Brought to you by:

vmware®

DevOps for Infrastructure

for dummies®
A Wiley Brand

Bridge the gap between development and operations

Transition to infrastructure as code

Implement GitOps best practices

Sam McGeown

Lefteris Marakas

VMware Special Edition

About VMware

VMware offers a breadth of digital solutions that power apps, services , and experiences, enabling organizations to deliver the best customer service and empower employees. VMware streamlines the journey for organizations to become digital businesses that deliver better experiences to their customers and empower employees to do their best work. Our software spans app modernization, cloud, networking and security, and digital workspace.

Since our founding in 1998, our employees and ecosystem of more than 30,000 partners have been behind the technology innovations transforming entire industries—from banking, healthcare, and government to retail, telecommunications, manufacturing, and transportation.

Every day, we work to solve our customers' toughest challenges through disruptive technologies—like edge computing, artificial intelligence, blockchain, machine learning, Kubernetes, and more—to define the digital foundation that will accelerate the next wave of innovation.

DevOps for Infrastructure

DevOps for Infrastructure

VMware Special Edition

by Sam McGeown and Lefteris Marakas

for dummies®
A Wiley Brand

DevOps for Infrastructure For Dummies®, VMware Special Edition

Published by, **John Wiley & Sons, Inc.,** 111 River St., Hoboken, NJ 07030-5774, www.wiley.com

ISBN 978-1-119-83780-0 (pbk); ISBN 978-1-119-83781-7 (ebk)

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Project Editor: Elizabeth Kuball

Acquisitions Editor: Ashley Coffey

Editorial Manager: Rev Mengle

Business Development Representative: Cynthia Tweed

Production Editor: Tamilmani Varadharaj

Special Help: Faithe Wempen, Jimmy Chunga

Table of Contents

Introduction

DevOps is a big deal, and it's getting bigger all the time. A recent VMware-commissioned study by Forrester Consulting found that 92 percent of the IT decision-makers in large corporations have already rolled out or are planning to implement DevOps initiatives.

All is not smooth sailing, though. The same research determined that 80 percent of the respondents expect IT leaders to sponsor such projects but that a mix of organizational and personal barriers inhibit DevOps adoption.

DevOps started as a movement aiming to break the walls between development and operations roles, but it has mostly enabled developers to expand their toolboxes into operations tasks. It hasn't always included the infrastructure and operations (I&O) peers. And that's a problem.

Such a fragmented adoption model can bring some early benefits but is destined to miss out on optimizing operations for existing traditional applications. In addition, it may expose unforeseeable risks that are outside the realm of developer knowledge. But how can I&O departments overcome their concerns and jump on the DevOps train?

About This Book

This book is designed to help late DevOps bloomers overcome the most common initial barriers for adoption. It starts by providing an overview of the evolution of DevOps adoption and introduces the role of platform engineering to scope out the I&O responsibility in DevOps.

It continues by establishing some basic concepts and terminology that help simplify the theory of DevOps. It also defines the pillars for DevOps for infrastructure and curves out an IT path to DevOps.

Most important, it provides a prescriptive path to understand DevOps from an I&O starting point. Anchoring on capabilities and use cases that are relevant for those roles, such as infrastructure

as code, infrastructure pipelines, and GitOps, this book offers practical examples on popular open-source and productized solutions that will help demystify DevOps.

Foolish Assumptions

In writing this book, we made some assumptions about you, the reader. We assume you work for a medium to large company that wants to adopt or has started adopting DevOps but hasn't been fully successful yet.

We also assume you fit one of these general profiles:

- **You're an IT practitioner looking to evolve your skill set through DevOps principles and platform engineering.** You may be tasked with bringing DevOps practices to traditional workloads or establishing a DevOps-friendly platform for developers to use. You are not expected to have any prior knowledge of DevOps practices.
- **You're an IT executive or manager who would like to reap the benefits of DevOps but you aren't sure how to get started.** There may be other teams in your company, probably application developers, already heavily leveraging DevOps frameworks. You want to make sure you can align your team's or department's practices with them, in order to inject operational expertise without affecting the developer experience.

Icons Used in This Book

This book uses icons in the margin to draw your attention to certain kinds of information. Here's a guide to the icons:


TIP

We use the Tip icon to highlight anything that'll save you time or money or just make your life a little easier.


REMEMBER

When we tell you something so important that you should commit it to memory, we mark it with the Remember icon.

WARNING

When we want you to avoid making a potentially costly mistake, we mark that material with the Warning icon.

TECHNICAL STUFF

Sometimes we get into the weeds, providing information that's a bit more technical in nature. When we do, we mark it with the Technical Stuff icon.

Beyond the Book

This book explains the basics of DevOps for infrastructure and platform engineering, but if you want resources beyond what this book offers, here are some suggestions:

- Find out more about platform engineering and the way it differs from site reliability engineering (SRE) in this blog post authored by VMware's chief technology officer (CTO), Kit Colbert: `https://octo.vmware.com/platform-engineering-and-sre-whats-the-difference`.
- Read the commissioned opportunity snapshot, conducted by Forrester Consulting on behalf of VMware, to see how DevOps trends are affecting large enterprise IT organizations today: `https://bit.ly/VMW-DevOps`.
- Watch a webinar on how to overcome fear and uncertainty to benefit from DevOps. It's co-presented by Forrester's lead DevOps analyst, Charles Betz, and VMware's product marketeer and coauthor of this book, Lefteris Marakas. Check it out at `https://bit.ly/ForresterVMDevOps`.
- Try it for yourself! Follow the examples in this book with a free 45-day trial for vRealize Automation Cloud. Get started at `www.vmware.com/go/vra-get-started`.

IN THIS CHAPTER

- Entering the mainstream
- Bridging the gap between Dev and Ops

Chapter 1
Crossing the Chasm: How DevOps Came to Be

Buzzwords are nothing new in the world of software development. We get bombarded with them so frequently that when the next "big" term comes along, it's often ignored. You may see one, pause for a few seconds, and give it just enough attention to roll your eyes before quickly moving on to an issue of actual importance. And justifiably so. Most of these buzzwords seem to appear out of nowhere, get instantly overused by the least popular person in the office, and — poof! — they're gone, like a stale magic trick. When was the last time you used the term *futureware?*

Occasionally however, a buzzword comes along and eventually manages to gain enough traction to jump across the chasm from hype to becoming a movement. That's precisely what happened back in 2007 and 2008 with DevOps. Software engineers and Agile practitioners were getting increasingly frustrated with the inefficiencies introduced by the siloed organizational approach in the software development process. They started preaching a culture that breaks the barriers between developers and operators, in a more flexible organizational model based on automation and collaboration.

It's hard to believe the DevOps movement has been around for more than a decade. That's an eternity in the world of software development! Although it's been around for what seems like forever, and it has reached a significant level of awareness, there still seems to be a fair amount of confusion about what DevOps is and how it should be applied to an organization. This chapter clears up some of that confusion by explaining how DevOps came into being — and how it came to be so popular.

Making the Leap from Buzz to Mainstream

How did DevOps cross the chasm? In fact, a few stars had to align:

- An increasing need for fast, high-quality software development and delivery
- An abundance of cloud technologies and innovation
- Success stories and scientific proof that DevOps works

All these factors played an essential part in how DevOps has become a top mandate for any large company today. Here's a quick look at each of those factors in more detail.

The rise of software

The DevOps movement was triggered as an answer to the pressure created by the fast, often extreme pace of modern business. As a new reality started to take shape in which every company had to be a "software company," speed and quality of software releases became the main competitive differentiator across seemingly unrelated markets. Software companies started dominating one market after the other. Wikipedia made encyclopedias a nostalgic memory of our childhood. Amazon became the largest bookseller. Tesla reshaped the automotive industry. Robinhood disrupted agelong financial institutions. Airbnb came to be the largest accommodations provider, without owning any hotels. All of that happened because the companies operated with a software-first philosophy.

