NEW AND UPDATED

ADVANCE COPY

2ND EDITION

The IT Manager's Guide to DevOps

How to Drive the Business Value of Software Delivery

The IT Manager’s Guide to DevOps
How to Drive the Business Value of Software Delivery
October 2018

Authors: Tim Buntel and Robert E. Stroud
ISBN/EAN: 978-0-359-09887-3

IN MEMORY OF ROBERT E. STROUD

Rob, thank you for your vision and leadership—but mostly, for your friendship. You were a guiding and inspiring force not only in the creation of this book, but to the entire DevOps community.

CONTENTS

FOREWORD

FOREWORD BY: DAVID FARLEY

Continuous Delivery and DevOps are at the leading edge of software development thinking. Most of the literature and discussion focuses on the technical aspects, but this is a process that crosses boundaries and helps to make businesses more efficient. This book is aimed at the IT manager or business person who wants to understand the Continuous Delivery and DevOps approach.

Software development is an extremely difficult undertaking. Software is unlike most endeavors with which we are familiar. Creating software is a technically demanding process. It often demands advanced problem-solving techniques and high levels of creativity. On top of all of that, software is remarkably fragile: a tiny error, one wrong character in the equivalent of a chapter, is enough to prevent it from working.

As a result of these demands, the software industry has often struggled to effectively meet the needs of the businesses that it serves. It has been far too common for software development projects to be late, deliver poor quality, and deliver software that doesn't meet the needs of users. There have been many attempts to tackle this problem, but over the past few years there has been a significant change. Learning from the scientific method, advanced quality-focused processes and experiments in software development processes we have evolved techniques that we can finally claim, with some authority, work.

Continuous Delivery is the most effective approach to developing high-quality software that we have found so far. This is the process that some of the most successful companies in the world now use to gain market advantage. The early adopters were web companies, including some of the largest and most successful, but increasingly this process is now being adopted in all types of organizations, developing all kinds of software and in widely different business sectors. This adoption is happening because this process works! Businesses that have adopted Continuous Delivery and DevOps are more successful as a result. They get higher-quality software into the hands of their users and customers faster than before, and so can react to business demand and change more rapidly.

Continuous Delivery works because it depends on a more scientific approach to development. We attempt to establish feedback mechanisms that show the results of our work more quickly. We aim to make our software more verifiable at all levels.

Continuous Delivery changes the way that you and your software development teams work together. This is not a process that stops at the boundaries of the software development team. It helps to foster a professional, high-performance team culture. It frees each group to make the professional decisions for which they are trained and yet encourages them to interact more and depend on one another’s skills and expertise.

I believe that in any market, a company that uses DevOps and Continuous Delivery for software development has a significant advantage over those that don’t. This is not because of the technicalities, but because it provides a business the freedom to experiment and so to react to change. If you work for a company that relies on software development, these changes are going to affect you. If you adopt them, they will change the way that you work, and improve the efficiency, effectiveness, and quality of your software and your business. If you don’t, you will see increasing competition from those who do.

I hope you enjoy this book, and I hope that you enjoy the process of discovery and the sense of teamwork that adopting Continuous Delivery and DevOps so often initiates.

David Farley
Co-author of *Continuous Delivery*

FOREWORD BY: GENE KIM

I wholeheartedly agree with everything Dave Farley wrote in the foreword. As co-author of the seminal 2009 book, *Continuous Delivery*, few people are more qualified than him to write on the topic.

I'm sure that Mr. Farley would agree that Continuous Delivery and other technical practices are critical to IT performance, and in turn, organizational performance. Indeed, this is a decisive finding over the last five years of the State of DevOps Report, research I'm proud to be a part of.

Some say the goal of science is to confirm deeply held intuitions and reveal surprising insights—I am so delighted that the State of DevOps research has done both. We certainly expected to discover that technical practices matter, but I am always amazed by how much they matter. In particular, the need for short-lived branches and trunk-based development as a way to achieve was one of the biggest surprises in this journey for me.

We now know that technical practices also reduce deployment fear, increase deployment success rates, reduce the MTTR when things go wrong, improve the ability to achieve security objectives, and significantly increase the likelihood of engineers recommending their organizations to friends and colleagues as a great place to work.

One of the lessons I learned from the Software Engineering Institute was that "the best are always getting better and accelerating away from the herd." Our research shows clearly this phenomenon, but also that the community is increasingly adopting DevOps into their daily work—in 2018, an astonishing 48% of the survey population are now deploying daily, with code deployment lead times between one day and one week.

Never has it been more evident that DevOps is not just a competitive differentiator, but also needed to survive.

Gene Kim
Portland, OR
September 2018

CHAPTER 1

INTRODUCTION

INTRODUCTION

Today, all great companies are software companies. Banks are software companies. Athletic apparel companies are software companies. Industrial manufacturing companies are software companies. No matter the product or service you offer, to win you need to be a software company. You may not sell something called "software," but successfully making, marketing, and delivering any product today relies on software and technology, and a "software-focused" mind-set. Software allows you to create more value for your customers faster, and it lets you beat the competition. It allows you to make your corporate processes more efficient, opens new channels to reach your market, and improves the flow of innovation within your organization.

But to benefit from being a software company, you need to be fast, reliable, and secure. Your first challenge is to get features "out there" quickly and at high levels of quality. Then you must rapidly collect feedback from your users to guide your next set of ideas. In larger organizations, this needs to scale for thousands of employees and applications. This cycle of delivering value through software and iterating rapidly to reflect feedback powers the most successful organizations.

First, deliver features better. Then, deliver better features better!

DevOps: A Business-driven Approach

DevOps is a set of cultural philosophies, processes, practices, and tools that radically removes waste from your software production process. It aims to create a culture where development, operations, and all other stakeholders work together, rather than being in isolated silos. It enables faster delivery of high-quality functionality and sets up a rapid and effective feedback loop between your business and your users. It's also not new. DevOps involves concepts that have been in place in high-performing organizations for years. It's not just about source code and servers, either. DevOps is about meeting the needs of your business. In a competitive economic environment, every organization should look to DevOps to better deliver value to customers and outpace the competition.

DevOps helps you reduce your time to market from months to weeks and weeks to hours or minutes!

DevOps or Continuous Delivery?

Many business leaders have heard the terms DevOps and Continuous Delivery used in seemingly similar contexts. Both involve rapid, regular delivery of software. Both involve teams working together in a way that differs from organizations using traditional software development processes. But they're not the same thing.

DevOps is a philosophy and a business-driven approach to delivering software solutions. It's a portmanteau of "development" and "operations," and it seeks to break down the barriers that traditionally existed between those parts of the development process. The main objective of DevOps is a culture and environment where teams work collaboratively to deliver software frequently and reliably. Culture and organizational dynamics are at the center of a DevOps transformation.

Continuous Delivery (commonly called CD) is an essential part of putting DevOps into practice. CD provides a way for changes to software to be released as they're made, in a fast, reliable manner, with little risk and little need for human intervention. It uses Agile Software Development methods and emphasizes the importance of automation wherever possible. While most conversations about DevOps begin with a discussion of people, CD most often centers around tools. CD leverages tools for Continuous Integration (CI), Application Release Orchestration (ARO), test automation, and more.

So, it's not a question of "DevOps" or "Continuous Delivery." Both are agile, both require healthy collaboration between all contributors to the development process, both are focused on bringing value to customers. They fit together perfectly as part of the goal of improving the process of delivering value to customers. Statements like "a key part of our DevOps transformation was setting up a CD pipeline..." show how these two parts combine.

Several other terms and acronyms are sometimes used for Application Release Orchestration (ARO). You may also hear it referred to as "Application Release Automation" (ARA) or "Continuous Delivery and Release Automation" (CDRA).

CHAPTER 2

MEET OSTRICH INSURANCE: THE TRADITIONAL COMPANY

DevOps can dramatically shrink the time it takes to bring customer value from idea to production. In this chapter, we illustrate some common problems that exist in a traditional software delivery process where software delivery often takes many months or years.

A LOSING STRATEGY

Ostrich is a traditional company that has been doing things a certain way for a long time. Initially, they scoffed as newer competitors came onto the scene with strong software-driven features and online sales. After all, they view themselves as an insurance company, not a technology company. In time, the competitive pressure grew, and Ostrich shifted to selling their products online. However, it is hard for them to compete in the online market because it simply takes them too much time to deliver new features and software. They're still stuck in planning meetings when their more agile competitors have already launched valuable, differentiating features.

Communication between business, development, QA, and operations takes place on many different levels and often leads to confusion. Handoffs between the different groups involved in the overall process introduce long waiting times and provide yet further opportunity for hesitation and disruption.

Slow, error-prone manual processes, delays, handoffs, and lengthy fix cycles inevitably lead to infrequent releases. And when large numbers of changes and fixes are infrequently released in a large batch, the go-live is always a time of high stress and is often followed by days or weeks of post-release emergency patches.

How can the business ever release features quickly and reliably under these circumstances?

VALUE STREAM MAP—OSTRICH INSURANCE

Identifying Ostrich's challenges can be done with the aid of a Value Stream Map for their software delivery process. Value stream mapping is a proven method for identifying and progressively eliminating waste in a process.

1. Requirements: We Know Everything

Business analysts are conditioned to work with large, infrequent releases, so they try to specify the complete feature set up front. To minimize ambiguity and differences in interpretation, they create many long and detailed descriptions that capture all use cases and scenarios the business can imagine. But in most cases, imagination is used in place of the customer's real wants and needs. Guesses and assumptions lead to many features that fail to meet customers' expectations.

In terms of delivering business value, this approach creates an early choke point causing time-to-market delays because a large collection of features must be gathered and documented before work on a single feature can even start.

