll on him. Indeed, switching between even two tasks instead of focusing on just one can cause a nearly 20 percent drop in productivity due to the overhead of context switching.⁶

At that same conference, Andrew Clay Shafer, a former software developer who was starting to take a great interest in IT concerns, proposed an Agile Infrastructure session. He thought, however, that nobody would be interested in this topic, and ended up not attending the session that he himself proposed. When Patrick saw this, he realized that he wasn’t the only one interested in Agile system administration and contacted Andrew out of band to discuss the concept further.

Around the same time, individual companies were beginning not only to make great strides toward processes that allowed them to keep up with the increasingly rapid changes of the internet, but also to share some of their stories publicly through communities that were building up around popular conferences like the **O’Reilly Velocity Conference**.

One such company was Flickr, a popular community site for photographers. After being purchased by Yahoo in 2005, Flickr needed to migrate all of its services and data from Canada to the United States. John Allspaw, a web operations enthusiast who had worked in systems operations for years, had joined the company as the Flickr operations engineering manager to help with scaling this new migration project. Paul Hammond joined the Flickr development team in 2007, and became the

⁶ Gerald Weinberg, *Quality Software Management: Systems Thinking* (New York: Dorset House Publishing Company, 1997).

Flickr engineering manager in 2008, heading the development organization in collaboration with Allspaw.

At Velocity Santa Clara 2009, Hammond and Allspaw co-presented “10+ Deploys per Day: Dev and Ops Cooperation at Flickr,” highlighting the revolutionary change that allowed the team to move rapidly. They didn’t do this by setting out to break down silos or start a big professional and cultural movement. They were able to collaborate a great deal in their work at Flickr, which was in contrast to Allspaw’s previous experiences at Friendster, where emotions and pressures ran high and there was little in the way of interteam collaboration.



You can’t declare that you’re “doing devops successfully” simply because you are “doing 10 deploys a day.” Pay attention to the specific problems that you are trying to solve in your organization, not the metrics you hear from other organizations. Keep in mind why you are making specific changes rather than simply looking at the number of deploys or any other arbitrary metrics.

The opportunities to work together that presented themselves were something that both managers took advantage of. Neither of them woke up one day and decided that the company needed a big change; rather, they recognized the little pieces of working together that made things work well. They took note of these little things, which ended up becoming much bigger cultural changes, and that cooperation had a much larger impact than just the number of deploys.

The Beginning of devopsdays

Don’t just say ‘no’, you aren’t respecting other people’s problems... #velocityconf
#devops #workingtogether

—Andrew Clay Shafer (@littleidea)

This tweet, from Andrew Clay Shafer on June 23, 2009, prompted Patrick Debois to lament on Twitter that even though he was watching remotely, he was unable to attend that year’s Velocity conference in person. Pris Nasrat, at the time a lead systems integrator at the *Guardian*, tweeted in reply, “Why not organize your own Velocity event in Belgium?” Inspired, Patrick did almost exactly that, creating a local conference that would allow for developers, system administrators, toolsmiths, and other people working in those fields to come together. In October of that year, the first devopsdays conference took place in Ghent. Two weeks later, **Debois wrote:**

I’ll be honest, for the past few years, when I went to some of the Agile conferences, it felt like preaching in the desert. I was kinda giving up, maybe the idea was too crazy: developers and ops working together. But now, oh boy, the fire is really spreading.

That first devopsdays event ignited the powder keg of unmet needs; people separated in silos and frustrated with the status quo identified with devops and a way of describing the work they felt they were already doing. The conference grew and spread as individuals started up new local devopsdays events across the world. With the real-time communication made available by Twitter, the hallway track never ended and *#devops* took on a life of its own.

The Current State of Devops

It is inspiring to see how far the devops movement has come in the six years since Patrick Debois held the first devopsdays in Belgium. The [2015 State of Devops Report](#), published by Puppet, found that companies that are doing devops are outperforming those that aren't, finally showing numerically what many people have already suspected—that an emphasis on having teams and individuals work together effectively is better for business than silos full of engineers who don't exactly play well with others. High-performing devops organizations deploy code more frequently, have fewer failures, recover from those failures faster, and have happier employees.

The number of devopsdays conferences has increased from 1 in 2009 to 22 all over the world in 2015. Each year brings devopsdays events in new locations worldwide; this is not a phenomenon limited to technology hubs like Silicon Valley or New York. There are dozens of local Meetup groups with thousands of members in even more locations around the globe, not to mention the conversations about the topic happening daily on Twitter.

Summary

Reflecting on our history, we see the trend of focusing on outcomes rather than people and processes. Many took away from John Allspaw and Paul Hammond's "10+ Deploys a Day" presentation that what was important was the quantity of deployments—10+ deploys in a day. The subtitle "Dev & Ops Cooperation at Flickr" was missed behind the hook.

Fixation on a specific outcome increases stress for those who are already stressed out by limitations within the organization. Unlike mechanical processes, outcomes in software rely heavily on human factors. Software can be outdated before completion, neglect to meet customer expectations, and fail in unexpected ways with drastic impact.

Focusing on the culture and processes encourages iteration and improvement in how and why we do things. When we shift our focus from *what* to *why*, we are given the freedom and trust to establish meaningfulness and purpose for our work, which is a key element of job satisfaction. Engagement with work impacts outcomes without

concentrating on achieving a specific outcome, allowing for happy and productive humans building the next leap for humankind.

The introduction of devops has changed our industry by focusing on people and processes across roles to encourage collaboration and cooperation, rather than competing with specialization.

Foundational Terminology and Concepts

Building a strong foundation for effective devops requires discussing some key terms and concepts. Some of these concepts may be familiar to readers; many have been mentioned in the preceding history of software engineering or will be well known to readers who have experience with various software development methodologies.

Throughout the history of computer engineering, a number of methodologies have been described to improve and ease the process of software development and operations. Each methodology splits work into phases, each with a distinct set of activities. One issue with many methodologies is a focus on the development process as something separate from operations work, leading to conflicting goals between teams. Additionally, forcing other teams to follow particular methodologies can cause resentment and frustration if the work doesn't fit their processes and goals. Understanding how these different methodologies work and what benefits each might bring can help increase understanding and reduce this friction.



Devops is not so rigidly defined as to prohibit any particular methodology. While devops arose from practitioners who were advocating for Agile system administration and cooperation between development and operations teams, the details of its practice are unique per environment. Throughout this book, we will reiterate that a key part of devops is being able to assess and evaluate different tools and processes to find the most effective ones for your environment.

Software Development Methodologies

The process of splitting up development work, usually into distinct phases, is known as a *software development methodology*.