REMEMBER

That mentality continues today, and it shows no signs of stopping. Developers are constantly required to deploy more code faster so their companies can maintain and advance their competitive positions. As a result, it just isn't possible for development teams to have long cycles before tossing their work over the fence to be prioritized and maintained by their operations counterparts in a traditional waterfall model. DevOps offers the promise of development and IT operations teams working in tandem to facilitate rapid qualitative delivery of applications and services.

Cloud everywhere

Did you know that in 2020 Amazon made more profit from its cloud business than everything else combined? While most people were dismissing DevOps, the innovators figured they could make a buck out of it. A wave of continuous innovation (pun intended) supplied the DevOps development with the new tools, processes, and resources it required.

Large companies that pivoted quickly enough are some of the most successful ones today. Legacy technology providers tried to transition (not always successfully), and new companies selling DevOps and cloud started sprouting up like mushrooms. And although the extreme fragmentation of the solution space is frequently more confusing than helpful, it also powered the innovation that lowered the barrier of adoption for Agile practices.

It works!

Early on, many businesses turned up their noses at the concept of DevOps. After all, it was another buzzword that would no doubt end up costing them a lot of time and money while making dubious promises of returns. That assumption was reasonable, but a few companies had the foresight to see the disruptive benefits that DevOps could create.

For these scrappy, innovative early adopters of DevOps, the equation was simple: DevOps equals agility with focus on collaboration. The results were swift and game changing. The explosive growth of companies like Google, Netflix, and Amazon was historic and hadn't been seen in decades.

Well-established traditional companies like Target, Capitol One, and Nordstrom were under the threat of disruption. Fortunately

for them, while they may not have been at the forefront of the DevOps movement, they had their eyes on the ball, were able to see how the world was about to change, and reacted quickly. It's because of their rapid adoption that DevOps was launched across the chasm! It made the transition from being an obscure buzzword to the mainstream standard.

Sadly, for every Netflix, there are several companies that reacted very slowly or poorly in adopting DevOps and found themselves trampled by a stampede of change (think: Blockbuster, Circuit City, and Game Stop). They found out too late that poor DevOps practices ultimately means lost revenue and irreparable damage to brand awareness. There's a tough life lesson that we've all had to learn at one point or another. Some of us learn it in school, when almost overnight, your style in clothing is no longer cool. This happens when a student has the courage to do something different and show everyone a fresh, new way of expressing themselves. If you think this dynamic is any different in business, you're lying to yourself.

In today's world if you're not disrupting, you're being disrupted. Figure 1-1 illustrates the DevOps adoption curve, and marks today's place in it.

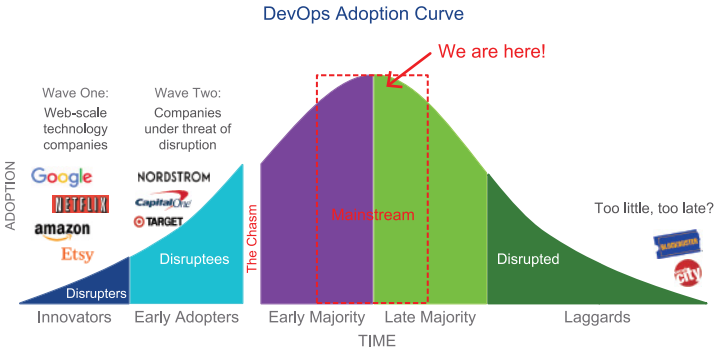

FIGURE 1-1: The DevOps adoption curve.

Anecdotal evidence from the different markets indicated that winning companies focus on their software development and delivery, but it wasn't until the mid to late 2010s that research on DevOps practices proved the benefits of DevOps. Books like *Accelerate* by Nicole Forsgren, Jez Humble, and Gene Kim (IT Revolution Press, 2018) quantify the benefits of adopting DevOps

practices in terms of not only efficiency, effectiveness, and customer satisfaction, but also commercial indicators like profitability, productivity, and market share. And because CEOs value (and understand) profits and market shares DevOps made it to their list of strategic initiatives.

How DevOps Bridges the Gap

The promise of DevOps has always been developers and operators working together and singing "Kumbaya," but the reality has evolved to be a bit different. Early DevOps innovation was powered by developers' frustration with the long IT waiting times and shifting priorities that affected their ever-shortening development cycles. Automation and collaboration platforms started coming out to relieve this developer pressure. That's why, in practice, DevOps has been mostly about developers applying their powerful tooling and best practices to traditional Ops responsibilities.

Recognizing the need

As developers established tools, they also established rules of conduct and best practices. Although they were proficient in optimizing the initial coding and building stages of the software release cycle, they lacked the expertise, context, and incentives to rigorously secure and operate a piece of code. Developers optimize for speed, but customers also care about security, reliability, and quality. A feature-rich platform that goes down every other hour is no good.

In the first wave of maturity, developers assumed that operations could become relatively trivial by leveraging their powerful toolchains and practices, even giving room for ephemeral trends like NoOps, a DevOps variation where operations is not needed at all because application developers can take care of that, too. They were wrong. A few unicorn startups found success in such models, but larger and more established companies were intimidated by this new trend of focusing solely on speed of delivery. The talent pool was just not big enough to support the demand. Separating concerns also typically made teams more effective and efficient.

As more and more large enterprises kept getting disrupted, and more CEOs added DevOps as a bullet point in their executive briefings, the need for someone to take care of the environment and shared services became increasingly apparent. This is where platform engineering comes into play.

The rise of platform engineering

Platform engineers and site reliability engineers (SREs) to the rescue! Platform engineers and SREs, in principle, have one job: providing the best environment for developer code to shine. In other words, they operate the infrastructure and platforms where new features will be hosted for customers to enjoy.

REMEMBER

Learning from developer best practices and leveraging the same or compatible tools (such as release pipelines, infrastructure as code, and Git-based source control), platform engineers are fueling the second phase of DevOps maturity, where even large legacy enterprises can implement Agile initiatives and benefit from the wonders of DevOps. DevOps is becoming mainstream!

IN THIS CHAPTER

- Discovering the DevOps cycle
- Identifying key principles of DevOps
- Understanding how DevOps makes processes continuous
- Defining DevSecOps and Shift Left
- Introducing DevOps for infrastructure

Chapter 2
Introducing DevOps

The DevOps methodology is designed to bridge the gap between development and operations teams, who quite naturally have different priorities, objectives, and approaches. A developer's primary goal is to create new features, whereas an operator's goal is to maintain site reliability and performance.

The essential source of the conflict is that making changes to implement new features (which is what Dev wants) risks the stability of the running system (which is what Ops *doesn't* want). Who's responsible for ensuring that a new feature launch doesn't directly impact users? The operations team has more skin in that game, but the development team owns the code. And speaking of code, who's responsible for checking code quality? What about analyzing performance issues or resolving incidents? And we haven't even started to consider security yet. . . .

REMEMBER

This is the heart of DevOps — bringing together these two seemingly conflicting sides of the same coin. Although the finer points of DevOps can be subjective, most definitions agree that DevOps should have the following benefits:

- **Minimizing the time to deliver value:** In other words, launching new features in the shortest amount of time possible.

- **Reducing risk:** Faster release cycles of smaller, more frequent updates lead to continuous, iterative improvements in a controlled manner. Smaller changes mean smaller risks.
- **Earlier detection and faster resolution of incidents:** Automated testing of code and better traceability lead to reduced failures in deployment and rollbacks from production.
- **Increased customer satisfaction:** Being able to launch new features more rapidly with increased stability makes for happier customers. Incorporating customer feedback into development planning means customers can have a direct influence on the features launched.

Now that you understand what people are trying to achieve with DevOps, here are some key concepts that will help get you where you want to be.

Looping through the DevOps Cycle

The DevOps life cycle is most often depicted as an infinite loop. This captures the continuous and iterative mode of development and operations. Code should pass through each stage in this repeatable system to minimize time to value, reduce risk, resolve incidents, and increase customer satisfaction.