Phase Duration: weeks to months

Waste: wrong features specified; features not specified correctly; features specified to an excessive level of detail; no customer feedback cycle in large up-front product design; no advice from developers to design for feasibility

2.Coding: Postponing Problems

The specifications of the feature requirements are assigned to one or more development teams one by one. Developers work in isolation, each making changes to their copy of the application code without being aware of any possible overlapping changes made by their colleagues. Only after several weeks is all work related to a particular feature complete.

Duration: several weeks

Waste: duplicated effort; lack of communication between developers

3. Integrating: Merge Hell

After weeks of working on an isolated copy of the code, the new features need to be combined, or "merged," into a single codebase. The longer the team puts off merging the code and the larger the development team, the greater the number of conflicting changes that need to be handled. Late and infrequent integration is a typical recipe for missing deadlines and introduces significant risk.

Duration: days

Waste: conflict resolution during merging; problem needs to be solved by the entire team

4. Installing: Error-prone Manual Deployment

After weeks of manually tweaking the development machines to get the application to run, the development team now needs to produce a document that the operations team can follow to deploy the new version to subsequent test environments. This “installation manual” usually describes all the steps that the developers remember, but it certainly does not describe a fully tested procedure.

Development hands off the installation manual and application binaries to operations via a shared network drive. A support ticket is created for operations to deploy the application to the test environment in the next available timeslot.

Just before the deadline of the service level agreement for handling the support ticket expires, an operator picks up the request and attempts to deploy the application according to the steps outlined in the manual. After a couple of steps, the operator runs into an error, adds a comment to the ticket, and passes it back to the development team.

The next morning, the QA team that is expecting to start a test cycle discovers that the application isn’t even running. The lead developer checks the support ticket, discovers a typo in the installation manual, uploads the updated manual to the network share (obscuring the audit trail in the process), and hands the ticket back to operations.

Eventually, the operator resumes work on the ticket and finally completes the installation.

Duration: 1 week

Waste: waiting for deployment to be carried out; fixing incorrect deployment instructions; fixing incorrect deployments; waiting for the target environment to become available; QA waiting for the application to become available for testing; “ping-pong” between different teams

5. Testing: Lowering the Bar After Manual QA

The QA team can now start their acceptance testing. During the development phase, the team translated the detailed business requirements into extensive test plans. Some of the requirements seem to be good candidates for the test automation tool used at Ostrich. For these types of requirements, the team tried to translate requirement descriptions into automation scripts that the tool can run. Like all translations, this is a challenging process resulting in many requirements incorrectly or insufficiently covered by automated tests.

The lengthy process of executing the test plan gets underway. On completion, the QA team prepares its report, which identifies 15 failing test scenarios—enough for the release to be put on hold. With many important features threatening to be delayed by the 15 failing tests and pressure from the business mounting, it is decided that only 3 of the defects are indeed showstoppers that must be fixed before continuing. A patch is quickly developed, the 3 scenarios are verified, and the updated version is reluctantly passed by the QA team. The release manager now creates a support ticket to release the patched version to production. This means that 12 of the original requirements are of poor quality and need to be addressed in an urgent maintenance release to follow the official release.

Duration: 3 weeks

Waste: translation of requirements into tests; waiting for test completion before fixing; developing features that do not work; testing scenarios that are determined to be non-essential; many correct features waiting for 3 defects to be fixed; miscommunication caused by differing interpretations of requirements; handoffs between different teams; later releases delayed by maintenance releases

6. Security Testing: The Dreaded Late Roadblock

Years of headlines about high-profile security vulnerabilities at other companies who deal with sensitive data has made the executive team at Ostrich very cautious about the security of their software. A full static and dynamic application security test is recommended, followed by a penetration test performed by a third party. But the AppSec team consists of only 3 security professionals supporting 350 developers company-wide. A full scan can take hours to perform and even longer to analyze the results, weeding out false positives.

Duration: days or weeks

Waste: false positives produced by testing tools waste time

7. Release Preparation: The Pre-production Bottleneck

Before the new version can be released to production, it must pass a final set of tests in the pre-production environment. These tests are required to increase the confidence level that the go-live will succeed, since the acceptance environment differs quite substantially from production.

Getting an exclusive slot for the busy pre-production environments must be planned well in advance. It is not uncommon for a wait of as much as 2 weeks for the next available slot.

Once the application is installed in the pre-production environment, the QA team starts the production integration, stability, and regression tests. After a week, the team finds two errors, which the development team fixes 2 days later. After a short retest of the failed scenarios, the release is accepted for production.

Duration: 4 weeks

Waste: waiting for the pre-production environment to become available; missed deadlines; errors found late in the process

8. Go-live: The Quarterly Release Cycle

The new application version has been approved just in time for the next available quarterly production release slot, happening in 3 weeks' time. As usual, it will be busy and stressful: from Saturday night to Sunday, the operations team is scheduled to manually deploy 9 new application versions. After 4 hours of manually executing the installation steps described in the various manuals, the QA team takes another 4.5 hours to verify the releases. Moments before the go/no-go deadline is reached, QA completes testing, and—with many sighs of relief—the application is live.

Duration: 3 weeks

Waste: waiting for the release window; out-of-hours work

Before this entire process began, the business identified a set of features that their customers would find valuable. It is now months later, and still only a small subset of those are in customers' hands. Important fixes are outstanding, and it will be difficult or impossible for Ostrich to gauge the impact of each of the new features so they can better address their customers' needs in the future.

9. Operating, Scaling, and Monitoring

The live application receives moderate traffic, and everyone quickly moves on to the next project. But as the end of the quarter approaches, a large spike in usage starts to cause problems. One ops engineer is on pager duty over the weekend and starts to get notifications that the servers are beginning to fail due to the spike in load. The escalation process is manual, relying on phone calls and text messages to try to find anyone who can address the problem over the weekend. Developers are unavailable to answer questions about a database error that has only appeared now that the application is under real-world load.

Duration: ongoing

Waste: inability to respond to variable demand; lost time responding to problems

Tl;dr[1]
A traditional software delivery process prevents you from responding to your customers' needs quickly and reliably:

- Individual features or requests cannot be quickly implemented.
- Delivery process is slow, unreliable, and error prone.
- Quality levels are low due to late discovery of errors.
- Security is an afterthought, if considered at all.
- No feedback loop is available to measure a feature's impact for customers.
- No ability to easily scale capacity up or down, or to effectively manage incident responses.

---

[1] *In online posts and news articles, "tl;dr" is often used to denote a summary associated with a long post. We'll include a summary after each major section in this guide. Tl;dr is an abbreviation for "too long; didn't read," but we hope you do read the full section and find the summary helpful to reiterate the main points.*

CHAPTER 3

MEET THE COMPETITOR: GAZELLE INSURANCE

The previous chapter showed how a traditional software delivery process prevents organizations from reacting quickly and reliably to market demand. In this chapter, an organization uses DevOps to enable the rapid delivery of high-quality features to production.

A WINNING STRATEGY

If you ask the CEO of Ostrich Insurance which competitor worries him most, he'll immediately say Gazelle Insurance. Gazelle had been a small regional competitor, but they rapidly expanded and are now on track to pass Ostrich within the next 24 months. It all started 18 months ago when an energetic and hungry new CEO took the helm and began to think about what they truly offered the market. She saw that insurance was about more than just the details of the policy; it was also about the ease of shopping for that policy, the convenience of filing and seeing the status of claims, the quality of communication between customers and the team, and more. Most importantly, she recognized that all those things relied on software. Software was what would allow Gazelle to create more value for their customers faster, and it would let them beat less nimble competitors (like Ostrich).

The new leadership team at Gazelle set about transforming into a software company. They had to acknowledge early that it was unacceptable to deliver new products or update product features only occasionally and at long-spaced intervals. They needed to continuously work to shrink the time from business idea to delivery to customers. And they needed to constantly measure the benefit that the features delivered for customers, so they could always improve and adjust to meet their needs.

The Gazelle IT team examined their software delivery value stream and pinpointed every step in the process that was costly, time consuming, or prone to errors. They removed, automated, or accelerated every single step until they had put in place a rapid, cost-efficient, and reliable process. This transition was based on the practices and principles of DevOps and Continuous Delivery.

At Gazelle Insurance, the feedback cycle between the business and the end user is incredibly tight, enabling a rapid "dialogue" with customers that ensures each idea and feature delivered aligns with the current needs and wishes of Gazelle's users.

For Gazelle, it's not just about delivering ideas faster and more efficiently. It's about enabling the business to reliably have better, more relevant, and more timely ideas in the first place. While there are significant technical differences in how Gazelle and Ostrich develop software, it is this business-value-driven approach that really contrasts the two companies. At Gazelle, the business never relies on guesswork to drive new features. Instead, they identify improvements based on analysis of user responses by looking at user interaction data, system metrics, social media output, and comparisons with groups of users exposed to a different subset of features—all delivered constantly through live dashboards and visualizations. Based on this analysis, the business can be truly customer-centric and data driven for decisions about new features to try, enhancements to make, or features that fail to add value and can be eliminated.

VALUE STREAM MAP—GAZELLE INSURANCE

1. Collaborating: End-to-End Feature Teams

At Gazelle, new ideas are formulated and prepared for inclusion in the delivery stream by an end-to-end team that involves all relevant groups as soon as possible: business analysts, developers, testers, and operators. This approach ensures that everybody has a shared understanding of an idea and can implement this idea effectively. This agile team covers the entire spectrum of service delivery and is involved in making the product a success at every step of the way.