These different phases of work may include:

- Specification of deliverables or artifacts
- Development and verification of the code with respect to the specification
- Deployment of the code to its final customers or production environment

Covering all methodologies is far beyond the scope of this chapter, but we will touch on a few that have in one way or another impacted the ideas behind devops.

Waterfall

The waterfall methodology or model is a project management process with an emphasis on a sequential progression from one stage of the process to the next. Originating in the manufacturing and construction industries and adopted later by hardware engineering, the waterfall model was adapted to software in the early 1980s.¹

The original stages were requirements specification, design, implementation, integration, testing, installation, and maintenance, and progress was visualized as flowing from one stage to another (hence the name), as shown in **Figure 4-1**.

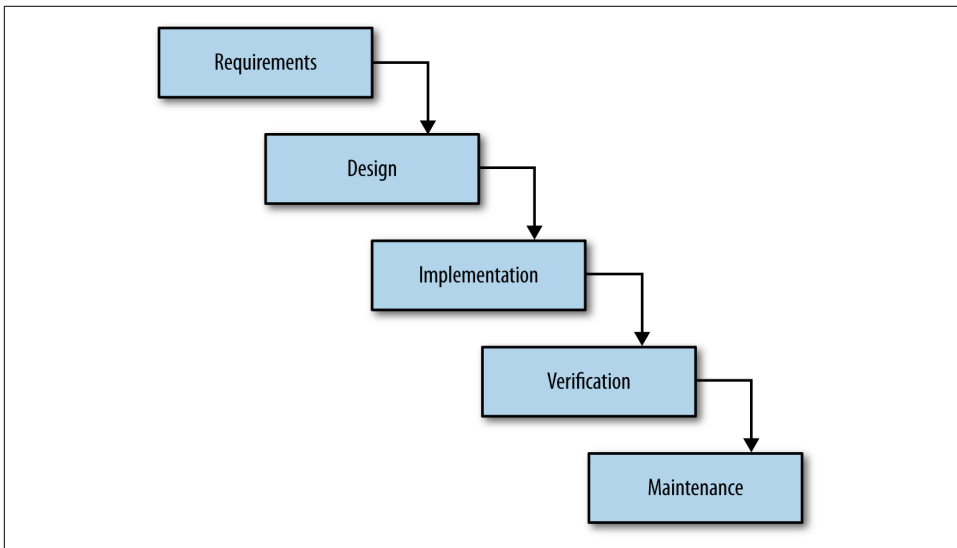


Figure 4-1. The waterfall model

¹ Herbert D. Benington, *Production of Large Computer Programs*. IEEE Annals of the History of Computing, October 1, 1983, <http://bit.ly/benington-production>.

Software development under the waterfall model tended to be very highly structured, based on a large amount of time being spent in the requirements and design phases, with the idea that if both of those were completed correctly to begin with it would cut down on the number of mistakes found later.

In waterfall's heyday, there was a high cost to delivering software on CD-ROMs or floppy disks, not including the cost to customers for manual installation. Fixing a bug required manufacturing and distributing new floppies or CD-ROMs. Because of these costs, it made sense to spend more time and effort specifying requirements up front rather than trying to fix mistakes later.

Agile

Agile is the name given to a group of software development methodologies that are designed to be more lightweight and flexible than previous methods such as waterfall. The Agile Manifesto, written in 2001 and described in the previous chapter, outlines its main principles as follows:

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

individuals and interactions over processes and tools
working software over comprehensive documentation
customer collaboration over contract negotiation
responding to change over following a plan

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

Agile methodologies include processes such as Scrum, which we will define next, and other methods that place a heavy emphasis on collaboration, flexibility, and the end result of working software.



Is Devops Just Agile?

Devops shares many characteristics with the Agile movement, especially with the focus on individuals, interactions, and collaboration. You might wonder if devops is just “rebranded” Agile. While devops has certainly grown around Agile methodology, it is a separate cultural movement steeped in the history of the computing industry with a broad reach that includes more than just developers. Devops adopts and extends Agile principles and applies them to the entire organization, not only the development process. As we will see in detail in later chapters, devops has cultural implications beyond Agile and a focus that is broader than speed of delivery.

Scrum

In the mid-1990s, Ken Schwaber and Dr. Jeff Sutherland, two of the original creators of the Agile Manifesto, merged individual efforts to present a new software development process called Scrum. Scrum is a software development methodology that focuses on maximizing a development team's ability to quickly respond to changes in both project and customer requirements. It uses predefined development cycles called *sprints*, usually between one week and one month long, beginning with a sprint planning meeting to define goals and ending with a sprint review and sprint retrospective to discuss progress and any issues that arose during that sprint.

One key feature of Scrum is the *daily Scrum* or *daily standup*, a daily meeting where team members (rather rapidly) each answer three questions:

- What did I do yesterday that helped the team meet its sprint goals?
- What am I planning to do today to help the team meet those goals?
- What, if anything, do I see that is blocking either me or the team from reaching their goals?

These meetings, which take place in the morning in order to help people align with what they are planning to do that day and help each other with any blocking issues, are often facilitated by the *Scrum master*. The Scrum master is an important role that also includes responsibilities such as helping the team self-organize and coordinate work efforts, helping remove blockers so the team will continue making progress, and involving project owners and stakeholders so there is a shared understanding of what “done” means and what progress is being made. The principles of Scrum are often seen applied less formally in many software development practices today.

Operations Methodologies

Similar to how software development methodologies split up software development work into different phases or otherwise try to bring more order to those processes, IT or operations work can be split up or organized as well. As with the software methodologies, covering all methodologies is far beyond the scope of this chapter.

ITIL

ITIL, formerly known as Information Technology Infrastructure Library, is a set of practices defined for managing IT services. It is published as a series of five volumes that describe its processes, procedures, tasks, and checklists, and is used to demonstrate compliance as well as measure improvement toward that end. ITIL grew out of a trend that saw the growing number of IT organizations in the 1980s using an increasingly diverse set of practices.

The British Central Computer and Telecommunications Agency developed the set of recommendations as a way to try to standardize these practices. First published in 1989, the books and practices have grown over the years, with the five core sections in the most recent (2011) version being service strategy, service design, service transition, service operation, and continual service improvement.

IT analyst and consultant Stephen Mann notes that while there are many benefits that come with ITIL's standardization and there are over 1.5 million ITIL-certified people worldwide, it has some areas where practitioners might want to put additional focus. Mann has said that ITIL is often more on the side of being reactive rather than proactive, so we suggest that organizations that have been using ITIL take note of ways that they can try to add more proactive planning and customer focus to their practices.

COBIT

Control Objectives for Information and Related Technology (COBIT) is an ISACA framework for governance and management of information and technology first released in 1996. A core principle of COBIT is to align business goals with IT goals.