REMEMBER

The DevOps cycle is broken down into various stages, as shown in Figure 2-1. There are differing definitions of these stages, but the majority seem to converge on the following:

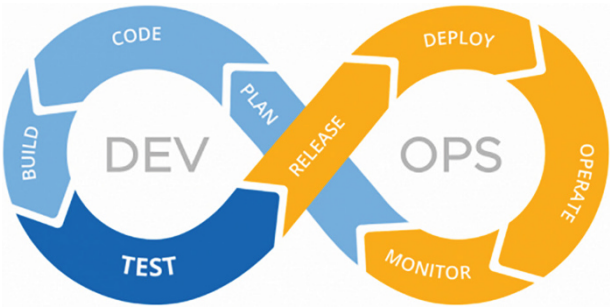

FIGURE 2-1: The DevOps cycle.

1. **Plan.**
   - Gather requirements and feedback to define a product road map consisting of prioritized user stories.
   - Create a backlog of small tasks that reflect the user stories.
   - Define sprints to plan which work should be completed in a set time.
2. **Code.**
   - Task developers with writing the code to deliver the tasks that are part of the sprint.
   - Commit code to a shared, centralized, version control system.
   - Peer-review committed code before merging with the existing codebase.
3. **Build.**
   - Run a series of automatic tests, such as linting, unit tests, integration tests, and smoke tests. These tests should verify that submitted changes don't destabilize the existing code base and should ensure that submissions meet the fundamental requirements for production readiness.
   - Build the source code into the desired format.
4. **Test.**
   - Deploy the code to a staging environment (ideally using infrastructure as code to ensure the environment is as close to production as possible).
   - Execute automated tests to detect functional, quality, security, and performance flaws.
   - Depending on the maturity of testing and the operational comfort with automation, hold the new build for manual testing (such as User Acceptance Testing) or automatically release it.
5. **Release.**
   - Confirm that the code is ready for release to production.
   - Depending on the maturity of the organization in DevOps terms, set up a pipeline to release new builds to production automatically. Alternatively, use feature flags, scheduled releases, or manual approval processes.

6. **Deploy.**
    - Deploy released builds to the production environment, and manage it using the same infrastructure as code as the staging environment to ensure the released build will be successful.
    - Use a green/blue deployment strategy to deploy in parallel with the existing production environment, and seamlessly switch to the new codebase when ensured that the new environment works as expected. Switch users back to the previous environment if the release introduces a problem and until a fix is released.
7. **Operate.**
    - Capture information from the operations team, the hosting provider, and the users.
    - Give users a mechanism to provide feedback from within the platform.
    - Task the operations team or the hosting provider with configuring auto-scaling based on the application load.
8. **Monitor.**
    - Capture performance metrics, logging data, and errors.
    - Feed this data, and the data captured during the operate phase, back into the next planning phase.
    - Capture any inefficiencies in the pipeline itself.

The loop then restarts at the plan stage. In reality, though, there should be no start or end to the loop — it's one long continuous process of integration, delivery, deployment, and feedback.

Exploring Key DevOps Principles

REMEMBER

As you understand the processes and frameworks that DevOps introduces, you also need to understand the key principles that most DevOps implementations incorporate:

- **Automation:** By reducing the number of manual steps in your release process, you can move toward faster, higher-quality launches. Removing manual steps minimizes the risk of human error, or missed steps, and is ultimately a force

multiplier for developers' effort. It also allows developers and operators to focus on more productive tasks.

- **Iterative development:** By making a high number of smaller, more incremental changes, you can reduce the risk each change introduces. Each iteration will provide another opportunity to "go around the loop" and gather feedback to further inform future iterations.
- **Self-service:** The release cycle is accelerated by enabling self-service for developers to deploy on demand without the need for manual intervention by a traditional IT organization.
- **Collaboration:** Collaboration between the development and operations teams is critical to ensure that operational feedback is pushed into the planning and coding phases.
- **Knowledge sharing:** In order for development teams to be conscious of security risks, they need to work closely with security engineers, operations teams, and compliance teams. Everyone working on the delivery should be familiar with application security, security testing, security engineering, and the operational considerations.
- **Observability:** The correlation of logs, metrics, traces, and dependencies to identify "unknown unknowns" and good and bad patterns should be part of DevOps from the moment the process starts.
- **Traceability:** Traceability allows you to follow a user story or transaction from beginning to end across the development cycle to where requirements are implemented in the code. This can play a crucial part in helping achieve compliance, reduce bugs, ensure secure code in application development, and help code maintainability.

Ensuring Continuity

REMEMBER

Continuity runs through every aspect of DevOps. DevOps came to disrupt waterfall release processes, and as such is focused on maintaining a continuous flow across the stages from the very beginning.

Here are six ways that DevOps disrupts traditional programming methods by enabling that flow:

- **Continuous integration (CI):** CI plays a pivotal role in a DevOps pipeline. It encourages developers to submit their code to a central code repository multiple times a day. Regularly integrating smaller chunks of code significantly reduces the likelihood of bad code moving down the pipeline and causing service interruptions. Some of the largest organizations that implement DevOps practices commit new code hundreds of times a day.

  Another critical aspect of CI is automated testing. Before developers commit their code to the master branch, they create builds of the project to determine if the new code is compatible with the existing code. If the build is successful, developers submit their code to the shared repository. CI relies on a version control system, a central code repository that helps teams track code changes and manage merge requests.

- **Continuous delivery (CD):** CD is all about releasing code updates frequently and quickly. CD relies on developers to manually deploy code to production, as opposed to continuous deployment, which takes an automated approach to code release. For CD to yield positive results, it's paramount to create a repeatable system that will push the code through the DevOps pipeline.

  Developers working in a CD environment must keep in mind that their code contributions may get deployed to production at any moment. Having passed all tests and reviews, code updates can be released to production with a click of a button.

  Numerous benefits come with continuous delivery, including the following:

  - It eliminates the risk of downtime and other performance issues because code changes are usually minor.
  - It enables organizations to release high-quality features much easier, ensuring faster time to market, and ultimately eliminating fixed costs associated with the deployment process.

- **Continuous monitoring:** Continuous monitoring builds on the concepts of CI/CD and ensures that the application

performs without issues. DevOps teams implement monitoring technologies and techniques to keep track of how the application is behaving.

DevOps teams monitor logs, apps, systems, and infrastructure. When a problem is detected, DevOps teams can quickly revert the app to a previous state. During that time, the team works on resolving known issues without making the user aware that the code is updated continuously.

Continuous monitoring also helps DevOps teams detect issues that are hindering productivity in the pipeline. After each release cycle, teams should optimize the CI/CD pipeline to eliminate any bottlenecks and ensure a smooth transition of code from one stage to the next.

- **Continuous testing:** Continuous testing allows faster, higher-quality releases by ensuring each new iteration has passed testing for functional, security, compliance, and user acceptance. Test-driven development can be used to ensure that code is written to fulfill a set of criteria that are defined in unit tests before the actual code is written.
- **Continuous feedback:** Continuous feedback helps DevOps teams thrive. Developers need actionable information from different sources to help them improve the overall quality of the application. Without feedback, DevOps teams can fall victim to spending time on building products that don't bring value to stakeholders or customers.
- **Continuous improvement:** Continuous improvement looks for ways to minimize waste, optimize for speed, reduce costs, reduce risk, and improve on the product or service being delivered with each loop of the DevOps cycle. These improvements include both the product and the processes involved in delivering the product.

  DevOps teams usually gather feedback from stakeholders, end users, and software-based monitoring technologies before, during, and after the release cycle. They collect feedback through various channels, such as social media and surveys, or by discussing with colleagues.

  Teams need to have procedures that will help them sift through the feedback they've gathered to extract the most valuable information. Misleading or inaccurate feedback could prove to be detrimental to the entire development process.