2. Validation: Skip the Endless Debates

The cross-functional feature teams know that users ultimately decide the value of a feature. So they look to answer key business questions through design, prototyping, and testing ideas with actual customers before making a commitment to costly development. Using short, focused exercises like design sprints, the team can experiment with feature design and measure customer reactions early. Validating features with users eliminates endless debates among the team where different members all claim to know "what the customer really wants." When every member of the team gets first-hand exposure to customers, they bring knowledge back to their colleagues that takes much of the guesswork out of development. It's not uncommon at Gazelle to hear one engineer say to another, "it was amazing to see how the user actually uses the feature: if we build it this way they'll really love it!" Everyone on the team understands why the work delivers value to customers. Finally, the artifacts created—often high-fidelity prototypes—can make subsequent specification steps much easier and faster.

Duration: 5 days

3. Defining: Specifications Become Automated Acceptance Tests

Even in the most agile or lean approaches, a change should be specified before it is implemented. In contrast to the lengthy documents common at Ostrich, specs at Gazelle are formulated by the entire delivery team, with the business collaborating with QA to write executable tests in business-level language. This increases efficiency and eliminates a common source of miscommunication. Rather than first writing a specification on paper and later requesting that someone else translate it into code for a test tool, both parties work together at one time to define a single desired outcome that can be tested.

Duration: ½ Day

4. Coding: Tests First, Then Code

Test-driven development (TDD) has led to higher-quality code at Gazelle. Before a developer starts implementing a feature, they write a failing unit test for the piece of code they are about to write. Only then does the developer start writing code to make the test succeed. This practice, which is followed by all other developers working on the application, has resulted in a large body of unit tests validating that all the pieces of code continue to behave as intended.

Since all these unit tests still pass after the new code has been added, the change is checked into version control. By doing so, the change is added to the same codebase that all other developers on the team are working on. The updated codebase is then verified again by the Continuous Integration server to check that the new code does not conflict with changes made by other developers. If this build fails, the entire team is immediately notified, and correcting the mistake to “get the build back to green” becomes the top priority for the team. Any member of the team can fix, or even simply remove, the new change to correct the mistake in minutes.

Duration: ½ Day

5. Off It Goes: The Continuous Delivery Pipeline

The integration build is the first step a new feature takes on its journey through the Continuous Delivery pipeline. It verifies that the feature passes the automated acceptance tests that were already created in the specification phase. It also ensures that the change meets requirements for security, performance, availability, and so on.

These tests are carried out in automated “hands-off” environments using containers. Test systems are created and torn-down as needed, without human intervention. The feature is automatically deployed onto these environments, and then all applicable tests are immediately executed.

The speed and reliability of automatically provisioning and configuring the production-like test environments is critical to the pipeline. It allows Gazelle Insurance to accelerate throughput by performing many types of tests in parallel, raising the team’s confidence that the high level of quality verified in the pipeline carries over to production.

As each feature moves through this pipeline, everyone with a stake in its delivery—from business owners to development, and from QA to operations—can immediately see its status. There is no confusion about what is in a release, and no one needs to ask, “did my feature make it in?” Compliance is also much easier to manage because a complete common audit trail is always available showing exactly what happened when. Gazelle avoids all the confusion that is so common among the teams at Ostrich with their frequent frantic status update calls and he-said/she-said situations that arise due to poor communication.

Duration: hours or minutes

6. Go-live: The Product Owner Hits the Button

If a new feature causes no failures in the pipeline, the business owner is notified and asked to approve the new version of the application. This approval is mostly a formality as the feature has already passed thousands of automated tests, which ran overnight. A quality dashboard shows a summary of all test results for review and drill-down if desired.

By approving the new feature, the business owner adds it to the next fully automated production deployment. These deployments take place at regular intervals around the clock, with no developer or operator required. Deploying a new feature is as easy as publishing an article in a content management system. For many applications at Gazelle, the business trusts the tests to the point that any feature that makes it through the pipeline is automatically approved.

When initially released, a new change is only exposed to a small percentage of the customer base in production. If significant problems are detected in this “canary” subset, the change is automatically rolled back. If all goes well, it is automatically released to the entire user base.

Duration: 1 hour

7. Intelligently Operating, Scaling, and Monitoring

The container architecture that Gazelle uses to package and run instances of its application are managed by a container orchestration platform. The platform includes an autoscaler that automatically scales the number of instances of the application based on CPU utilization and other metrics that the team defined. If an instance goes down for some reason, a new one is created automatically ensuring high availability in addition to high performance. The team also instrumented the application with runtime analytics to measure performance and monitor for security problems. These measurements are integrated with the entire team’s incident response system, so there’s never a single point of failure when a problem does arise.

Duration: immediate and automated

Tl;dr
A DevOps process means customers receive value through better quality software delivered faster:

- Cross functional teams make sure everyone is on the same page on both the “why” and “how” of delivering valuable features to customers, and which specific features will ship in a given release.
- Early validation gives the team an opportunity to measure the proposed solution’s customer fit before investing in high-cost development.
- TDD ensures higher-quality features that always have quality tests available.
- CD pipelines ensure repeatable processes and promote visibility and compliance.
- Self-service and automation empowers value to be delivered when it’s ready.
- Containers and orchestration platforms allow high performance, high availability, and scalability. They offer excellent visibility into problems before things get out of hand.

CHAPTER 4

DEVOPS ISN'T (JUST) A TECHNICAL THING

DEVOPS ISN'T (JUST) A TECHNICAL THING

The Role of the Business

In traditional software development environments, the business is kept, or chooses to keep, far away from the feature development process. The relationship with dev is transactional, where the business "throws a bunch of requirements over the wall to IT" and waits to see what comes out some months later. It's the exact opposite in DevOps. The business is an essential central participant throughout the entire process, from idea to launch to enhancement.

Most business stakeholders love this idea and get excited about the opportunity to drive the process. But it comes with significant new responsibilities.

- **Blameless Culture (No Blame Games):** When problems occur in traditional processes, blame is usually the first thing to appear. The business blames dev for failing to deliver, while dev blames the business for omitting detail or lacking clarity. In DevOps, every member of the team shares in the success, and every member of the team shares when things don't go as planned. DevOps recognizes that mistakes happen—the key is to identify them as early as possible and address them without delay.
- **A Filled Seat:** Having a seat at the table only works if someone is sitting in it! In DevOps, the business needs to be prepared to commit full-time resources—dedicated Product Owners—to gather and evaluate feedback from the user community, work with testers and developers to refine new features and improvements, and define specifications in the form of automated acceptance tests.
- **A Full Pipeline of Value:** DevOps means better features faster, driving business differentiation and value. The fast and efficient development pipeline needs to be fed continuously with new features and ideas. This is an area where there's risk that the business itself can become a bottleneck. Early management of expectations and communication of the new responsibilities to the business are essential to realizing the potential of DevOps. Otherwise, the full potential of the well-oiled machine that is the new CD pipeline will go unrealized.

The Role of Culture

You don't "do" DevOps with just a tool. DevOps is about people working together in a culture where collaboration is central to delivering value. It's about breaking down barriers between groups. In DevOps, development and operations collaborate through a sense of shared responsibility. In traditional development models, individuals focus on their own small part of the process and care little what happens after they hand their piece off to the next stage and its team. But DevOps' spirit of shared responsibility means that every member of the team stays involved with the solution over its entire lifetime.

Physical changes can facilitate cultural changes too. Co-locating cross-functional teams, from dev to ops and beyond, enhances their sense of shared responsibility. These are real people, not just some random team on another floor of the building or another city.

Dealing with Holdouts

It's not uncommon to find critical individuals in an organization who resist the cultural and other changes that come with a DevOps transformation. They may be cynics who see DevOps as "just another trend" that management is foisting on them. Or they might be fearful of what the transformation means to them and their career. There's a huge difference between "I am a developer" and "I am an ops person." In many cases, an individual's self-identity has been developed over years of professional life. That can't be expected to change overnight just because of a proclamation that they're now "doing DevOps."

Once they're identified, it's important to offer holdouts lightweight, easy to implement, non-destructive/non-threatening approaches to align them with the efforts and—ultimately—provide value to the transformation. And it is important to share the results of the effort with them as it progresses. Everyone wants to deliver better software with speed and stability. There are few better ways to quiet the skeptics than with evidence that the entire team's efforts are delivering that result.

Tl;dr

- DevOps isn't something achieved using a tool. It requires a cultural shift with buy-in from everyone involved.
- The role of the business is greatly increased in DevOps compared to traditional development approaches.
- Increased involvement means increased responsibility for the business.
- A DevOps transformation may not be welcomed by everyone at first. Understanding the holdouts and why they object helps to win them over.

CHAPTER 5

INTRODUCTION TO A DEVOPS PLATFORM

INTRODUCTION TO A DEVOPS PLATFORM

In a DevOps organization, Continuous Delivery is used to create a constant flow of changes to production. The entire automated software production line is commonly referred to as the Continuous Delivery pipeline. The CD pipeline breaks the software delivery process down into several stages, each of which further verifies the quality of new features before moving them along to the next stage until they finally are available to users, with little to no errors. The pipeline provides feedback to the team and visibility into the flow of changes to everyone involved in feature delivery. To successfully scale DevOps in an enterprise, a set of components are leveraged to help teams achieve CD. This is called a DevOps platform.

While there may be slight variations from organization to organization, or within the type of software project, a typical DevOps platform facilitates Continuous Delivery through the following stages:

1. The Initial Stages: Build Automation and Continuous Integration

New features that the developers implement start their journey through the CD pipeline as code commits to a central code repository. The pipeline starts by building the binaries to create the deliverable(s) that are passed to the subsequent stages. Each developer's code commits are integrated into the central code base on a continuous basis where they are built and unit tested. This is called Continuous Integration. It provides an early opportunity for clear feedback about the "health" of the new application code. If something is wrong, the build will "break," and the cause is clearly identified. The developer who introduced the problem code can commit a fix, and everyone can see the build "go green" as it passes the phase.