COBIT is based on 5 principles:

- meeting stakeholder needs;
- covering the enterprise from end to end;
- applying a single integrated framework;
- enabling a holistic approach; and
- separating governance from management.

Systems Methodologies

Some methodologies focus on thinking about systems as a whole, rather than limiting focus to more specific areas such as software development or IT operations. Systems thinking skills are crucial for anyone working with complex systems like many of the software products that are created today; readers interested in learning more about systems thinking in general would do well to read *Thinking in Systems* by Donella Meadows and *How Complex Systems Fail* by Dr. Richard Cook.

Lean

After a five-year study on the future of automobile production and the Toyota Production System (TPS), James P. Womack, Daniel T. Jones, and Daniel Roos coined the

term *Lean Production*.² Womack and Jones defined the five principles of Lean Thinking as follows:³

- Value
- Value stream
- Flow
- Pull
- Perfection

These ideas, especially the pursuit of perfection through systemic identification and elimination of waste, drove the definition of *Lean* as the maximization of customer value and minimization of waste.

Lean systems focus on the parts of the system that add value by eliminating waste everywhere else, whether that be overproduction of some parts, defective products that have to be rebuilt, or time spent waiting on some other part of the system. Stemming from this are the concepts of Lean IT and Lean software development, which apply these same concepts to software engineering and IT operations.

Waste to be eliminated in these areas can include:

- Unnecessary software features
- Communication delays
- Slow application response times
- Overbearing bureaucratic processes

Waste in the context of Lean is the opposite of value. Mary Poppendieck and Thomas Poppendieck have mapped Lean manufacturing waste to software development waste as follows:⁴

- Partially done work
- Extra features
- Relearning
- Unnecessary handoffs

² James P. Womack, Daniel T. Jones, and Daniel Roos, *The Machine That Changed the World* (New York: Rawson Associates, 1990).

³ James P. Womack and Daniel T. Jones, *Lean Thinking* (New York: Simon & Schuster, 1996).

⁴ Mary Poppendieck and Thomas David Poppendieck. *Implementing Lean Software Development* (Upper Saddle River, NJ: Addison-Wesley, 2007).

- Task switching
- Delays
- Defects

As with devops, there is no one way to do Lean software development. There are two main approaches to Lean: a focus on waste elimination through a set of tools, and a focus on improving the flow of work, also known as *The Toyota Way*.⁵ Both approaches have the same goal, but due to the differing approaches may result in different outcomes.

Development, Release, and Deployment Concepts

There are several terms related to the development, release, and deployment of software that have not previously been covered in the definitions of the methodologies discussed so far in this chapter. These are concepts that describe the *hows* of developing and deploying software, and understanding what they are and how they relate will give readers a more mature understanding of how tools can be used to facilitate these practices down the line.

Version Control

A version control system records changes to files or sets of files stored within the system. This can be source code, assets, and other documents that may be part of a software development project. Developers make changes in groups called *commits* or *revisions*. Each revision, along with metadata such as who made the change and when, is stored within the system in one way or another.

Having the ability to commit, compare, merge, and restore past revisions to objects to the repository allows for richer cooperation and collaboration within and between teams. It minimizes risks by establishing a way to revert objects in production to previous versions.

Test-Driven Development

In test-driven development, the code developer starts by writing a failing test for the new code functionality, then writes the code itself, and finally ensures that the test passes when the code is complete. The test is a way of defining the new functionality clearly, making more explicit what the code should be doing.

⁵ Jeffrey K. Liker, *The Toyota Way: 14 Management Principles from the World's Greatest Manufacturer* (New York: McGraw-Hill, 2004).

Having developers write these tests themselves not only greatly shortens feedback loops but also encourages developers to take more responsibility for the quality of the code they are writing. This sharing of responsibility and shorter development cycle time are themes that continue to be important parts of a devops culture.

Application Deployment

Application deployment is the process of planning, maintaining, and executing on the delivery of a software release. In the general sense, the craft of application deployment needs to consider the changes that are taking place underneath the system. Having infrastructure automation build the dependencies required to run a specific application—whether they be compute, operating system, or other dependencies—minimizes the impact of inconsistencies on the released software.

Depending on the application type, different engineering concerns may be important. For example, databases may have strict guarantees in terms of consistency. If a transaction occurs, it must be reflected in the data. Application deployment is a critical aspect to engineering quality software.

Continuous Integration

Continuous integration (CI) is the process of integrating new code written by developers with a mainline or “master” branch frequently throughout the day. This is in contrast to having developers working on independent feature branches for weeks or months at a time, merging their code back to the master branch only when it is completely finished. Long periods of time in between merges means that much more has been changed, increasing the likelihood of some of those changes being breaking ones. With bigger changesets, it is much more difficult to isolate and identify what caused something to break. With small, frequently merged changesets, finding the specific change that caused a regression is much easier. The goal is to avoid the kinds of integration problems that come from large, infrequent merges.

In order to make sure that the integrations were successful, CI systems will usually run a series of tests automatically upon merging in new changes. When these changes are committed and merged, the tests automatically start running to avoid the overhead of people having to remember to run them—the more overhead an activity requires, the less likely it is that it will get done, especially when people are in a hurry. The outcome of these tests is often visualized, where “green” means the tests passed and the newly integrated build is considered clean, and failing or “red” tests means the build is broken and needs to be fixed. With this kind of workflow, problems can be identified and fixed much more quickly.

Continuous Delivery

Continuous delivery (CD) is a set of general software engineering principles that allow for frequent releases of new software through the use of automated testing and continuous integration. It is closely related to CI, and is often thought of as taking CI one step further, that beyond simply making sure that new changes can be integrated without causing regressions to automated tests, continuous delivery means that these changes can *be deployed*.

Continuous Deployment

Continuous deployment (also referred to as CD) is the process of deploying changes to production by defining tests and validations to minimize risk. While continuous delivery makes sure that new changes can be deployed, continuous deployment means that they *get deployed* into production.

The more quickly software changes make it into production, the sooner individuals see their work in effect. Visibility of work impact increases job satisfaction, and overall happiness with work, leading to higher performance. It also provides opportunities to learn more quickly. If something is fundamentally wrong with a design or feature, the context of work is more recent and easier to reason about and change.

Continuous deployment also gets the product out to the customer faster, which can mean increased customer satisfaction (though it should be noted that this is not a panacea—customers won't appreciate getting an updated product if that update doesn't solve any of their problems, so you have to make sure through other methods that you are building the right thing). This can mean validating its success or failure faster as well, allowing teams and organizations to iterate and change more rapidly as needed.



The difference between Continuous Delivery and Continuous Deployment is one that has been discussed a great deal since these topics became more widely used. Jez Humble, author of *Continuous Delivery* defines continuous delivery as being a general set of principles that can be applied to any software development project, including the *internet of things* (IoT) and embedded software, while continuous deployment is specific to web software. For more information on the differences between these two concepts, see the Further Resources for this chapter.