Defining DevSecOps and Shift Left

As a trend gains momentum and establishes success, sub-trends emerge to highlight specific elements. Marketing teams take advantage of these sub-trends to popularize their companies' perspectives and claim a piece. DevOps is no different. Two important branches to be aware of are DevSecOps and Shift Left.

When working in a DevOps cycle, the speed at which you can release software increases exponentially with multiple (and sometimes even hundreds of) releases to production in a single day. This poses a challenge for traditional ways of ensuring secure code and platforms because you can't just evaluate each release at the end of the cycle before it goes live. Security either becomes a roadblock to iteration or is sacrificed for the sake of faster releases.

DevSecOps distributes the responsibility for securing across the DevOps team. In the same way that DevOps brings operational considerations into the development life cycle, DevSecOps bakes in security considerations and risk mitigation to each stage of the DevOps life cycle.

In the context of DevSecOps, the term *Shift Left* refers to moving security considerations from right (the end) to left (the start) of the DevOps delivery process. Security skill sets are required right from the start of the development process to ensure that the DevOps team is incorporating security through planning, development, delivery, and operational phases. The benefit of this is that the development team is implementing security from the ground up, and any security defect or risk is identified and addressed much earlier, long before the code hits production.

Recently the term *Shift Left* has also come to refer to other critical quality assurance, testing, and operating processes that should be incorporated in the early stages of the software development life cycle.

Introducing DevOps for Infrastructure

In practice, DevOps has mostly consisted of developers applying their powerful toolchain and Agile processes to traditional Ops responsibilities. Developers' frustration with the long IT waiting

times impacting their ever-shortening development cycles played a big role in the growth of the DevOps movement. Consequently, DevOps toolchains available in the market have assumed a developer skill set.

IT has been reluctant to embrace the DevOps model, but it has also never been "given the key." This lack of accessibility has prevented people in IT Ops roles from learning and adopting such technologies.

REMEMBER

DevOps for infrastructure tries to address this problem. It carves out an IT path to DevOps, unlike other approaches that have been from the Dev point of view. DevOps for infrastructure is not about introducing new DevOps-enabling capabilities, but rather translating established DevOps principles and processes into language that can help IT professionals learn and adopt the everything-as-code approach. DevOps for infrastructure converges the IT automation requirements with the agility and scalability of DevOps continuous delivery and release automation (CDRA) tools.

As organizations adopt DevOps principles, the traditional infrastructure team becomes a service provider to internal teams. This means providing not just infrastructure, but a platform that removes friction between the provider and the consumer — it becomes invisible.

So, when we talk about DevOps for infrastructure, we're really talking about how we can apply the principles of DevOps (including automation, iterative development, self-service, continuous improvement, collaboration, and feedback) and the mechanics of DevOps (the DevOps cycle and phases) to the delivery of an infrastructure platform that provides services to support the needs of internal development teams.

In the world of infrastructure, this typically boils down to three things, which we explore in more detail in upcoming chapters:

- **The language:** *Infrastructure as code* (IaC) is the ability to express and manipulate infrastructure the same way as application code. The goal of IaC is to automate the provisioning and maintenance of the infrastructure to the highest degree possible. What's coded is the Ops intent via policies and variables and the end-user needs via inputs and tags.

  Infrastructure as code is a fundamental part of any DevOps implementation.

- **The implementation:** *Infrastructure pipelines* are continuous delivery pipelines, specifically built to manage infrastructure in an automated way. CI/CD tools are the skeleton of any DevOps implementation. Infrastructure pipelines bring the best practices from the CI/CD world to processes traditionally driven by IT departments. Direct integrations to IaC tools and low code pipeline interfaces can catalyze DevOps adoption in IT.
- **The process:** *Iterative development with GitOps* is the collaborative updating of the infrastructure by the developers and/or operators as infrastructure needs evolve. Iterative development can use imperative or declarative language and ensures proper collaboration and accountability by leveraging Git-based version control systems (VCSs).

IN THIS CHAPTER

- Uncovering some key concepts
- Understanding how infrastructure as code works
- Looking at some implementation examples

Chapter 3
Speaking the Language: Infrastructure as Code

Modern, cloud-like infrastructure is highly dynamic in nature, with changes being made almost constantly. When configuration changes are not consistently applied across servers, each environment becomes a unique "snowflake" with its own unique peculiarities. That can lead to deployment problems, with these environments becoming inconsistent, unreliable, and fragile.

Infrastructure as code (IaC) solves such problems by capturing all the components and configuration of your infrastructure (such as virtual machines [VMs], containers, networks, load balancers, and topology) in a code format that describes the desired end state. A provider can then interpret this code format to provision, configure, and manage the infrastructure in repeatable, reliable, and on-demand fashion. The code description can then be source-controlled and managed as a developer would an application's source code.

Changes to the configuration of an environment are made in the source code (not the target environment), which is then interpreted and fulfilled by the provider. The change is self-documenting

because the code is in a well-known format that describes the end state, and the commit message describes what has changed and by whom.

Changes can be staged across multiple environments to ensure consistency. Environments can also be rolled back to previous states should a change be detrimental to the environment simply by using source control versioning (see Chapter 4).

Mastering Some Key Concepts

When implementing DevOps for infrastructure, you'll likely come across an abundance of unfamiliar terms. Here are a few of the most common ones you should know.

Declarative code versus imperative code

REMEMBER

IaC is typically *declarative* — it describes the desired state of the infrastructure, but it doesn't describe how to achieve that desired state. How to achieve the desire state is left up to the IaC provider.

Declarative code differs from the *imperative* approach, which describes a series of steps or commands executed to achieve the desired state. Imperative code typically doesn't have the same scalability, reusability, or resilience as declarative code.

Let's say you want a sandwich — who doesn't like a sandwich? But you don't want to make the sandwich yourself. Instead, you want to describe that sandwich and have it made for you.

With a declarative approach, you would describe (or declare) what the result should look like:

```
sandwich:
    bread: brown
    fillings:
    - cheese
    - ham
    salad:
    - lettuce
    - tomato
    condiment:
    - mayonnaise
```

You expect that the person making your sandwich understands those requirements and can translate them into the sandwich you want. You don't tell them how to make the sandwich, where the ingredients are, or how those ingredients are to be put together. The responsibility for making the sandwich lies with the person making it.

With an imperative approach, you describe a series of commands that, if followed correctly, will result in your sandwich being made to your specifications. So, your imperative request would go something like this:

1. **Get two slices of brown bread from the bread bin.**
2. **Get the mayonnaise, cheese, ham, tomato, and lettuce from the fridge.**
3. **Use a knife to spread mayonnaise on one side of each slice of bread.**
4. **Place two slices of ham on a slice of bread.**
5. **Slice some cheese.**

And so on.

Anyone who can understand your instructions should be able to make the sandwich to your specifications. All the responsibility lies in the accuracy of the instructions. What if something goes wrong? You either must anticipate that problem and write instructions for handling it or expect the making of your sandwich to fail until you've fixed the problem.

Idempotency


REMEMBER

Deployed environments should be configured in the same way regardless of the starting state of the environment, a property known as *idempotency*. Idempotency can be achieved by modifying the existing environment to the desired state (in a mutable infrastructure that can be "mutated" or changed), or by destroying the current state and deploying the desired one (in an immutable infrastructure that can't be "mutated" or changed).

When we talk about code being *idempotent*, it means that running the same code with the same input multiple times will have no further effect on the subject after the first time it has been performed.

Going back to the sandwich example, if you ask the same person for the same sandwich more than once, will you get just one sandwich or multiple sandwiches? Or will multiple sandwich makers end up fighting for the resources to make the sandwich, resulting in an all-out battle royal to crown the one true sandwich maker?!

Both declarative and imperative instructions can be idempotent, but it's once again a case of responsibility. With imperative instructions, you must include some checks for each instruction to see if the subject exists before you try to create it. With declarative code, it rests on the person who is fulfilling the request.

When we're talking about IaC, idempotency can also depend on the mutability of the deployed infrastructure.