Common tools you may hear mentioned:

- Git
- Jenkins
- Bamboo
- CircleCI
- AWS CodeBuild
- VSTS

2. The Verification Stages: Test Automation

The earlier problems are detected, the easier (and cheaper) it is to fix them. It is important that the pipeline verifies all relevant aspects—whether user functionality, security, performance, or compliance. Manual testing can identify some problems, but it is time consuming, costly, and error prone. High-performing DevOps teams use a combination of manual and automated testing to ensure an increasingly well-tested application as it moves through the pipeline.

Common tools you may hear mentioned:

- Selenium
- Gatling
- JUnit
- Cucumber
- JMeter
- Fortify
- Veracode

3. The Rollout Stage: Deployment Automation

The final stage of the pipeline is deployment to production. Since the preceding stages have verified the overall quality of the system, this is now a low-risk step. The deployment can be staged, with the new version being initially released to a subset of the production environment and monitored before being completely rolled out. The deployment is automated, allowing for reliable delivery of new functionality to users within minutes whenever this is needed.

Common tools you may hear mentioned:

- XebiaLabs DevOps Platform
- IBM UrbanCode Deploy
- CA Automic
- Octopus Deploy
- AWS Code Deploy

4. The Live Stage: Operating, Monitoring, and the Feedback Loop

The image of a pipeline is somewhat misleading since it implies a distinct end. In a DevOps platform, things that happen after the application is deployed to production provide feedback to the team so that the entire cycle can begin again with fixes to problems, enhancements, or ideas for new features. So it's really more of a loop than a pipeline.

At this stage, the DevOps platform is concerned with ensuring that users have a good experience with the application. This means making sure the application is stable and fast, and that it's being used in a way that adds value for the user. There are many tools to help with this. Some monitor running applications to alert and take action when a problem arises, some automate the process of scaling application instances up and down and reacting if instances become unavailable, and some provide a way to observe and interact with users while they are using the application.

Common tools you may hear mentioned:

- ELK Stack
- New Relic
- Splunk
- Google Analytics
- Mixpanel
- Intercom.io
- Drift

5. Orchestrating It All: Release Coordination and Automation

Cross-functional DevOps teams include many participants in every stage of the pipeline, each with many tasks (both automated and manual) to be performed. Keeping track of it all can be challenging. Release coordination provides a top-level view of the entire pipeline. It defines the overall software delivery process and puts appropriate controls in place for each stage. Analyzing this software delivery value stream highlights any inefficiencies and hot spots that can be improved for further optimization of the pipeline. Tools can help teams identify bottlenecks, reduce errors, and lower the risk of release failures. You may hear such tools referred to as Application Release Automation, Application Release Orchestration, or Continuous Delivery Release Automation.

Common tools you may hear mentioned:

- XebiaLabs DevOps Platform
- IBM Urbancode Release
- Spinnaker

CD in the Cloud

A DevOps platform supports traditional on-premises deployments for all kinds of applications, from mainframe to web. But it also works extremely well for applications deployed in the cloud, be that public, private, or hybrid. Many public cloud platforms, like Amazon AWS, Google Cloud Platform, and Microsoft Azure, offer their own CD pipelines that deliver source, build, test, and deployment in a single service. There are some tradeoffs between convenience and flexibility that need to be considered, but for some simpler scenarios that are focused on a single vendor ecosystem, these services can be very appealing.

Common tools you may hear mentioned:

- AWS CodeBuild
- CodePipeline
- Spinnaker
- Google Container Builder
- Visual Studio Team Services
- Red Hat OpenShift
- Cloud Foundry
- XebiaLabs DevOps Platform

Containers and Microservices

Microservices is an architectural pattern where applications are delivered as a set of loosely coupled services instead of as one large monolithic application. Containers are a good way to deploy microservices. Containers are a complete, lightweight runtime environment containing an application, plus all its dependencies, libraries and other binaries, and configuration files needed to run it, bundled into a single package. The way you deploy applications is different when you use containers, but the CD pipeline still exists.

Common tools you may hear mentioned:

- Docker
- Kubernetes
- Helm
- Mesos
- XebiaLabs DevOps Platform

Tl;dr

- A DevOps platform gives teams the intelligence, automation, and control necessary to perform DevOps at enterprise scale.
- The Continuous Delivery pipeline enables a continuous flow of changes to production in a DevOps organization.
- Functionality is only added when the quality is right.
- All changes to the source code immediately result in a new version of the application.
- Each new version is automatically tested against all available tests.
- New versions are automatically deployed to production.
- Installation and configuration of machines and environments is fully automated.
- DevOps doesn’t end when the application is live: monitoring and feedback loops help drive the next iteration of improved value for customers.

CHAPTER 6

DESIGNING YOUR PIPELINE

DESIGNING YOUR PIPELINE

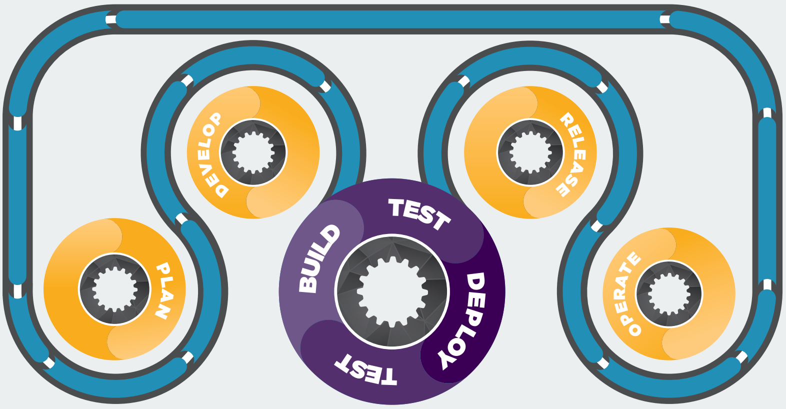
The previous section described various phases in the CD pipeline. The pipeline breaks the software delivery process into phases. While those are typical phases, each individual pipeline consists of its own phases and tasks. Teams design pipelines that reflect their own process, both as it is presently and as it evolves through additional automation or other process improvements.

The pipeline defines the path that the software takes on its way to release, represented as blocks of work that happen in succession. Pipeline design begins by breaking the entire development process into a set of tasks, which are logically grouped in phases. The entire team should work together to discuss and capture the phases and tasks that make up a pipeline so that equal attention is paid to all parts of the process. It can be helpful to represent the pipeline visually—either in a document, or, ideally, with an application release orchestration tool.

When designing a pipeline keep the following questions in mind:

- What tasks need to be run sequentially? Which can be run in parallel? Parallel tasks can improve efficiency and remove bottlenecks in the process. For example, automated acceptance tests can run while awaiting a manual approval for user acceptance.
- What tasks need to happen each and every time? Which are optional? This can be particularly important for compliance purposes in regulated industries.
- What happens when a task fails? Does the entire release fail? Should just that phase restart?

The following example shows a common visual representation of a CD pipeline. It begins when any developer commits code changes to the source code control system (SCM). This initiates a number of manual and automated tasks across multiple phases, from building and testing, to deployment and verification.



Pipelines can also capture phases and tasks across multiple releases. A common example of this is in the microservices architectural pattern where one solution (a mobile banking app, for example) may be composed of multiple smaller applications (a payment service, an authentication service, and so on). In this case, a “master” pipeline may start several “subreleases” and then wait for them to finish.

DevOps teams can capture and review metrics for releases, phases, and tasks to find opportunities for improvement. (See the chapter “Measuring the Impact” for more information on DevOps metrics.)

CHAPTER 7

CONSISTENT HIGH-QUALITY SOFTWARE WITH DEVOPS

CONSISTENT HIGH-QUALITY SOFTWARE WITH DEVOPS

There's great value to having a well-tuned CD pipeline enabling a continuous flow of changes to production—but only if those changes are of high quality. With DevOps and an embedded CD pipeline, quality is built in, not bolted on, guaranteeing high levels of quality in every change.

The traditional approach to testing was to first develop new features and then to test them. This method made testing a roadblock, something that prevented value from reaching customers. Development would tell the business that the feature was ready, but the changes were still not live as often understaffed quality teams tried to keep up with the work. When faced with a choice between delays or more testing, compromises were often made where testing was cut short, resulting in the low quality of many released features.

DevOps breaks down the wall between dev and test. With automatically testable specifications, testing is in the process even before development starts. By automating tests, quality verification is an ongoing activity performed continuously as part of the development process. Security testing too, is an ongoing part of the process instead of an afterthought.

Visibility is also improved. In the test-driven development model, a real-time picture of the current level of functionality and quality is available at all stages in the delivery pipeline. The business is never surprised by last-minute delays caused by late-breaking quality or security issues.

Executable Specifications: Automated Acceptance Testing

Traditional software specifications were obsolete almost as soon as they were written. Countless hours would be spent capturing requirements in documents that could stretch to hundreds of pages. They would be printed, bound, and nearly immediately disappear into a desk drawer or be used to balance out a wobbly table leg. Why? Because requirements and functionality change while documents tend to remain frozen in time. And for testing purposes, these lengthy documents would be translated into test cases to be executed manually after development was complete. Writing page upon page of requirements up front was a costly, time-consuming process that frequently resulted in tests that did not actually verify the originally intended functionality.

Continuous Delivery avoids these problems by employing executable specifications. These specs define the requirements in simple English that non-technical members of the DevOps team can understand, but that also represent the actual tests run by a test tool. It eliminates "lost in translation" problems because the specification and the functional acceptance test are the same thing.

Behavior Driven Development (BDD) expresses both the behavior and the expected outcomes of the application from a user's point of view. These scenarios allow automated acceptance tests to be run using tools such as Cucumber or RSpec.