Minimum Viable Product

One theme that has become apparent especially in recent years is the idea of reducing both development costs and waste associated with creating products. If an organization were to spend years bringing a new product to market only to realize after the

fact that this new product didn't meet the needs of either new or existing customers, that would have been an incredible waste of time, energy, and money.

The idea of the minimum viable product (MVP) is to create a prototype of a proposed product with the minimum amount of effort required to determine if the idea is a good one. Rather than developing something to 100 percent completion before getting it into users' hands, the MVP aims to drastically reduce that amount, so that if significant changes are needed, less time and effort has already been spent. This might mean cutting down on features or advanced settings in order to evaluate the core concept, or focusing on features rather than design or performance. As with ideas such as Lean and continuous delivery, MVPs allow organizations to iterate and improve more quickly while reducing cost and waste.

Infrastructure Concepts

All computer software runs on infrastructure of some sort, whether that be hardware that an organization owns and manages itself, leased equipment that is managed and maintained by someone else, or on-demand compute resources that can easily scale up or down as needed. These concepts, once solely the realm of operations engineers, are important for anyone involved with a software product to understand in environments where the lines between development and operations are starting to blur.

Configuration Management

Started in the 1950s by the United States Department of Defense as a technical management discipline, configuration management (CM) has been adopted in many industries. Configuration management is the process of establishing and maintaining the consistency of something's functional and physical attributes as well as performance throughout its lifecycle. This includes the policies, processes, documentation, and tools required to implement this system of consistent performance, functionality, and attributes.

Specifically within the software engineering industry, various organizations and standards bodies such as ITIL, IEEE (the Institute of Electrical and Electronics Engineers), ISO (the International Organization for Standardization), and SEI (the Software Engineering Institute) have all proposed a standard for configuration management. As with other folk models, this has led to some confusion in the industry about a common definition for the term.

Often this term is conflated with various forms of infrastructure automation, version control, or provisioning, which creates a divide with other disciplines' usage of the term. To ensure a common understanding for this book's audience, we define configuration management as the process of identifying, managing, monitoring, and audit-

ing a product through its entire lifecycle, including the processes, documentation, people, tools, software, and systems involved.

Cloud Computing

Cloud computing, often referred to as just “the cloud,” refers to a type of shared, internet-based computing where customers can purchase and use shared computing resources offered by various cloud providers as needed. Cloud computing and storage solutions can spare organizations the overhead of having to purchase, install, and maintain their own hardware.

The combination of high performance, cost savings, and the flexibility and convenience that many cloud solutions offer has made the cloud an ideal choice for organizations that are looking to both minimize costs and increase the speed at which they can iterate. Iteration and decreased development cycle time are key factors in creating a devops culture.



While some see the cloud as being synonymous with devops, this is not universally the case. A key part of devops is being able to assess and evaluate different tools and processes to find the most effective one for your environment, and it is absolutely possible to do that without moving to cloud-based infrastructure.

Infrastructure Automation

Infrastructure automation is a way of creating systems that reduces the burden on people to manage the systems and their associated services, as well as increasing the quality, accuracy, and precision of a service to its consumers. Indeed, automation in general is a way to cut down on repetitious work in order to minimize mistakes and save time and energy for human operators.

For example, instead of running the same shell commands by hand on every server in an organization’s infrastructure, a system administrator might put those commands into a shell script that can be executed by itself in one step rather than many smaller ones.

Artifact Management

An artifact is the output of any step in the software development process. Depending on the language, artifacts can be a number of things, including JARs (Java archive files), WARs (web application archive files), libraries, assets, and applications. Artifact management can be as simple as a web server with access controls that allow file management internal to your environment, or it can be a more complex managed service with a variety of extended features. Much like early version control for source

code, artifact management can be handled in a variety of ways based on your budgetary concerns.

Generally, an artifact repository can serve as:

- a central point for management of binaries and dependencies;
- a configurable proxy between organization and public repositories; and
- an integrated depot for build promotions of internally developed software.

Containers

One of the bigger pain points that has traditionally existed between development and operations teams is how to make changes rapidly enough to support effective development but without risking the stability of the production environment and infrastructure. A relatively new technology that helps alleviate some of this friction is the idea of *software containers*—isolated structures that can be developed and deployed relatively independently from the underlying operating system or hardware.

Similar to virtual machines, containers provide a way of sandboxing the code that runs in them, but unlike virtual machines, they generally have less overhead and less dependence on the operating system and hardware that support them. This makes it easier for developers to develop an application in a container in their local environment and deploy that same container into production, minimizing risk and development overhead while also cutting down on the amount of deployment effort required of operations engineers.

Cultural Concepts

The final concepts we define in this chapter are cultural ones. While some software development methodologies, such as Agile, define ways in which people will interact while developing software, there are more interactions and related cultural concepts that are important to cover here, as these ideas will come up later in this book.

Retrospective

A retrospective is a discussion of a project that takes place after it has been completed, where topics such as what went well and what could be improved in future projects are considered. Retrospectives usually take place on a regular (if not necessarily frequent) basis, either after fixed periods of time have elapsed (every quarter, for example) or at the end of projects. A big goal is local learning—that is, how the successes and failures of this project can be applied to similar projects in the future. Retrospective styles may vary, but usually include topics of discussion such as:

What happened?

What the scope of the project was and what ended up being completed.

What went well?

Ways in which the project succeeded, features that the team is especially proud of, and what should be used in future projects.

What went poorly?

Things that went wrong, bugs that were encountered, deadlines that were missed, and things to be avoided in future projects.

Postmortem

Unlike the planned, regular nature of a retrospective, a postmortem occurs after an unplanned incident or outage, for cases where an event's outcome was surprising to those involved and at least one failure of the system or organization was revealed. Whereas retrospectives occur at the end of projects and are planned in advance, postmortems are unexpected before the event they are discussing. Here the goal is organizational learning, and there are benefits to taking a systemic and consistent approach to the postmortem by including topics such as:

What happened?

A timeline of the incident from start to finish, often including communication or system error logs.

Debrief

Every person involved in the incident gives their perspective on the incident, including their thinking during the events.

Remediation items

Things that should be changed to increase system safety and avoid repeating this type of incident.

In the devops community, there is a big emphasis placed on postmortems and retrospectives being *blameless*. While it is certainly possible to have a blameful postmortem that looks for the person or people “responsible” for an incident in order to call them out, that runs counter to the focus on learning that is central to the devops movement.

Blamelessness

Blamelessness is a concept that arose in contrast to the idea of blame culture. Though it had been discussed for years previously by Sidney Dekker and others, this idea really came to prominence with John Allspaw's [post on blameless postmortems](#), with the idea that incident retrospectives would be more effective if they focused on learning rather than punishment.