Some IaC providers are platform-specific, such as AWS CloudFormation (`https://aws.amazon.com/cloudformation`) or Azure Resource Manager (`https://azure.microsoft.com/en-gb/features/resource-manager`). Other IaC providers are more modular and can deploy to multiple platforms, such as Terraform (`www.terraform.io`) or Pulumi (`www.pulumi.com`). Kubernetes is also a great example of IaC.

Outlining the Mechanics of Infrastructure as Code

Let's start with the mechanics of IaC. What are the components, and how do they work together to achieve the desired outcome?

The code

In Figure 3-1 there are two blocks of code in the IaC section on the left. This is typical of any IaC implementation. Some code defines variables; other code defines the desired infrastructure state.

Variables are values that change on a per-deployment basis, like the name of a VM or the network port on which to load-balance. These can be provided at execution time as inputs, as environment variables, or in an answer file or application programming interface (API) call.

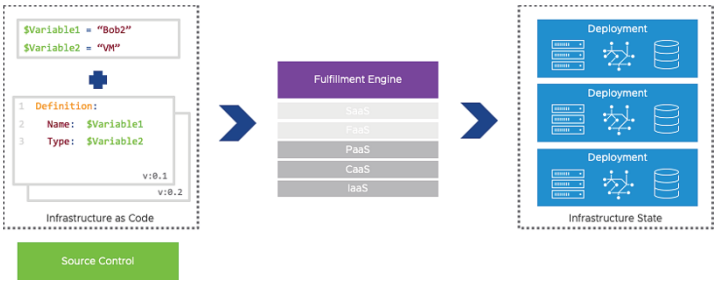

FIGURE 3-1: The mechanics of IaC.

The *definition* is a code description of a generic desired end state; it uses the variables provided to customize each deployment of the end state. It doesn't typically describe *how* to get to the desired state; that's determined by the fulfillment engine, explained in the next section.

Together, the variables and definition describe the desired infrastructure state in a well-defined format. That format could be a Domain Specific Language like HashiCorp Configuration Language (HCL) for Terraform or a more generic language such as JSON or YAML (for example, in AWS CloudFormation).

TIP

Defining IaC enables you to manage these text files using the same techniques that software developers use to manage their code. The files can be edited with any text editor and managed within a source control system such as Git, with all the advantages of tightly controlled versioning. For example, the system can

- Roll back to a previous configuration with confidence.
- Maintain environments at different versions.
- Automatically document changes, including what's changed, who changed it, and why.

The fulfillment engine

The code is processed by a *fulfillment engine*, which is responsible for implementing the code definition on the end platform. The fulfillment engine knows how to communicate with the various platform endpoints, such as infrastructure as a service (IaaS), containers as a service (CaaS), or platform as a service (PaaS). The fulfillment engine translates the generic definition that the code provides into a specific infrastructure deployment and state.

The fulfillment engine processes changes to the variables or definition to update the infrastructure state. Depending on the infrastructure's mutability and the type of change, which may mean either redeploying or modifying the deployment.

For example, if you've deployed a traditional three-tier application, changing the *number* of front-end web servers in a cluster (scale-out/scale-in) could deploy or remove a specified number of front-end web servers. But changing the underlying operating system of those servers would mean deploying an entirely new web server.

State management

Management of state is a tricky problem. How do you know that the current infrastructure state is as you described it? In an ideal world, any changes to the infrastructure's state are made through changes to your code and are then fulfilled by the IaC provider. You can enforce this by restricting who has access to the infrastructure (see Chapter 4).


WARNING

Often, despite your best intentions, the infrastructure state changes over time. You need to be able to recognize when this has happened and either remediate these changes to ensure the infrastructure matches the desired state or capture them in code to update the desired state.

Because the objective of infrastructure is always to support an application and users, you need to consider application and user state. This typically involves separating this state out from the infrastructure so you can manage it as *stateless.*

Implementing Infrastructure as Code

In this section, we look at a couple of simple examples of implementing IaC to deploy, update, and delete a simple vSphere VM.

A Terraform example


TECHNICAL STUFF

With Terraform you typically define variables in a `variables.tf` file and inputs for the variable values in a `.tfvars` file. The desired state definition is captured in a `.tf` file. The fulfillment engine is in the form of the Terraform command line executable,

with providers configured for various infrastructure platforms that abstract an upstream. The main `.tf` file also configures the provider connection and retrieves any data required.

Because these files are simple text files written in HCL, they can easily be stored in version control systems.

variables.tf

This file defines the variables that are consumed by the `main.tf` file.

```
variable "vsphere_server" {
  description = "vSphere server"
  type        = string
}
variable "vsphere_user" {
  description = "vSphere username"
  type        = string
}
variable "vsphere_password" {
  description = "vSphere password"
  type        = string
  sensitive   = true
}
variable "datacenter" {
  description = "vSphere data center"
  type        = string
}
variable "cluster" {
  description = "vSphere cluster"
  type        = string
}
variable "datastore" {
  description = "vSphere datastore"
  type        = string
}
variable "network_name" {
  description = "vSphere network name"
  type        = string
}
```

```
variable "template_name" {
  description = "vSphere template name"
  type        = string
}
variable "vm_name" {
  description = "vSphere VM name"
  type        = string
}
```

variables.tfvars

This file provides input values for the variables defined in the `variables.tf` file. If this file is not provided, Terraform will prompt the end user for manual inputs.

```
vsphere_server     = "sc2vc03.cmbu.local"
vsphere_user       = "tfuser@vsphere.local"
vsphere_password   = "VMware1!"
datacenter         = "sc2dc03"
datastore          = "sc2c02vsan01"
cluster            = "sc2c04-ATMM"
network_name       = "Mgmt VMs"
template_name      = "Templates/ubuntu1804"
vm_name            = "iac-ubuntu"
```

main.tf

The `main.tf` file first configures the vSphere provider and then configures data sources to retrieve the object IDs for the vSphere datacenter, compute cluster, datastore, network, and template. Finally, it defines a new resource — a vSphere VM called iac-ubuntu.

```
provider "vsphere" {
  vsphere_server = var.vsphere_server
  user           = var.vsphere_user
  password       = var.vsphere_password
  # If you have a self-signed cert
  allow_unverified_ssl = true
}

data "vsphere_datacenter" "dc" {
  name = var.datacenter
}
```

```
data "vsphere_compute_cluster" "cluster" {
  name          = var.cluster
  datacenter_id = data.vsphere_datacenter.dc.id
}

data "vsphere_datastore" "datastore" {
  name          = var.datastore
  datacenter_id = data.vsphere_datacenter.dc.id
}

data "vsphere_network" "network" {
  name          = var.network_name
  datacenter_id = data.vsphere_datacenter.dc.id
}

data "vsphere_virtual_machine" "ubuntu" {
  name          = "/${var.datacenter}/vm/
                  ${var.template_name}"
  datacenter_id = data.vsphere_datacenter.dc.id
}

resource "vsphere_virtual_machine" "iac-ubuntu" {
  name             = var.vm_name
  resource_pool_id = data.vsphere_compute_cluster
                     .cluster.resource_pool_id
  datastore_id     = data.vsphere_datastore.
                     datastore.id
  num_cpus = 2
  memory   = 1024
  wait_for_guest_net_timeout = -1
  wait_for_guest_ip_timeout  = -1
  guest_id = "ubuntu64Guest"

  network_interface {
    network_id = data.vsphere_network.network.id
  }

  disk {
    label            = "disk0"
    thin_provisioned = true
    size             = 32
  }
```

```
  clone {
    template_uuid = data.vsphere_virtual_machine.
                    ubuntu.id
  }
}
```