There's More to Quality Than Functionality: Automated Non-functional Testing

The needs of a user are not the only requirements to be met for a high-quality software experience. A great deal of non-functional requirements, such as performance, availability, accessibility, compliance, security, and so on, are all critical for real-world success.

As with functional specifications, non-functional requirements should be identified and stated in a testable—ideally, automatically testable—form. In a Continuous Delivery environment, it's important to remember that non-functional requirements can be part of the feedback loop just as much as feature ideas; system metrics, such as latency, load, and stability, can equally suggest important improvements that ultimately deliver value to customers.

Treating non-functional improvements as "just another type of change" is also the right way to balance between functionals and non-functionals, which all contribute to overall system quality. Non-functional requirements tend to be overlooked by the business, while developers may focus on them too strongly.

DevOps teams encourage everyone to contribute ideas for improvements, not just the business owner. Using data about system performance and user experience allows architectural improvements and other changes that tackle "technical debt" to be added to the backlog and prioritized more fairly. They can then be handled by the Continuous Delivery pipeline in the same manner as a new feature.

Non-functional requirements are part of the overall quality specification and should be included with other acceptance criteria. These requirements should be documented, understood, and tested automatically and often.

Keeping the System Running: Automated Regression Testing

Since a high-performing DevOps team can use a CD pipeline to automatically deliver new features quickly and regularly, a high degree of confidence is needed that those new changes won't break existing functionality. Such errors are called regressions and quickly result in a poor user experience, especially if they affect frequently used or highly visible parts of the system.

Testing internals, or white box testing, is useful for preventing regressions caused by the small, incremental changes common in Continuous Delivery. Unit tests written by a developer before the actual code is written serve as a description and verification of intended behavior. Once a unit test is created, it is run every time, safeguarding the ongoing correct functioning of the component. As the system grows, so does the collection of unit tests. This practice helps ensure all the pieces of the system still work as intended as changes add functionality, and especially when restructuring the system to improve non-functional qualities.

But a large collection of automated tests can't always be run for every single commit, or the process will quickly run into hours or days. Two measures determine good regression tests: coverage and execution time. Coverage is a percentage that indicates how much of the system's behavior is validated by the tests. Aim for 100%, but make sure the tests are meaningful and relevant, verifying especially those parts of the system that will result in the greatest loss in business value if regressions occur. Aim also to keep the time required to run each test as low as possible, otherwise, there simply won't be enough time to run the tests needed to achieve 100% coverage. Overall, a balance must be found between the maximum acceptable runtime for tests and the degree of coverage achievable. An investment in test infrastructure and tooling, as well as running tests in parallel, can help.

DevSecOps

At its core, good software is secure—so secure software is good software. High-performing DevOps teams view security as an element of quality and treat it as integral part of the entire process. Certain types of security testing—static code analysis and dynamic analysis, for example—can be automated and integrated with other testing steps. Security should be considered from the very beginning of the project. At the requirement planning stage, for example, activities, including threat modeling, help teams build security into the application when it's easiest to implement. Strong DevOps teams include security professionals as members, just like developers, ops, and business owners.

Tools such as Fortify, SonarQube, Sonatype, and Snyk build application security quality evaluation into the release process. If application code does not meet security standards, the developer who introduced the code takes action immediately. Contrast this with non-DevSecOps approaches where security evaluation is only occasionally performed, perhaps with an annual third-party penetration test. By the time the results of that test are available, the development team has long since moved on to other work, and it's much more difficult to address the problems.

Tl;dr

- Automated acceptance tests are written with the business to verify new functionality.
- Automated tests for non-functional requirements are included in the process.
- Automated regression tests ensure the system remains stable.
- Testing can be more efficient with a tiered approach, where tests covering highly critical parts of an application are executed for every change, while the full test set is run less frequently.
- Application security is tested along with other quality concerns, so that security vulnerabilities are addressed earlier when they're easier and cheaper to fix.

CHAPTER 8

DEVELOPMENT:
INSTANT VISIBILITY

DEVELOPMENT: INSTANT VISIBILITY

In a traditional software development organization, developers often work in isolation on large pieces of a project. Every couple of weeks or months, an attempt is made to merge all the changes into a central version control system and "build the whole thing." Merging is an error-prone and frustrating process, where developers spend a lot of time getting all parts of the code to work together instead of focusing on creating tangible improvements and value for end users.

With Continuous Delivery, developers commit their changes to central version control several times a day. From there, changes are automatically built and tested to produce an updated version of the product. The application is always "ready to go," allowing the team to focus on functionality and delivering high-quality features to customers.

Developers use various tools and practices to ensure that each new feature works in the context of the overall system. Test-driven development, Continuous Integration, and product dashboards are all about identifying errors and defects as early as possible and making them clearly visible. After all, problems found at this stage of the delivery process are still relatively easily and quickly fixed.

Test-driven Development—Test First, Then Code

A developer should only be writing code required to ensure the new feature meets specifications. If the code needs to be modified later, for instance in case the functionality needs to be extended, or maintainability or performance need to be improved, the developer also must make sure that such changes do not break anything else.

These challenges are addressed by writing code-level tests before writing the actual program code. This practice is called test-driven development (TDD). The developer starts by writing failing "unit tests" for each component, or "unit," of the new feature. Code implementing the feature is then added until all these tests pass. The result of the unit tests makes the progress of the implementation of a feature clearly visible. The updated code is finally committed to version control, where Continuous Integration takes over to verify that the new version of the entire system meets business requirements and desired quality levels.

Continuous Integration—Making a Broken Build a Non-event

The individual pieces of an application contributed by developers must all fit together and work. For software to work, it must first build. Regularly building the application helps the development team ensure that the various components fit together at any moment in time.

The most effective way to ensure code is regularly assembled is to maintain a central Build and Continuous Integration server. This approach automatically tries to run a new build on a "clean" machine whenever the code in the central version control system is changed. When the build succeeds, the team's build dashboard stays green. If the build fails, it turns red.

Failed builds happen. The important thing is to have policies in place that help the team deal with them quickly. If the build fails and the dashboard goes red, the developer has two options:

1. Fix the error.
2. Remove the change and return to the last working version (rollback).

Fixing the problem should always be the preferred option, but it's not always possible in the moment. Having the ability to easily roll back the change empowers the developer to work on fixing the problem later or investigating it more without leaving the application in a "broken" state, which would hold up progress for the rest of the team.

Dashboards—Visibility Promotes Quality

CD pipelines can utilize automated tools to make flaws in code more visible. They can automatically identify insufficient test coverage, failing tests, or complex, unmaintainable code. When a team has high standards for quality and the right tools to make it easy to find quality problems, quality gets built in from the start.

High-performing DevOps teams use dashboards to publicly share information about the quality of the software, often on a large screen in the teams' working areas. High visibility of build results showcases achievement and fosters a sense of pride—it's not about blame and finger pointing. They can also create a sense of friendly competition between colleagues that drives standards up. Ultimately, they remind the entire DevOps team that everyone's focused on a high-quality project, and everyone can take pride in knowing that a product version is eligible for release only when the agreed quality standards have been met.

Take Build Infrastructure Seriously

With automated builds and automated tests taking such a central position in the CD pipeline, it's important that they be treated as mission-critical components. Teams can't be forced to wait for their build to start because the build server is at maximum capacity. The entire pipeline can't be halted when a server cannot be restored after a crash because no backups are available. Build and test servers are a critical part of software infrastructure that needs to be scalable, properly maintained, and backed up on a regular basis. Without them, new features can't reach customers, and the gains of the DevOps transformation can't be realized. Always deliver hardened platforms to dev teams—on demand—to accelerate development work and increase security.

Tl;dr

- Use test-driven development for high-quality and consistency.
- Use Continuous Integration to ensure the application works after every change made by every developer.
- Provide instant visibility into the quality of the software through prominent dashboards.
- Invest in a robust, performant, and scalable build and test infrastructure to keep your CD pipeline flowing at maximum speed.

CHAPTER 9

DEPLOYMENT:
MAKE IT A NON-EVENT

DEPLOYMENT: MAKE IT A NON-EVENT

Deployment time is a time of fear for most traditional software development organizations. Deploying a new version of an application to an environment is a time-consuming and error-prone process. The development team puts together a deployment package, prepares lengthy documentation, and hands the whole thing off to operations for the actual execution. Slight differences between the configurations of each environment frequently cause deployments to fail and result in even more delays and idle time. Deploying to production is an extremely stressful exercise that keeps organizations on edge. Teams fear late nights or weekends at the office or wearing a pager, always at the ready for a disaster.

It's a dramatically different activity for a DevOps organization. Using Continuous Delivery frees them to automatically deploy new versions of applications in a matter of minutes, providing instant feedback to the delivery team, and allowing them to respond rapidly to customer demand. For these enterprises, deploying to production is a routine non-event that occurs multiple times a day. Additional deployment strategies, such as canary releases, allow new features to be exposed to a small set of customers to monitor whether they are functional and effective—both technically and commercially—before exposing the features to the complete user base.

Deploy Anytime, Anywhere

In traditional IT organizations, the deployment of a new application is a highly bureaucratic, often manual process. Deploying a new version to a target environment is so time consuming and mired in red tape that development teams avoid it for as long as they can. When the paperwork has finally been dealt with, it still takes a lot of time and technical effort on the part of both development and operations to actually get the new version up and running.

In Continuous Delivery, the deployment process across all target environments is fully automated. Teams can quickly test new functionality in any environment, and individuals are empowered to deploy new versions to production on demand. Automating the deployment process helps bridge the gap between development and operations and build end-to-end delivery teams. Just like automated tests, an automated deployment process should be a standard project deliverable from day one, with development and operations jointly determining and implementing an automated strategy to ensure that deployments “just work.”