A culture of blamelessness exists not as a way of letting people off the hook, but to ensure that people feel comfortable coming forward with details of an incident, even if their actions directly contributed to a negative outcome. Only with all the details of how something happened can learning begin to occur.

Organizational Learning

A learning organization is one that learns continuously and transforms itself...Learning is a continuous, strategically used process—integrated with and running parallel to work.

—Karen E. Watkins and Victoria J. Marsick, *Partners for Learning*

Organizational learning is the process of collecting, growing, and sharing an organization's body of knowledge. A learning organization is one that has made their learning more deliberate, setting it as a specific goal and taking actionable steps to increase their collective learning over time.

Organizational learning as a goal is part of what separates blameful cultures from blameless ones, as blameful cultures are often much more focused on punishment than on learning, whereas a blameless or learning organization takes value from experiences and looks for lessons learned and knowledge gained, even from negative experiences. Learning can happen at many different levels, including individual and group as well as organization, but organizational learning has higher impact to companies as a whole, and companies that practice organizational learning are often more successful than those that don't.

Summary

We have discussed a variety of methodologies relating to the development, deployment, and operation of both software and the infrastructure that underlies it, as well as cultural concepts addressing how individuals and organizations deal with and learn from incidents and failures.

This is far from an exhaustive list, and new methodologies and technologies will be developed in the future. The underlying themes of development, deployment, operations, and learning will continue to be core to the industry for years to come.

Devops Misconceptions and Anti-Patterns

When talking about concepts such as devops, it can also be helpful to discuss what the concept is *not*. This process will help clear up some common misconceptions or misconstrued ideas about devops. In this chapter, we will provide additional context around devops as well as define some common anti-patterns.

Common Devops Misconceptions

Throughout the industry, there are common misconceptions about devops. Within your organization, your teams may struggle to clarify and articulate beliefs and values. In this section, we'll examine some of the issues people run into when trying to establish this common language in their organization.

Devops Only Involves Developers and System Administrators

While the name is a portmanteau of *development* (or *developers*) and *operations*, this is more of a signal of the origin of the movement than a strict definition of it. While the devopsdays conference tagline is “the conference that brings development and operations together,” the concepts and ideas of devops include all roles within an organization. There is no one definitive list of which teams or individuals should be involved or how, just as there is no one-size-fits-all way to “do devops.”

Ideas that help development and operations teams communicate better and work more efficiently together can be applied throughout a company. Any team within the organization should be considered—including security, QA, support, and legal—in order to be most effective. For example, effective devops processes between legal and sales allow for automated contract creation based on a consistent sales catalog.

Devops Is a Team

In general, the creation of a designated “devops team” is a less-than-ideal circumstance. Creating a team called devops, or renaming an existing team to devops, is neither necessary nor sufficient for creating a devops culture. If your organization is in a state where the development and operations teams cannot communicate with each other, an additional team is likely to cause more communication issues, not fewer. Underlying issues need to be addressed for any substantial and lasting change to stick.

Creating a separate team as an environment in which to kickstart new processes and communication strategies can be effective if it is seen as a greenfield project. In large companies this is generally a useful short-term strategy to kick off meaningful change and usually results in blending the team members back into designated-role teams as time progresses.

In a startup environment, having a single team that encompasses both functions can work if it allows for the team to embrace the responsibility and mission of the service as a collaborative unit rather than burning out a single individual on-call. Management will still need to facilitate clear roles and responsibilities to ensure that as the company grows the team can scale out as required.

This book will cover different team organizational options and interteam communication and coordination strategies, but ultimately it is important to remember that there is no one right or wrong way of doing devops, and if having a team with devops in its name genuinely works for you, there’s no reason to change it. Keep in mind that devops is a culture and a process, and name and structure your teams accordingly.

Devops Is a Job Title

The “devops engineer” job title has started a controversial debate. The job title has been described in various ways, including:

- a system administrator who also knows how to write code;
- a developer who knows the basics of system administration; or
- a mythical 10x engineer (said to be 10 times as productive as other engineers, though this is difficult to measure and often used figuratively) who can be a full-time system administrator and full-time developer for only the cost of one salary without any loss in the quality of their work.

In addition to being totally unrealistic, this concept of a devops engineer doesn’t scale well. It might be necessary to have the developers deploying the code and maintaining the infrastructure in the early days of an organization, but as a company matures and grows, it makes sense to have people become more specialized in their job roles.

It doesn't usually make much sense to have a director of devops or some other position that puts one person in charge of devops. Devops is at its core a cultural movement, and its ideas and principles need to be used throughout entire organizations in order to be effective.

Still, “devops engineer” is a job title that keeps appearing in the wild. Engineers with “devops” in their title earn higher salaries than regular system administrators, according to the [2015 DevOps Salary Report from Puppet](#). With the lack of parity between roles at different organizations based on different interpretations of what a devops engineer role looks like, this is a very slippery slope. Who wouldn't want to be called a devops engineer if it immediately gave them a \$10K bump in salary?



A concerning trend as reported by the 2015 DevOps Salary Report is that “more devops engineers reported working in excess of 50 hours, systems engineers reported working 41–50 hours per week, and system administrators reported working 40 or fewer hours per week.” When choosing your job, make sure that the higher salary doesn't come at the cost of less personal time and higher rates of burnout.

Devops Is Relevant Only to Web Startups

A primary product of web-based companies is the web application that the user sees when browsing the company website. These websites can often be implemented without relying on information stored between interactions with the website (i.e., stateless sessions). Upgrading the site can be done in stages or through the migration of live traffic to a new version of the site, making the process of continuous deployment easy.

It is easy to see why devops makes sense for web-based companies like this: the movement helps break down barriers that can impede development and deployment. If a web company's processes are so slow that it takes a matter of weeks to fix a typo, chances are they aren't going to do very well compared to newer, more Agile companies in the space.

Web startups are not alone in benefiting from improved collaboration, affinity, and tools. It is easier to iterate on team structures and processes at a small startup; enterprises have been trained to resist rapid change and governmental agencies even more so with laws that may actively restrict and impede change. Change is possible even in these sorts of organizations, however.

You Need a Devops Certification

A significant part of devops is about culture: how do you certify culture? There is no 60-minute exam that can certify how effectively you communicate with other people,

how well teams in your company work together, or how your organization learns. Certifications make sense when applied to specific technologies that require a high level of expertise to use, such as specific software or hardware. This allows companies that require that specific technology or expertise to gain some understanding of a particular individual's knowledge in that area.