When the desired state (consisting of the variables, and definition) is processed by the fulfillment engine, a new VM is created on the vSphere endpoint specified. Terraform will print out a list of changes that will be made and then apply those changes to create the new VM.

```
terraform apply --var-file=variables.tfvars
  --auto-approve

Terraform used the selected providers to generate
  the following execution plan. Resource actions
  are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # vsphere_virtual_machine.iac-ubuntu will be
    created
  + resource "vsphere_virtual_machine" "iac-
    ubuntu" { ... }

Plan: 1 to add, 0 to change, 0 to destroy.
vsphere_virtual_machine.iac-ubuntu: Creating...
vsphere_virtual_machine.iac-ubuntu: Still
  creating... [10s elapsed]
vsphere_virtual_machine.iac-ubuntu: Still
  creating... [20s elapsed]
vsphere_virtual_machine.iac-ubuntu: Still
  creating... [30s elapsed]
vsphere_virtual_machine.iac-ubuntu: Still
  creating... [40s elapsed]
vsphere_virtual_machine.iac-ubuntu: Still
  creating... [50s elapsed]
vsphere_virtual_machine.iac-ubuntu: Still
  creating... [1m0s elapsed]
```

```
vsphere_virtual_machine.iac-ubuntu: Creation
  complete after 1m5s
  [id=4226bba0-53bb-2466-a286-6d6c891633b4]

Apply complete! Resources: 1 added, 0 changed, 0
  destroyed.
```

Terraform stores the state of the deployment in a separate `.tfstate` file that is created. Running a Terraform `apply` with the same variables and definition will achieve the same result (which we previously defined as idempotency). The current state is refreshed and compared to the desired state and, because no changes have been made to either, no action is required.

If either the desired state (variables plus definition) or the actual state of the infrastructure has changed, then Terraform will reconcile the actual state to match the desired state. So, you could manually modify the VM name to iac-changed and run the Terraform `apply` command. Terraform would see that the actual state has diverged from the desired state and remediate it, and the VM would be renamed back to iac-ubuntu.

If you changed the desired state by modifying the input variable for the VM name to iac-changed and ran the Terraform `apply` command, Terraform would see the desired state had diverged from the actual state and remediate it: The VM would be renamed iac-changed.

```
terraform apply --var-file=variables.tfvars
  --auto-approve
vsphere_virtual_machine.iac-ubuntu: Refreshing
  state...
  [id=4226bba0-53bb-2466-a286-6d6c891633b4]

Terraform used the selected providers to generate
  the following execution plan. Resource actions
  are indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  # vsphere_virtual_machine.iac-ubuntu will be
    updated in-place
```

```
  ~ resource "vsphere_virtual_machine" "iac-
    ubuntu" {
        id = "4226bba0-53bb-2466-a286-
        6d6c891633b4"
      ~ name = "iac-ubuntu" -> "iac-changed"
        # (65 unchanged attributes hidden)
        # (3 unchanged blocks hidden)
    }

Plan: 0 to add, 1 to change, 0 to destroy.

vsphere_virtual_machine.iac-ubuntu: Modifying...
  [id=4226bba0-53bb-2466-a286-6d6c891633b4]
vsphere_virtual_machine.iac-ubuntu: Still
  modifying... [id=4226bba0-53bb-2466-a286-
  6d6c891633b4, 10s elapsed]
vsphere_virtual_machine.iac-ubuntu: Modifications
  complete after 17s
  [id=4226bba0-53bb-2466-a286-6d6c891633b4]

Apply complete! Resources: 0 added, 1 changed, 0
  destroyed.
```

In either case, you're modifying a mutable property (the VM name), so Terraform simply updates in place. If the property you update is immutable (for example, the template on which the VM is based), then Terraform will destroy and re-create the entire VM. Terraform will show which objects will be updated and which will be re-created, either with the `plan` command or as part of the approval for the `apply` command.

Finally, you can use `terraform destroy` to remove any objects defined in the desired state. The variables and definition will still exist, but the objects themselves will be removed.

A vRealize Automation example

vRealize Automation offers a "low code" approach to getting started with IaC. The cloud template designer within the platform, shown in Figure 3-2, provides a palette of components that you can drag onto a design canvas. The visual representation shown in the design canvas is rendered from the YAML in the code pane to the right of the canvas.

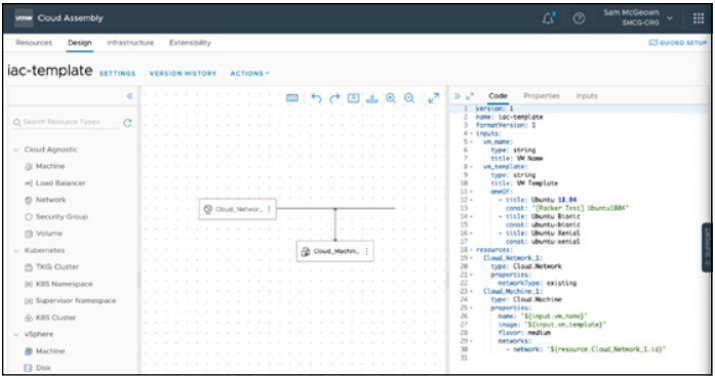

FIGURE 3-2: vRealize Automation cloud template designer.

The low code approach enables cloud template designers to quickly build out the YAML required and learn the structure of the code until they're more familiar with it. Everything on the canvas can be modified by editing the code. The graphical interface can configure most of the options, but some more advanced component features need to be created using the code editor.

In the vRealize Automation world, the vSphere configuration is abstracted away as an endpoint that the fulfillment engine will use to implement the desired state described in the cloud template definition and the input variables.

TECHNICAL STUFF

The variables are created as part of the cloud template code using the inputs section. This is defining the vm_name and the vm_template variables. The resources section defines an existing network and a Cloud.Machine (which is an abstraction of the vSphere VM). The input variables are used to provide values for the cloud machine name and image (an abstraction of the vSphere template).

```
formatVersion: 1
inputs:
  vm_name:
    type: string
    title: VM Name
  vm_template:
    type: string
    title: VM Template
```

```
    oneOf:
      - title: Ubuntu 18.04
        const: '[Packer Test] Ubuntu1804'
      - title: Ubuntu Bionic
        const: 'ubuntu-bionic'
      - title: Ubuntu Xenial
        const: 'ubuntu-xenial'
resources:
  Cloud_Network_1:
    type: Cloud.Network
    properties:
      networkType: existing
  Cloud_Machine_1:
    type: Cloud.Machine
    properties:
      name: '${input.vm_name}'
      image: '${input.vm_template}'
      flavor: 'medium'
      networks:
        - network: '${resource.Cloud_Network_1.
          id}'
```

vRealize Automation can be configured to provide one-way synchronization from a Git repository, allowing the release of new cloud template code directly from source control. Different vRealize Automation projects can be mapped to a specific branch within a repository, allowing for the promotion of code through environments using Git as the source of truth. Cloud templates are versioned and released through the Git repository to enable controlled versioning of deployed infrastructure state through the described desired state.

After you've added a Git integration endpoint, you can configure projects to synchronize with the intended branch. For example, in Figure 3-3, the development project is synchronizing with the development branch of the smc-org-content repository, the staging project with the staging branch, and the production project with the main branch.

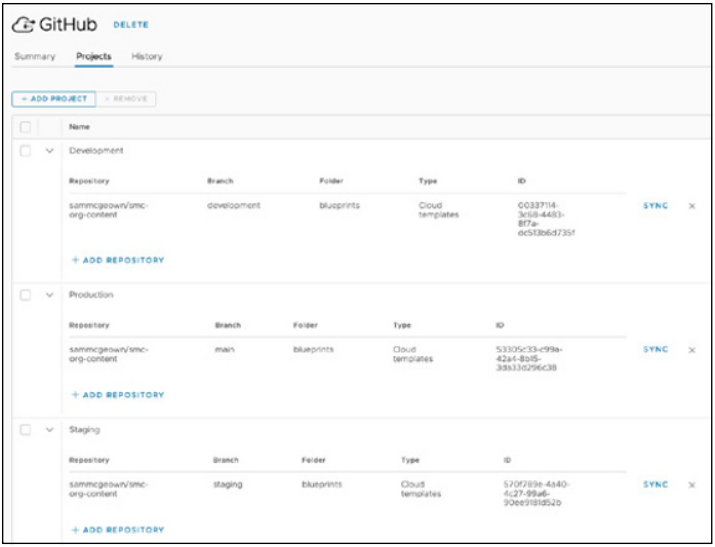

FIGURE 3-3: GitHub integration synchronizing with Projects.