With deployment automation in place, any team member with sufficient authorization, including the business owner, can initiate a deployment to any of the target environments with a simple click. This “self-service” deployment process eliminates traditional dependencies on operations or even specific operators. It also ensures that an identical process is used every time. Reproducibility is guaranteed, and the deployment server itself maintains a full audit trail for every deployment—a big win for compliance teams.

Since the CD pipeline includes an appropriate set of tests that run automatically prior to deployment, the number of times a “bad” build goes into production should be extremely small. However, if something does go wrong with a deployment, rollback can be automated as well. More often, high-performing DevOps teams opt for a quick “roll forward” in the form of a hot fix. In either case, automation makes it possible to roll back quickly, reducing the likelihood of impact to customers.

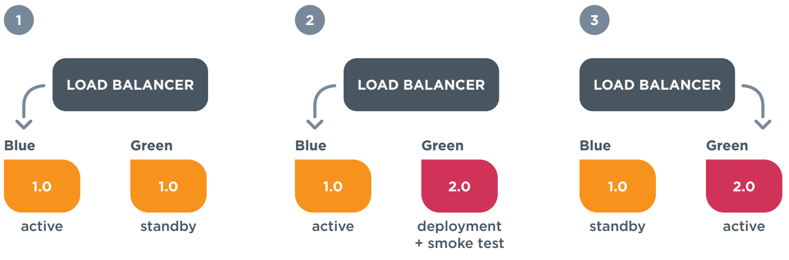
Validate in Production—Blue/Green Deployments, Canary Releases, and Dark Launches

In a traditional software delivery process, there are often fears that things that worked in development won’t work in production. Or teams fear that the real-world use of a feature by real users will reveal problems not anticipated in testing. These problems that surface after the go-live cause organizations to add ever more checks and tests into the release process, lengthening the time to market even more.

Since moving changes into and out of production with CD is easy and quick, “production” can still be a time of learning, validating, and testing. Blue/Green deployments and canary releases are two types of incremental deployment strategies that minimize the risk of downtime and allow the DevOps team to discover how the application behaves in a production environment. Feature flags and dark launches can measure user response to new functionality. In a traditional process, these techniques are nearly impossible, but CD lets them be used as a powerful everyday tool. The insight into real-life customer behavior enables the organization to learn faster and make better business decisions, day after day.

Blue/Green Deployments and Canary Releases

Blue/Green deployment is a deployment pattern that provides a safe way to upgrade applications without interrupting their use.[2] In a Blue/Green deployment pattern, a load balancer directs traffic to the active (Blue) environment while you upgrade the standby (Green) environment. After smoke testing the application in the Green environment and establishing that it is operating correctly, you adjust the load balancer to direct traffic from the Blue environment to the Green environment. If something goes wrong once the Green environment is in active use, you simply adjust the load balancer to direct traffic back to the Blue environment. Blue/Green deployments work particularly well for monolithic applications that can take significant time to deploy because you have full control over the point at which users can access the new version of the software.



Step 1
The Blue environment is active; the identical Green environment is on standby.

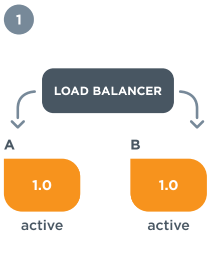
Step 2
Upgrade and test the application in the Green environment.

Step 3
Switch the load balancer from Blue to Green. The new application version is immediately live.

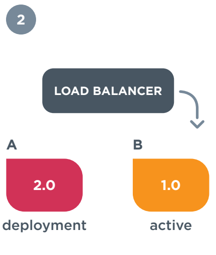
In a canary release, you upgrade an application on a subset of the infrastructure and allow a limited set of users to access the new version. This approach allows you to test the new software under a production-like load, evaluate how well it meets users' needs, and assess whether new features deliver on the business objective. A percentage of the system's users is directed to the new "trial" version, where the behavior of the users and system is closely monitored. Any anomalies are quickly detected without affecting the vast majority of customers.

[2] *The term Red/Black deployment is sometimes used to describe a similar deployment pattern, notably by Netflix.*

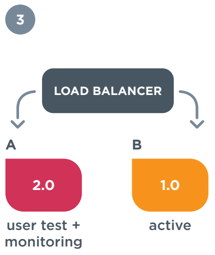
When necessary, the sample users can instantly be redirected back to the stable version, and the “canary servers” rolled back to the prior application version with a single click using deployment automation. If all goes well, the new version is rolled out to the entire production environment.



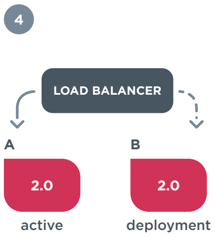
Step 1
Divide the active environment into a “main” part and a “canary” part.



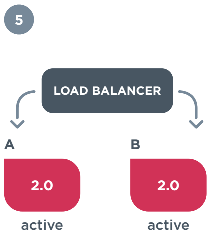
Step 2
Take the canary part of the environment offline and upgrade it.



Step 3
The canary environment is now live. Route a selected stream of traffic to it for testing and monitoring.



Step 4
If the canary tests succeed, upgrade the rest of the environment. You can use the rolling update pattern to minimize downtime.



Step 5
The entire application has been upgraded.

Dark Launches and Feature Flags

A dark launch is a way to gradually release a feature into production to test performance, quality, and user response. A feature flag (or feature toggle) wraps certain portions of application functionality, allowing the team to control who gets to see the feature and when. At runtime, these features can be exposed to a controlled subset of the user base—an “early access” group, employees, partner companies, or “friends and family”—while still being hidden from all other users. It’s a great way for teams to get real-world feedback on a new feature outside of a development or test environment.

Continuous Deployment With Containers

Containers provide a standardized, self-contained unit for software deployment. Each container includes everything the application or a service needs to run: code, runtime, system tools, and system libraries, in a form that is easily installed on a server. The contents of the container run in a protected execution space, isolating the application or service and making it easier to manage. Containers have a smaller runtime and storage footprint than Virtual Machines, so they can start and stop in seconds and require less storage and network bandwidth.

One of the biggest benefits of containers is that some deployment tasks become easier: developers can “ship” the application code together with the operating environment. However, the actual release process doesn’t go away, and the typical challenges faced throughout an enterprise-scale software delivery process stay the same.

Since containers wrap everything into a single package, extra attention should be paid to understanding what’s really “in” them. Version drift can become a problem as containers may have different OS configuration and middleware versions, as well as different versions of the same application. Problems grow rapidly as versions of components multiply and environments proliferate. Even with popular container orchestration platforms like Kubernetes, scaling containers in the enterprise can be challenging. Managing Kubernetes Secrets, Pods, Namespaces, ReplicationControllers, and Services on the Kubernetes container cluster manager involves complex processes and dependencies, especially as hundreds of applications get deployed thousands of times a month.

Release Orchestration and Deployment Automation tools can play a vital role in bridging the gap between the promise of containers and the realities of complex enterprise application delivery.

Tl;dr

- Continuous Delivery makes moving a new version through the pipeline and deploying it to production a low-risk, everyday event.
- DevOps removes error-prone handoffs between development and operations.
- Deployment of new versions of the software to all environments is fully automated.
- Production environments provide testing and learning opportunities when features are deployed incrementally to production and exposed to users gradually.
- Containers can further improve DevOps teams’ productivity, but they need to fit into all the other pieces of the CD pipeline.

CHAPTER 10

RELEASE COORDINATION: ORCHESTRATING YOUR PIPELINE

RELEASE COORDINATION: ORCHESTRATING YOUR PIPELINE

Traditional software delivery organizations typically rely on oversight from a single release manager. This individual is responsible for getting the entire team together to plan what needs to be done, coordinating and tracking all the activities across multiple teams as the release progresses, and reporting back to the business—inevitably, not frequently enough. Release managers depend on spreadsheets, emails, frantic phone calls, and constant legwork to try to stay up to date on what's actually happening in a project.

DevOps teams share responsibility for managing a project, so dedicated release managers are seldom required. The CD pipeline orchestrates the sequence of automated tasks that constitute the release process, and the pipeline is the responsibility of the entire delivery team. Anyone—including the business—can track the real-time progress of a new application version at any time via the pipeline dashboards.

Pipeline Optimization

In a full Continuous Delivery pipeline, all activities in each stage, all approvals, and all transitions between stages are automated. A developer checks a change to the source code into version control, the updated codebase is automatically built, unit and integration tested, deployed to an acceptance test environment, acceptance tested, approved, and deployed to production.

A basic pipeline starts with a simple, linear sequence of automated stages. Every new application version progresses through these stages, passing a series of automated approval gates before deployment to production. Each approval aims at ensuring a different aspect of quality of the deliverable to prevent production errors.

One immediate observation is that the simple pipeline in this example executes stages in sequence even when these stages are independent of each other. Provided sufficient execution capacity and a suitable target environment is available, pipeline throughput time can be improved by running such stages in parallel. In addition, work can also be parallelized within stages. For example, if automated acceptance tests can be run across four systems rather than one, the acceptance test stage can be completed in a quarter of the time.

From Release Plans to Delivery Pipelines

Release coordination tools help teams improve gradually. The team can start with their existing manual or partially automated release processes, identify the biggest bottlenecks and delays through automated value stream mapping, and then progressively replace these with automated tasks. This automation process may not always happen all at once. Even incremental steps towards a fully automated Continuous Delivery pipeline can add significant value to the speed and quality of a team's delivery.