Devops doesn't have required technology or one-size-fits-all solutions. Certification exams are testing knowledge where there are clear right or wrong answers, which devops generally does not have. What works best for one company won't necessarily be optimal for another; there's no easy way to write questions that would be universally correct for devops. A devops certification is likely to be a money-making opportunity or specific to a vendor's solution and mislabeled devops certification.

Devops Means Doing All the Work with Half the People

There are some people under the impression that devops is a way to get both a software developer and a system administrator in one person—and with one person's salary. Not only is this perception incorrect, but it is often harmful. At a time when too many startups are offering perks such as three meals a day in the office and on-site laundry as a way of encouraging workers to spend even more time at work, and when too many engineers find themselves working 60–80 hours a week, misconceptions that drive people further away from work–life balance and more toward overwork are not what our industry needs.

During very early stages, it is true that a startup can benefit from having developers who understand enough about operations to handle deployments as well, especially with cloud providers and other “as a services” to handle a lot of operational heavy lifting. Once an organization gets past the point where every single employee must wear multiple hats out of sheer necessity, expecting one person to fill two full-time roles is asking for burnout.



Devops doesn't save money by halving the number of engineers your company needs. Rather, it allows organizations to increase the quality and efficiency of their work, reducing the number and duration of outages, shortening development times, and improving both individual and team effectiveness.

There Is One “Right Way” (or “Wrong Way”) to Do Devops

Early adopters of devops practices and principles, especially those well-known for this in the industry such as Netflix and Etsy, are often regarded as “unicorns”¹ who have cornered the market on the “right” way to do devops. Other companies, eager to get the benefits of a devops culture, will sometimes try to emulate their practices.

Just because a company presents their successful devops strategy does not mean that these same processes will be the right way of doing devops within every environment. Cargo culting² processes and tools into an environment can lead to the creation of additional silos and resistance to change.



Devops encourages critical thinking about processes, tools, and practices; being a learning organization requires questioning and iterating on processes, not accepting things as the “one true way” or the way that things have always been done.

One should also beware of people saying that anyone who isn’t following their example is doing devops “the wrong way.” It bears repeating that while there are valid criticisms of devops teams or devops engineers, there are also documented cases of companies and people who make those terms work for them. Devops intentionally is not rigidly defined like Scrum or ITIL. The companies that are doing devops most successfully are comfortable learning and iterating to find what tools and processes are most effective for them.

It Will Take *X* Weeks/Months to Implement Devops

If some sort of management buy-in is required for an organizational transformation such as those involved in devops, one of the questions often asked about the transformation is how long it will take. The problem with this question is that it assumes that devops is an easily definable or measurable state, and once that state is reached then the work is done.

In reality, devops is an ongoing process; it is the journey, not the destination. Some parts of it will have a fixed end point (such as setting up a configuration management system and making sure that all the company’s servers are being managed by it), but the ongoing maintenance, development, and use of configuration management will continue.

1 Within devops, a *unicorn* is an internet company that is a practitioner, innovator, and early adopter of devops. This is not to be confused with the financial definition of *unicorn* as a startup valued at over \$1 billion.

2 *Cargo culting* describes the practice of emulating observed behaviors or implementing tools into an environment without fully understanding the reasoning or circumstances that led to success using those strategies.

Because so much of devops is cultural, it is harder to predict how long some of those changes will take: how long will it take people to break old siloed habits and replace them with new collaborative ones? This cannot be easily predicted, but don't let that stop you from working toward these kinds of significant cultural transformations.

Devops Is All About the Tools

While tools are valuable, devops does not mandate or require any particular ones. This misconception is a contributing factor to the idea that devops is only for start-ups, as enterprise companies are often less able to adopt new technologies.

Devops is a cultural movement. Within your environment the tools you use currently are part of your culture. Before deciding on a change, it behooves you to recognize the tools in the environment that have been part of the existing culture, understand individuals' experiences with those tools, and observe what is similar and different between others' experiences. This examination and assessment helps clarify what changes need to be made.

Technology impacts velocity and organizational structures within your organization. Making drastic changes to tools, while they may have value to an individual or team, may have a cost that decelerates the organization as a whole.

The principles discussed in this book don't require a specific set of tools and are applicable to any technology stack. There is a fair amount of overlap between companies that practice devops and those who use containers or cloud providers, but that doesn't mean that those particular technologies are required—there are certainly companies successfully implementing devops while running on bare metal.

Devops Is About Automation

Many innovations in devops-adjacent tools help to codify understanding, bridging the gaps between teams and increasing velocity through automation. Practitioners have focused on tools that eliminate tasks that are boring and repetitive, such as with infrastructure automation and continuous integration. In both of these cases, automation is a result of improved technology.

If there are repetitive tasks that could be automated to free up a human from having to do them, that automation helps that person work more efficiently. Some cases like this have fairly obvious gains: automating server builds saves hours per server that a system administrator can then spend on more interesting or challenging work. But if more time is spent trying to automate something than would be saved by having it be automated, that is no longer time well spent (see [Figure 5-1](#)).

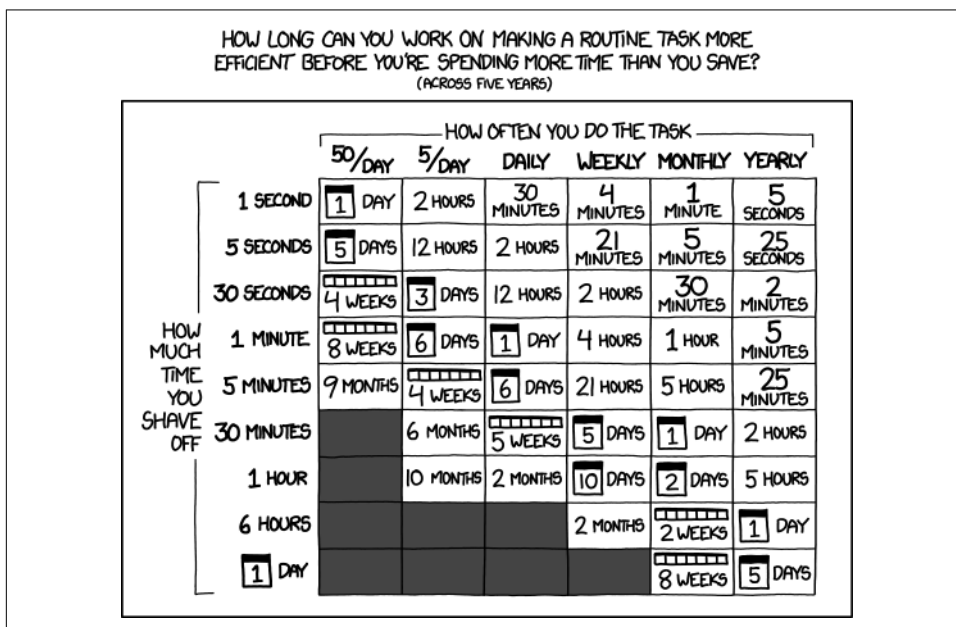


Figure 5-1. A comic from XKCD on time spent versus time saved (<https://xkcd.com/1205>); originally published with the alt text: “Don’t forget the time you spend finding the chart to look up what you save. And the time spent reading this reminder about the time spent. And the time trying to figure out if either of those make sense. Remember, every second counts toward your life total, including these right now.”