To add the new YAML cloud template iac-template to the staging project in vRealize Automation in Figure 3-3, you simply check out the staging branch, create a new folder with the YAML template inside, and add two lines of metadata to the code you generated earlier:

```
version: 1 # version of the cloud template
name: iac-template # name of the cloud template
```

This metadata enables the cloud template to be created and a version released. In this example, we're using Visual Studio Code as our integrated development environment (IDE), as shown in Figure 3-4:

```
git checkout staging
mkdir -p blueprints/iac-template
# Create the blueprint.yaml file
git add blueprints/iac-template/blueprint.yaml
git commit -m "Add iac-template"
git push
```

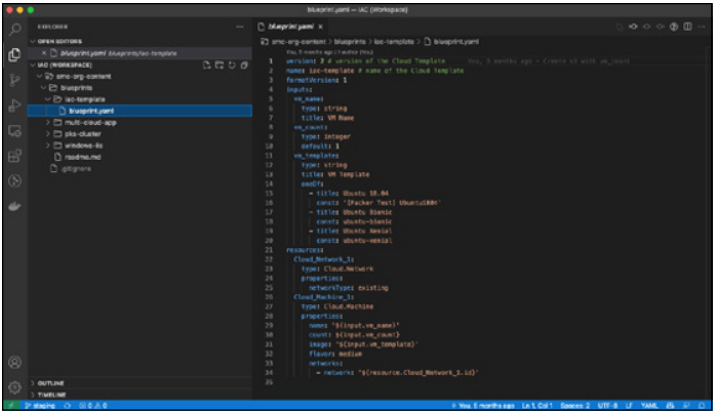

FIGURE 3-4: Using Visual Studio Code to manage a cloud template.

When the repository synchronizes, the new iac-template cloud template is created, as shown in Figure 3-5.

FIGURE 3-5: A cloud template is created from the GitHub source.

When deploying the new cloud template, you're prompted to enter the vm_name and vm_template values (the variable part of the desired state), which is then used with the cloud template (the definition part of the desired state) to deploy the infrastructure (see Figure 3-6).

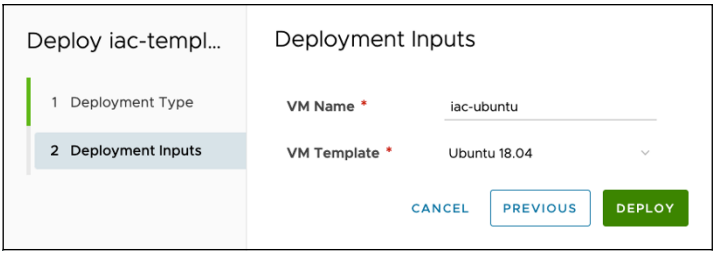

FIGURE 3-6: Deploying a cloud template with input variables.

To update the cloud template, you can increment the version property in the YAML, and make your changes. When these are pushed up to the Git repository and synchronized with the fulfillment engine, you have a new version of the cloud template available and can update the existing deployment to the new version.

For example, the updated YAML shown here increments the version to 2, adds a vm_count input variable, and uses that variable to specify the number of VMs to deploy.

```
version: 2 # version of the cloud template
name: iac-template # name of the cloud template
formatVersion: 1
inputs:
  vm_name:
    type: string
    title: VM Name
  vm_count:
    type: integer
    default: 1
  vm_template:
    type: string
    title: VM Template
    oneOf:
      - title: Ubuntu 18.04
        const: '[Packer Test] Ubuntu1804'
      - title: Ubuntu Bionic
        const: ubuntu-bionic
      - title: Ubuntu Xenial
        const: ubuntu-xenial
resources:
  Cloud_Network_1:
    type: Cloud.Network
    properties:
      networkType: existing
  Cloud_Machine_1:
    type: Cloud.Machine
    properties:
      name: '${input.vm_name}'
      count: ${input.vm_count}
```

```
      image: '${input.vm_template}'
      flavor: medium
      networks:
        - network: '${resource.Cloud_Network_1.
          id}'
```

vRealize Automation stores the state as a deployment within the platform, as shown in Figure 3-7. The desired state can be updated using a day-2 action that enables you to modify the value of the input variables, or by updating the deployment with a new definition stored in the cloud template.

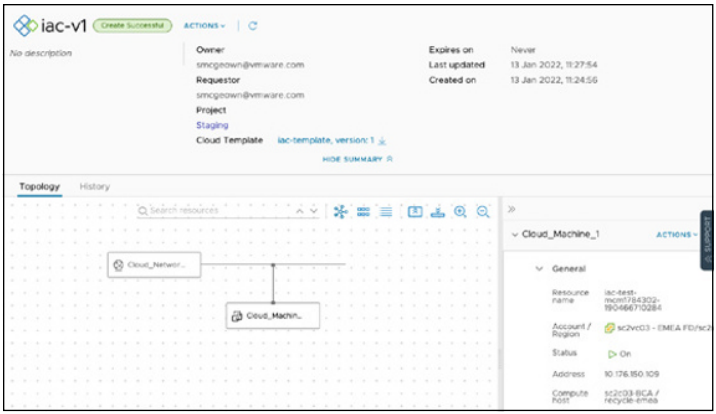

FIGURE 3-7: Deployment state shown in vRealize Automation.

Depending on the mutability of the object being updated, this will either delete and re-create the object, or update a mutable property, as shown in Figure 3-8. The plan stage shows which components will be updated and which will be re-created.

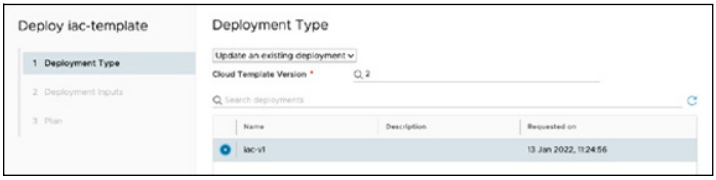

FIGURE 3-8: Updating an existing deployment to a new cloud template version.

IN THIS CHAPTER

- Getting familiar with infrastructure pipelines
- Identifying the principles of GitOps
- Understanding the GitOps process
- Discovering how to implement GitOps

Chapter 4
Infrastructure Pipelines and GitOps

This chapter takes a brief but illuminating look at infrastructure pipelines and their role in DevOps for Infrastructure. Pipelining is an essential component of any DevOps implementation, and infrastructure use cases are no different. GitOps, the practice of using Git as the source of truth, brings IaC and infrastructure pipelines together through merge requests. In this chapter, we also examine GitOps, looking at its principles, processes, and implementation through simple examples.

Understanding the Role of Infrastructure Pipelines

In their simplest form, infrastructure pipelines capture the already well-defined processes that you have for deploying and operating infrastructure and encapsulate them as a series of automated tasks that are performed sequentially. Infrastructure pipelines should apply the principles seen in DevOps to platform delivery, such as the use of source control, infrastructure as code (IaC),

continuous integration (CI), testing, and continuous delivery (CD). This could be the build, test, and release stages used in this chapter's example, but they could also capture processes around the continuous monitoring concept (for example, monitoring load on the infrastructure and auto-scaling in response).

Infrastructure pipelines are typically *imperative* in nature. That is, they perform a sequence of well-defined tasks in a specific order. Those tasks may involve declarative code for an individual task, but simply due to the sequential nature of a pipeline, they aren't declarative themselves.