Scripting Pipelines—A Brittle Solution

Many tools in a DevOps process define the pipeline—and the connection between multiple pipelines—using scripts that can be automatically run without the need for human intervention. Scripting is attractive to developers: the syntax is familiar, they like using their own development tools to interact with CD tools, and the scripts can benefit from the same controls that are used with application code (version control, peer code review, etc.). But the number of scripts required to automate the pipeline increases very quickly. Valuable development time that could be used for features gets spent writing, testing, and maintaining customized scripts that must be adapted for different applications and environments. Security can be a concern as well: many high-profile breaches have been caused by API keys, credentials, or other information that can compromise systems being stored in automation scripts.

Manual scripting costs time and money in both the short run and in the long run. Instead, consider a model-based, highly scalable tool to orchestrate complex release pipelines from end to end, across all stages and all environments.

Tl;dr

- DevOps teams don't need dedicated release managers to coordinate all the people and activities needed to deliver a new release to production.
- Every member of a DevOps team should be able to track a fully automated release process via a dashboard.
- Different tasks and stages in a release process can be run in parallel to optimize the overall process.
- A release pipeline orchestration tool can help DevOps teams move from traditional processes to Continuous Delivery.
- Scripting everything in a DevOps pipeline presents maintenance challenges and quickly becomes difficult to manage at scale.

CHAPTER 11

MEASURING THE IMPACT

MEASURING THE IMPACT

A core principle of DevOps is fast feedback. The sooner a determination can be made that the value created is delivering in the way the business expects for customers, the sooner it can be enhanced—or changed—if necessary. The same goes for the process of developing and delivering that value. An early understanding of what works and what doesn't work in the development process enables teams to invest more in high-value activities and stop or modify those that fail to deliver. These ideas are not new, but the amount of data available with which to make those measurements is significantly greater today than ever before. As more of the development process is automated, more data is produced. And better instrumentation of applications produces more data about how users interact with a system and get value from it. Business leaders need to mine this valuable data to understand the impact that a DevOps transformation is having on both the customers and the development process itself.

Impact to the Development Process

DevOps teams are typically focused on goals such as "improve quality" and "increase speed of delivery." There are some key metrics that provide evidence of attainment of those goals:

- **Time to delivery:** How long does it take for a single line code change to get fully deployed? Answering this question helps you understand the efficiency of your process and see where bottlenecks might exist.
- **Frequency of releases:** How often are releases pushed live? Keeping batch size small helps drive early feedback, so it's important to release often.
- **Change volume:** What value is going into production with each release? This may be measured in user stories, for example.
- **Success rate:** What percentage of deployments failed, resulting in an outage? This highlights the quality of both the code and the release process.
- **Mean time to Recovery (MTTR):** How long does it take to recover from a failed change? High-performing DevOps teams can react rapidly and respond to unfamiliar problems.
- **Defects:** How many defects were identified and resolved while still in development versus being released into production and fixed later?
- **Team comparisons:** How are teams performing against their peers? Is their performance on this list of metrics better than the average?

Impact to Customers

Recent studies suggest that as little as one third of all changes actually improve outcomes for users. The rest either have no impact on desired outcomes or, shockingly, make outcomes worse! High-performing DevOps teams, therefore, focus less on time to delivery and more on time to value. The tangible and intangible benefits of a DevOps transformation for customers may include increased adoption, higher conversion rates, improved happiness, and reduced need for technical support. Some metrics to consider:

- Application utilization and traffic
- Calls to APIs
- New user sign-ups
- Performance (i.e., page response time)
- Support ticket volume
- Public or social media feedback (e.g., customer tweets)
- Net Promoter Scores or other sentiment analysis techniques
- Customer churn

Impact to the Team

DevOps has productivity benefits, but it has also been shown to impact team satisfaction. The 2017 DevOps Research and Assessment (DORA) State of DevOps Report found that employees in high-performing DevOps organizations are 2.2 times more likely to recommend their organization as a great place to work. DevOps improves the development workflow for all members of a team, which can lead to higher job satisfaction and lower turnover. DevOps teams report spending less time on less enjoyable tasks, such as rework and bureaucracy, leaving more time for more creative new work that delivers more value to customers.

With its focus on cross-functional, collaborative teams, employees using DevOps feel more connected to one another. Developers understand the business benefit of a feature because they work directly with the business stakeholders who drive them. Operations professionals feel less antagonistic to developers because they see that everyone on the team takes responsibility for the successful operation of the software (not just the development of it). And a host of other participants, from regulation and compliance, to security, to business managers, are no longer kept on the periphery of a project.

Insight for Compliance

Compliance considerations play a key role in how software is built, especially for highly regulated enterprises, such as publicly traded companies, financial services, insurance organizations, government agencies, and many others. Teams in these situations need real-time answers to questions such as:

- Is a standardized and approved software delivery process being followed?
- What changes were made to the software?
- Who made them and when?
- Were they approved?
- Did the company follow the necessary rules for each activity as dictated by SOX, PCI, and other regulations?
- Was the appropriate testing conducted?
- Were permissions properly controlled?

Automated CD pipelines and other DevOps practices make answering these questions much easier than in traditional development processes. And individuals responsible for compliance are part of a DevOps team, so they no longer need to rely on second-hand information about how a project was delivered. The DevOps platform further supports compliance by applying controls to the CD pipeline itself. Steps can be made mandatory, and access controls can be applied in granular ways if needed.

Ongoing Customer Conversation

The work of a DevOps team doesn't end when a feature is released to production. If customers do not react positively to the new functionality—if it fails to deliver value—then it doesn't matter how rapidly it was delivered. DevOps teams rely on ongoing dialogue with their user base to identify desired improvements and feed them back into the process. This feedback loop is critical for realizing the benefits of DevOps.

Tl;dr

- DevOps impacts both the process of building software and the value the software delivers to customers.
- Measuring the impact of a DevOps transformation can drive continual improvement.
- Look for ROI in unexpected places, such as compliance and staff retention.

CHAPTER 12

GETTING STARTED

GETTING STARTED

DevOps is a philosophy, and a DevOps transformation requires more than just introducing a few new tools. It affects organizational structure, roles and responsibilities, how developers go about their work and, ultimately, the relationship between IT and the business. It can seem daunting to many organizations but can succeed anywhere with some manageable steps.

Acknowledging the Challenge

The greatest problem a traditional software delivery organization faces is not the inefficient, error-prone delivery and feedback process as such. The biggest impediment is the acceptance that the current process represents the "natural order of things."

Acknowledging that your delivery process can be improved, and communicating clearly that the way things are currently done is open to debate and revision, is a critical prerequisite for a successful DevOps transformation.

Agile Development

DevOps requires teams to work in an iterative manner. "Agile Development" is a proven approach to iterative and incremental software development and is an excellent first step for organizations wanting to embrace DevOps. An agile mindset, focusing on business value and continuity through short, iterative development cycles, is a natural generator of the continuous flow of changes to production enabled by Continuous Delivery.

Value Stream Mapping

Teams can map out their current value steam to identify where the most immediate improvements are available. A value stream describes all steps in a particular process, their duration, and any intermediate idle time. In a traditional software delivery process, a large proportion of the overall throughput time is spent idling.

With the Value Stream Map, the DevOps team can determine where the most painful bottlenecks are and which improvements are likely to deliver the greatest benefit. Based on an estimate of the benefit of each improvement in quantifiable terms, such as cost, quality, and time-to-market, improvements can be incrementally delivered with increasing ROI seen every time.

One Bottleneck at a Time

There is no such thing as a standard “DevOps Transformation Plan” suitable for every organization. However, a pattern common to successful implementations is to favor incremental approaches over blanket, organization-wide rollouts. In many cases, an implementation can start with a single team reworking their own processes or by efforts that tackle the “biggest pain” first.

Tl;dr

- DevOps transformations can seem daunting due to their wide reach into so many aspects of a business.
- Value stream mapping can highlight waste in a traditional process, helping identify the biggest “bang for your buck” improvements.
- DevOps transformations rarely happen all at once. They expand through the enterprise gradually, and steadily demonstrate their ROI.

CHAPTER 13

SCALING THE TRANSFORMATION

SCALING THE TRANSFORMATION

Value is seen soon after starting a DevOps transformation. But after those initial, often isolated successes, the challenge becomes how to spread DevOps through an entire organization to achieve a total DevOps transformation. Most high-performing DevOps organizations who have been successful have favored "opt in" participation over mandated adoption, using a "pull" approach to change rather than a "push."

Share and Celebrate Success

DevOps is all about people, and people like to share. Making DevOps success visible allows early adopters to share their knowledge, and it allows potential allies to find them. It also allows the discussion about DevOps to center around practical, real-world results instead of just opinion. Many organizations host internal "DevOps Days," recreating the popular community events as internal tech conferences. Others facilitate cross-team sharing with chat rooms, wikis, or other knowledge sharing tools.

Standardize When Appropriate

DevOps teams should feel empowered to experiment and find what works best for them, and that recipe for success may differ from team to team within an organization. But a certain degree of standardization—specifically when it comes to tools—can increase the efficiency of scaling DevOps throughout an enterprise. This allows a more efficient investment in infrastructure and improved knowledge sharing. And when teams find it easy to adopt common services (e.g., CI/CD, analytics, self-service portal[s], and code repository) they offer less pushback to adopting those standardized tools and processes.

DevOps for Everyone

Making software is a technical activity, but not everyone on a DevOps team is technical. Some organizations adopt tools that appeal to technical users, for example, tools that represent every piece of the CD pipeline as a code artifact. This can make DevOps difficult to scale, since all participants require high-cost coding skills, and maintaining those code artifacts is a task that often falls through the cracks. Instead, successful DevOps teams look to a dual-mode approach that combines low-code/no-code tools for less technical team members with other code-centric approaches that may be preferred by engineers. This flexibility ensures a “bigger umbrella” for a DevOps team and supports more rapid scaling.