There has been a great deal of discussion about the role of automation in any environment and the way that human factors affect what and how we choose to automate. Paying attention to these stories is an example of cross-discipline affinity; other industries may have much to teach us about our own if we pay attention to them.

Automation is critical to us as systems become more complex and organizations become interdependent due to shared services. Without shared mutual context or concern for human needs, however, automation creates unknown additional risk. Automation may make work faster, but in order to be most effective it must also increase transparency, collaboration, and understanding.

Early Automation in the Aviation Industry

While reviewing the aviation industry in 1977, the House Committee on Science and Technology identified cockpit automation as a leading safety concern. Studies of the aviation field discovered that while pilots could still fly planes, the automation in use was causing their critical thinking skills to atrophy. Pilots were losing the ability to track position without the use of a map display, decide what should be done next, or recognize instrument system failures. This is a warning for us as we implement automation within our environments. Tools change our behavior and the way that we think.

In July 2013, Asiana Airlines flight 214 struck a seawall at San Francisco International Airport, with three people being fatally injured. During the investigation, the National Transportation Safety Board (NTSB) identified a number of issues, among them that there was insufficient monitoring of airspeed due to an overreliance on automated systems that the pilots didn't fully understand.

In their efforts to compensate for the unreliability of human performance, the designers of automated control systems have unwittingly created opportunities for new error types that can be even more serious than those they were seeking to avoid.

—James Reason, *Managing the Risks of Organizational Accidents*

Devops Is a Fad

As devops is not specific to a technology, tool, or process, it is less likely to become obsolete or replaced with something new. A movement about improving organizational effectiveness and individual employee happiness may be absorbed into common use, but it will not age out.

One of the primary differences between devops and methodologies like ITIL and Agile is that the latter have strict definitions, with context being added over time. Devops, on the other hand, is a movement defined by stories and ideas of individuals, teams, and organizations. It is the continuing conversations and evolution of processes and ideas that lead growth and change.

There has been some discussion in the devops community as to whether or not devops has lost its direction. Critics of the movement say that it is too defined by negative spaces, by people saying what devops isn't rather than what it is (or not providing a concise definition for it at all). They also claim that devops isn't unique, that it is merely a rebranding of ideas that have come before it, and that it will be abandoned as soon as the next name or trend comes along.

Many of the driving ideas behind the devops movement have indeed been around for some time under different names, but the zeitgeist of devops as something more than the sum of its parts is new and different. People have certainly argued against func-

tional silos before, suggested learning organizations, advocated for humane systems, or advocated for automation and measurement.

The devops movement is the first to combine all of these ideas, and to do so with **measurable success**. Harnessing and leveraging effective devops will lead to growth and evolution in tools, technology, and processes in your organization.

Devops Anti-Patterns

In this section, we will define some more terms that it will help you to be familiar with. Unlike the terms defined in the previous chapter, however, these concepts are generally seen as being anti-patterns, with ideas that run counter to those that give devops its strength and effectiveness.

Blame Culture

A *blame* (or *blameful*) *culture* is one that tends toward blaming and punishing people when mistakes are made, either at an individual or an organizational level. In this kind of culture, a root cause analysis as part of a postmortem or retrospective is generally misapplied, with a search for one thing that ultimately caused a failure or incident. If this analysis happens to point toward a person's actions as being the "root cause," that person will be blamed, reprimanded, or even fired for their role in the incident. This sort of culture often arises from one that must answer to external auditors, or where there is some top-down mandate to improve performance according to a given set of metrics.

Heavily segregated or segmented environments that lack an appreciation for transparency are fertile ground for blame culture. If management is set on finding one person or group of people to blame for each incident that occurs, in order to get rid of that "bad apple," individual contributors will be motivated to try to shift blame away from themselves and their own teams onto somebody else. While this sort of self-preservation is understandable in such an environment, it doesn't lend itself well to a culture of openness and collaboration. More than likely, people will begin withholding information about incidents, especially with regards to their own actions, in an effort to keep themselves from being blamed.

Outside of incident response, a culture of blame that calls people out as a way of trying to improve performance (e.g., which developers introduced the most bugs into the codebase, or which IT technician closed the fewest tickets) will contribute to an atmosphere of hostility between coworkers as everyone tries to avoid blame. When people are too focused on simply avoiding having a finger pointed at them, they are less focused on learning and collaboration.

Silos

A departmental or organizational silo describes the mentality of teams that do not share their knowledge with other teams in the same company. Instead of having common goals or responsibilities, siloed teams have very distinct and segregated roles. Combined with a blameful culture, this can lead to information hoarding as a form of job security (“If I’m the only one who knows how to do X, they won’t be able to get rid of me”), difficulty or slowness completing work that involves multiple teams, and decreases in morale as teams or silos start to see each other as adversaries.

Often in a siloed environment you will find different teams using completely different tools or processes to complete similar tasks, people having to go several levels up the managerial chain of command in order to get resources or information from people on another team, and a fair amount of “passing the buck”—that is, moving blame, responsibility, or work to another team.

The issues and practices that can come from organizational silos take time, effort, and cultural change to break down and fix. Having silos for software developers and system administrators or operations engineers, and trying to fix the issues in the software development process that came from that environment, was a big part of the root of the devops movement, but it is important to note that those are not the only silos that can exist in an organization. Cross-functional teams are often touted as being the anti-silos, but these are not the only two options, and just because a team serves only one function it’s not necessarily a silo. Silos come from a lack of communication and collaboration between teams, not simply from a separation of duties.

Root Cause Analysis

Root cause analysis (RCA) is a method to identify contributing and “root” causes of events or near-misses/close calls and the appropriate actions to prevent recurrence. It is an iterative process that is continued until all organizational factors have been identified or until data is exhausted. Organizational factors are any entity that exerts control over the system at any stage in its lifecycle, including but not limited to design, development, testing, maintenance, operation, and retirement.

One method of identifying root causes is known as the *5 Whys*. This method entails asking “why” until the root causes are identified. It requires that the individuals answering “why” have sufficient data to answer the question appropriately. A second and more systematic approach is to create an *Ishikawa diagram*. Developed by Kaoru Ishikawa in 1968, this causal diagram helps teams visualize and group causes into major categories to identify sources of variation, discover relationships among sources, and provide insight into process behaviors.