Tl;dr

- DevOps is a transformation involving people, so people are the most effective tools to spread its adoption.
- Pull works better than push for spreading DevOps.
- Teams need freedom to experiment with what works best for them, but some standardization—particularly around tools—can reduce cost and complexity in organization-wide efforts.
- DevOps teams combine both technical and non-technical members, so they can only scale if the tools they use are appropriate for all skill levels.

CHAPTER 14

FINAL THOUGHTS

FINAL THOUGHTS

Software is critical for business success, no matter what your company's core business is. Software allows you to create more value for your customers faster, and it lets you beat the competition. But complex, "big bang" releases, slow, error-prone manual processes, repeated handovers, and delays lead to failures in production, poor quality, costly security breaches, and dissatisfied users.

High-performing DevOps teams release high-quality software straight to production and set up rapid and effective feedback loops between the business and users. They succeed with agility, delivering deployments more frequently with fast lead times that get value into customers' hands quickly. And releases are more reliable with changes that fail less often.

A DevOps platform is key to successfully implementing Continuous Delivery. It provides the entire set of tools and processes to the cross-functional teams who deliver Continuous Delivery at enterprise scale. A DevOps platform speeds enterprise adoption and provides the intelligence, automation, and control needed for even the most complex global organizations.

Start your path to DevOps now and experience the benefits of delivering business value in days, not months!

GLOSSARY

GLOSSARY

A/B Testing. A technique in which a new feature, or different variants of a feature, are made available to different sets of users and evaluated by comparing metrics and user behavior.

Acceptance Testing. Typically high-level testing of the entire system carried out to determine whether the overall quality of both new and existing features is good enough for the system to go to production.

Agile. A precursor to DevOps. Agile is a software development and, more broadly, business methodology, that emphasizes short, iterative planning and development cycles to provide better control and predictability and to support changing requirements as projects evolve.

Application Release Orchestration (ARO). Tools, scripts, or products that automatically install and correctly configure a given version of an application in a target environment, ready for use. Also referred to as "Application Release Automation" (ARA) or "Continuous Delivery and Release Automation" (CDRA).

Behavior-Driven Development (BDD). A development methodology that asserts software should be specified in terms of the desired behavior of the application, and with syntax that is readable for business managers.

Black Box Testing. A testing or quality assurance practice that assumes no knowledge of the inner workings of the system being tested, and which thus attempts to verify external rather than internal behavior or state.

Build Agent. A type of agent used in Continuous Integration that can be installed locally or remotely in relation to the Continuous Integration server. It sends and receives messages about handling software builds.

Build Artifact Repository. A tool used to organize artifacts with metadata constructs and to allow automated publication and consumption of those artifacts.

Build Automation. Tools or frameworks that allow source code to be automatically compiled into releasable binaries. Usually includes code-level unit testing to ensure individual pieces of code behave as expected.

Canary Release. A go-live strategy in which a new application version is released to a small subset of production servers and heavily monitored to determine whether it behaves as expected. If everything seems stable, the new version is rolled out to the entire production environment.

Configuration Drift. A term for the general tendency of software and hardware configurations to drift, or become inconsistent, with the template version of the system due to manual ad hoc changes (like hotfixes) that are not introduced back into the template.

Container. Similar but more lightweight than a virtual machine, containers are stand-alone, executable packages containing everything needed to run a piece of software: code, runtime, system tools, system libraries, settings, and so on.

Configuration Management. A term for establishing and maintaining consistent settings and functional attributes for a system. It includes tools for system administration tasks, such as IT infrastructure automation.

Continuous Delivery. A set of processes and practices that radically remove waste from your software production process, enable faster delivery of high-quality functionality, and set up a rapid and effective feedback loop between your business and your users.

Continuous Integration (CI). A development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early.

Dark Launch. A go-live strategy in which code implementing new features is released to a subset of the production environment but is not visibly, or only partially, activated. The code is exercised, however, in a production setting without users being aware of it.

Delivery Pipeline. A sequence of orchestrated, automated tasks implementing the software delivery process for a new application version. Each step in the pipeline is intended to increase the level of confidence in the new version to the point where a go/no-go decision can be made. A delivery pipeline can be considered the result of optimizing an organization’s release process.

DevOps. A portmanteau of development and operations, DevOps is a set of processes, practices, and tools that improve communication, collaboration, and processes between the various roles in the software development cycle, resulting in delivery of better software with speed and stability.

DevOps Intelligence. An approach to continually improving software development practices by analyzing the data produced by the tools and development process itself. This allows organizations to improve efficiency, lower risk, and create better software.

DevSecOps. The practice of integrating security into the DevOps process.

Functional Testing. Testing of the end-to-end system to validate functionality. With executable specifications, Functional Testing is carried out by running the specifications against the application.

Infrastructure as a Service (IaaS). Cloud-hosted virtualized machines, usually billed on a "pay as you go" basis. Users have full control of the machines but need to install and configure any required middleware and applications themselves.

Infrastructure as Code. A system configuration management technique in which machines, network devices, operating systems, middleware, and so on are specified in a fully automatable format. The specification, or "blueprint," is regarded as code that is executed by provisioning tools, kept in version control, and generally subject to the same practices used for application code development.

Lean. "Lean manufacturing," or "lean production," is an approach or methodology that aims to reduce waste in a production process by focusing on preserving value. Largely derived from practices developed by Toyota in car manufacturing, lean concepts have been applied to software development as part of agile methodologies. The Value Stream Map (VSM), which attempts to visually identify valuable and wasteful process steps, is a key lean tool.

Microservices. A software architecture design pattern in which complex applications are composed of small, independent processes communicating with each other using language-agnostic APIs. These services are small, highly decoupled, and focus on doing a small task.

Non-functional Requirements (NFRs). The specification of system qualities, such as ease-of-use, clarity of design, latency, speed, and ability to handle large numbers of users, that describe how easily or effectively a piece of functionality can be used, rather than simply whether it exists. These characteristics can also be addressed and improved using the Continuous Delivery feedback loop.

NoOps. A type of organization in which the management of systems on which applications run is either handled completely by an external party (such as a PaaS vendor) or fully automated. A NoOps organization aims to maintain little or no in-house operations capability or staff.

Orchestration Pipeline. Tools or products that enable the various automated tasks that make up a Continuous Delivery pipeline to be invoked at the right time. They generally also record the state and output of each of those tasks and visualize the flow of features through the pipeline.

Platform as a Service (PaaS). Cloud-hosted application runtimes, usually billed on a “pay as you go” basis. Customers provide the application code and limited configuration settings, while the middleware, databases, and so on are part of the provided runtime.

Product Owner. A person or role responsible for the definition, prioritization, and maintenance of the list of outstanding features and other work to be tackled by a development team. Product Owners are common in agile software development methodologies and often represent the business or customer organization. Product Owners need to play a more active, day-to-day role in the development process than their counterparts in more traditional software development processes.

Provisioning. The process of preparing new systems for users. In a Continuous Delivery scenario, this work is typically done by development or test teams. The systems are generally virtualized and instantiated on demand. Configuration of the machines to install operating systems, middleware, and so on is handled by automated system configuration management tools, which also verify that the desired configuration is maintained.

Regression Testing. Testing of the end-to-end system to verify that changes to an application did not negatively impact existing functionality.

Release Coordination. The definition and execution of all the actions required to take a new feature or set of features from code check-in to go-live. In a Continuous Delivery environment, this is largely or entirely automated and carried out by the pipeline.

Release Management. The process of managing software releases from the development stage to the actual software release itself.

Release Orchestration. The use of tools for defining, automating, securing, tracking, and controlling manual and automated tasks in application releases.

Test-Driven Development (TDD). A development practice in which small tests to verify the behavior of a piece of code are written before the code itself. The tests initially fail, and the aim of the developer(s) is then to add code to make them succeed.

Unit Testing. Code-level (i.e., does not require a fully installed end-to-end system to run) testing to verify the behavior of individual pieces of code. Test-driven development makes extensive use of unit tests to describe and verify intended behavior.

Value Stream Mapping. A process visualization and improvement technique used heavily in lean manufacturing and engineering approaches. Value Stream Maps are used to identify essential process steps vs. “waste” that can be progressively eliminated from the process.

Virtualization. A systems management approach in which users and applications do not use physical machines, but simulated systems running on actual, “real” hardware. Such “virtual machines” can be automatically created, started, stopped, cloned, and discarded in a matter of seconds, giving operations tremendous flexibility.

Waterfall. A software development methodology based on a phased approach to projects, from “Requirements Gathering” through “Development” and so on, to “Release.” Phases late in the process (typically related to testing and QA) tend to be squeezed, as delays put projects under time pressure.

White Box Testing. A testing or quality assurance practice that is based on verifying the correct functioning of the internals of a system by examining its (internal) behavior and state as it runs.

Software is more than just code. It brings immense value to your customers when it gives them what they need: a way to play games, manage investments, book a flight, or even put a satellite in space. And software that's good for customers is good for the business, as long it's high-quality, reliable, secure—and delivered ahead of the competition.

The promise of delivering better software faster is why so many companies have moved to DevOps. But DevOps isn't just a "technical thing" that magically leads to success. It takes people, processes, and tools from across the organization seamlessly working together.

Continuous Delivery is a set of processes and practices essential for achieving DevOps. Together, DevOps and Continuous Delivery form the most e ffective approach for getting high-value features to market fast.

The IT Manager's Guide to DevOps is an update to the popular *The IT Manager's Guide to Continuous Delivery*. It provides essential knowledge about DevOps and Continuous Delivery—their business benefits and the cultural, technical, and process considerations involved in achieving them. Whether you're new to DevOps or looking to scale, this book helps you create a high-performing organization that drives business value up and time to market down.

XebiaLabs

ISBN 978-0-359-09887-3
90000
9 780359 098873