Often RCA is associated with determining a single root cause. Tools that provide event management often allow only a single assignment of responsibility. This limits

RCA's usefulness, as it focuses attention on the direct causes rather than the additional elements that may be contributing factors. There is an implicit assumption with root cause analysis that systems fail (or succeed) in a linear way, which is not the case for any sufficiently complex system.

Human Error

Human error, the idea that a human being made a mistake that directly caused a failure, is often cited as the root cause in a root cause analysis. With this often comes the implication that a different person would not have made such a mistake, which is common in a blame culture when somebody has to be reprimanded for their role in an incident. Again, this is an overly simplistic view, and is used prematurely as the stopping point for an investigation. It tends to assume that human mistakes are made due to simple negligence, fatigue, or incompetence, and neglects to investigate the myriad factors that contributed to the person making the decision or taking the action they did.

In a blameful culture, discussion stops with the finding that a specific person made a mistake, with the focus often being on who made the mistake and its end result. In a blameless culture or a learning organization, a human error is seen as a starting point rather than an ending one, sparking a discussion on the context surrounding the decision and why it made sense at the time.

Summary

Familiarity with these terms provides a more in-depth understanding of the rest of the material in this book. Combined with the foundational terminology and concepts in the previous chapter and the patterns and themes gleaned from history in the chapter prior to that, you now have a much clearer picture of the landscape that makes up devops in our industry today. With this foundation of understanding in place, we can now define and discuss our four pillars of effective devops.

The Four Pillars of Effective Devops

Patrick Debois has said that **devops is a human problem**, implying that every organization will have a devops culture that is unique to the humans within it. While there is no one “true” way of doing devops that will be identical for every organization, we have identified four common themes that any team or organization looking to implement devops will need to spend time and resources on.

Here are the four pillars of effective devops:

- Collaboration
- Affinity
- Tools
- Scaling

The combination of these four pillars will enable you to address both the cultural and technical aspects of your organization. It makes sense for your organization to focus on one or two pillars at a time while trying to make changes, but ultimately it is the combination of all four working together that will enable lasting, effective change.

It is important not to gloss over the first two pillars, which cover the norms and values of our cultures and interpersonal interactions, in favor of skipping straight to reading about tools. Effective tool usage is necessary for a successful devops transformation, but not sufficient—if that were the case, we could just provide a list of best practices for Chef or Docker and be done. However, resolving the interpersonal and interteam conflicts that arise within organizations is critical to fostering the lasting relationships that ultimately make a devops environment.

Collaboration

Collaboration is the process of building toward a specific outcome through supporting interactions and the input of multiple people. A guiding principle that shaped the devops movement was **the cooperation of software development and operations teams**. Before one team can successfully work with another team with a different focus, the individuals on a team need to be able to work with each other. Teams that don't work well on an individual or intrateam level have little hope of working well at the interteam level.

Affinity

In addition to the growth and maintenance of collaborative relationships between individuals, teams and departments within an organization and across the industry at large need to have strong relationships. Affinity is the process of building these interteam relationships, navigating differing goals or metrics while keeping in mind shared organizational goals, and fostering empathy and learning between different groups of people. Affinity can be applied between organizations as well, enabling companies to share stories and learn from each other as we build a collective body of cultural and technical knowledge within our industry.

Tools

Tools are an accelerator, driving change based on the current culture and direction. Tool choices can be perceived as easy wins. Understanding why they are wins, and their impact on existing structures, is important to prevent obscuring issues in teams and organizations. Failure to examine the problems in values, norms, and organizational structure leads to invisible failure conditions as cultural debt builds up. If tools, or lack thereof, get in the way of individuals or teams working well together, your initiatives will not succeed. If the cost of collaboration is high, not investing in tools (or worse, investing in poor tools) raises this cost.

Scaling

Scaling is a focus on the processes and pivots that organizations must adopt throughout their lifecycles. Beyond simply considering what it means to address devops in large enterprise organizations, scaling takes into account how the other pillars of effective devops can be applied throughout organizations as they grow, mature, and even shrink. There are different considerations, both technical and cultural, for organizations operating at different scales.

About the Authors

Jennifer Davis is a global organizer for **devopsdays** and a local organizer for devopsdays Silicon Valley, and the founder of **Coffeeops**. She supports a number of community meetups in the San Francisco area. In her role at Chef, Jennifer develops Chef cookbooks to simplify building and managing infrastructure. She has spoken at a number of industry conferences about devops, tech culture, monitoring, and automation. When she's not working, she enjoys hiking Bay Area trails, learning to make things, and spending quality time with her partner, Brian, and her dog, George.

Ryn Daniels is a senior operations engineer working at Etsy. They have taken their love of automation and operations and turned it into a specialization in monitoring, configuration management, and operational tooling development, and have spoken at numerous industry conferences including Velocity, devopsdays, and Monitorama about subjects such as infrastructure automation, scaling monitoring solutions, and cultural change in engineering. Ryn is one of the co-organizers of devopsdays NYC and helps run Ladies Who Linux in New York. They live in Brooklyn with a perfectly reasonable number of cats, and in their spare time enjoy playing cello, rock climbing, and brewing beer.

Colophon

The animal on the cover of *Effective DevOps* is a wild yak (*Bos mutus*). This formidable yet friendly bovid occupies remote, mountainous areas of the northwestern Tibetan Plateau, the highest dwelling of any mammal.

The wild yak's distinguishing characteristics include high, humped shoulders and shaggy, dark fur that hangs nearly to the ground. It is one of the largest members of the *Bovidae* family, which also includes the American bison, the African buffalo, and domestic cattle. Males can reach between five and seven feet high at the shoulder and weigh up to 2,200 pounds; females are typically a third of that size.

Bos mutus is extremely well adapted to its high-altitude habitat, with a large lung capacity, high red blood cell count, and warm, woolly coat. In spite of their bulk, wild yaks are also very adept climbers, using their split hooves and strong legs to navigate rocky, icy terrain. Their dense horns, which curve out from the sides of their broad heads, allow them to dig through snow for food. Conversely, they are very sensitive to warm temperatures and move seasonally to avoid the heat.

Wild yaks are gregarious and peaceful herbivores, feeding on grass, herbs, and lichens. Female yaks gather with their young in herds of as many as 100; males tend to be more solitary, traveling in groups of around 10. Together they travel long distances to obtain vegetation.

Though the yak population is thought to number over 12 million worldwide, that figure mostly comprises the smaller domesticated yak (*Bos grunniens*). The wild species is considered vulnerable, reportedly declining more than 30 percent in the last 3 decades. Its most serious threat is poaching, although interbreeding with domestic yaks is another factor. The wild yak's average lifespan is about 23 years in the wild.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from *Lydekker's Royal Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.