We'll use an example that's common to many people who manage infrastructure — virtual machine (VM) image management. Whether it's a vSphere template, an Amazon Machine Image (AMI), or an Azure VM image, there are typically several tasks that need to be completed before you can release a new image:

1. **Build/update the image.**

   Either you need a new image or you need to update the existing image and slipstream the patches or install the new software.

2. **Test.**

   Can you deploy this image successfully? Does it have the right tooling, monitoring, and agents? Does it meet your security standards? Can the applications teams still deploy what they need to deploy on this image?

3. **Release.**

   If you're happy with the new or updated image, how do you get this into the hands of your end users? It may be updating a repository, Amazon Web Services (AWS), Azure, or vSphere Content Library.

These three stages (and there may be more — this is just an example) can be built into a pipeline that may look like Figure 4-1.

Depending on the tools used, the image configuration can be captured as code, and we can bring the code under source control. This means the process for creating or updating an image involves creating or updating the code and committing that code to source control. This triggers an execution of the pipeline.

FIGURE 4-1: Example for infrastructure pipeline.

This example pipeline incorporates several of the tools and principles covered elsewhere in this book. Being able to describe our image (infrastructure) as code enables us to bring it under source control. The pipeline automates the process of building an artifact from the source code. We're incorporating testing early into the life cycle of the artifact, and this includes "shifting left" security and vulnerability scanning, as well as the more common integration and functional testing. Finally, the release stage incorporates automated release to production with a high level of confidence because we've already tested and approved of the artifact.

Getting Familiar with GitOps

Git is a free, open-source distributed version control system. *GitOps* is a method of using Git to manage DevOps. Originally conceived by Alexis Richardson of Weaveworks, GitOps places Git as the single and authoritative source of truth for declarative infrastructure and applications. GitOps is aimed primarily at cloud-native applications running on Kubernetes clusters, but its principles can be applied more broadly where an entire system can be captured declaratively.

GitOps requirements

Here are the fundamental requirements to begin managing infrastructure or applications using a GitOps workflow:

- **Declared desired state:** The desired state of the infrastructure or application needs to be captured declaratively as code to provide a definitive single source of truth for the desired state. The implementation of the desired state needs to be idempotent.
- **Versioned desired state:** By storing declarative code for the desired state in a version control system, you create a single, authoritative source of truth for that desired state. Each change to state is stored as a version to which you can roll

back. Commit logs and signed commits, along with code reviews, provide strong controls around what's committed and by whom, and ensure that changes are reviewed before being applied.

- **Automatically applied desired state:** When a new or updated desired state is pushed to Git and approved to merge with the existing code, that state should be automatically applied to the system. One of the major advantages of this is that you don't need to provide credentials to make changes. The pipeline or workflow reconciles the desired state with the actual state.
- **Monitor and remediate desired state:** If the current state and the desired state diverge, then the GitOps system should have a process by which the state deviation is monitored and remediated, so the running state is always as described in the desired state. This ensures that any changes made to the running system are made in the source code and not via manual changes.

The GitOps process

Much like DevOps, GitOps can be interpreted in different ways depending on the tools used to achieve it. There is no single way to implement GitOps, but we can distill a generic process to describe what it could look like:

1. **Write code.**

   Create a new branch in Git and describe the desired state in code.

2. **Merge the code.**

   This is typically done using a pull request (GitHub) or a merge request (GitLab) and reviews from peers.

3. **Build.**

   Like the DevOps CI phase, the code triggers a build pipeline to create and test artifacts (such as a new container or a VM image) and update the desired state for the environment.

4. **Deploy.**

   The new artifacts and state trigger a deployment pipeline that releases the changes to the environment to match the new desired state.

5. **Monitor.**

   The state of the environment is monitored to ensure that the current state is the same as the desired state and resolves any discrepancies.

The deployment in the GitOps process can be either a "push" (in which the deployment pipeline is triggered when changes are made to the desired state for the environment) or a "pull" (where an agent, such as an operator in Kubernetes, observes the repository and pulls configuration when it differs from the current state).

The advantage of an agent pull-based versus push-based setup is that the agent can compare the current and desired state and remediate if required, so manual changes to the current state should be overridden. The push-based setup will update the environment only when a change is made to the repository to trigger the pipeline.

Implementing GitOps

GitOps was first conceived to operate Kubernetes (`https://kubernetes.io/`) clusters, so let's look at an implementation with Flux (`https://fluxcd.io/`), which, same as Kubernetes, is currently a Cloud Native Computing Foundation (CNCF) incubating project. An alternative tool for use with Kubernetes may be Argo CD, which works in a similar fashion but with a more graphical interface and is also a CNCF incubating project.

Kubernetes and Flux

Flux works by bringing together several tools in a pipeline approach. It deploys Kubernetes controllers for synchronizing with Git (source-controller), customizes source configurations with Kustomize (kustomize-controller) or Helm (helm-controller), and handles incoming events from systems such as GitHub or Jenkins and outgoing notifications to systems such as Slack or Discord via a notification controller (notification-controller).

With Flux, the IaC desired state is defined by a YAML manifest or a Helm chart, using variables to customize the desired state in the form of a Kustomization manifest, or a `values.yaml` file for Helm.

The following simple example from the Flux Getting Started guide demonstrates how to:

- Bootstrap Flux and create a GitRepository for the cluster.
- Create a GitRepository source manifest pointing to the application repository.
- Create a Kustomization manifest that deploys the application.

When the Kustomization manifest is pushed to the cluster's Git repository, the kustomize-controller reconciles the desired state stored in the Kustomization manifest in the application repository with the actual state on the Kubernetes cluster, on the schedule defined in the YAML.

Bootstrapping Flux

To get Flux up and running on the Kubernetes cluster, we use the flux command line. This example uses a cluster called "services," and the current kubectl context is configured to point to this cluster. It uses GitHub to host the configuration, in the VMwareCMBUTMM org. It creates a repository called autotmm-flux and creates a folder called services within that repository.

```
export GITHUB_TOKEN="my github token"
flux bootstrap github --owner=VMwareCMBUTMM
  --repository=autotmm-flux --branch=main --path
  services
```

After the deployment completes, you can examine the `pods` and `deployments` created using the following command:

```
kubectl get deployments,pods -n flux-system
```

The created repository, autotmm-flux, will be the source of truth for deployments to the Kubernetes cluster. Any manifests added to the repository will automatically be synchronized and deployed to the Kubernetes cluster. Any changes made to the existing manifest will be synchronized to the cluster. Changes made directly on the Kubernetes cluster (for example, with `kubectl edit`) will be detected and reverted to the desired state in the manifests. If a manifest is removed from the repository, the objects described in the manifest will be deleted from the Kubernetes cluster.

Create a GitRepository source

A GitRepository manifest configures a "source" custom resource that points to a Git repository, and it specifies which branch to target and the interval at which it should check for changes.

This example uses the podinfo repository, which is a public repository with a simple web application that includes a Kustomization manifest in the source code. The flux command line can be used to create a manifest for the Source:

```
flux create source git podinfo \
  --url=https://github.com/stefanprodan/podinfo \
  --branch=master \
  --interval=30s \
  --export > services/podinfo-source.yaml
```

The generated YAML manifest is then pushed up to the cluster's flux repository with standard git commands:

```
git add . && git commit -m "Add podinfo
   GitRepository" && git push
```

Create a Kustomization manifest

Now that we have a GitRepository "source" configured, we can use that source to deploy a Kustomize manifest. This manifest will tell the Kustomize controller the source (podinfo), the path to the Kustomize files within the repository (./kustomize) and the interval at which the controller should check for changes to the configuration.

The manifest can again be generated from the flux command-line interface (CLI):

```
flux create kustomization podinfo \
  --source=podinfo \
  --path="./kustomize" \
  --prune=true \
  --validation=client \
  --interval=5m \
  --export > services/podinfo-kustomization.yaml
```

Again, after the generated manifest is pushed up to the cluster's flux repository, the controllers will synchronize and deploy the manifest:

```
git add . && git commit -m "Add podinfo
  Kustomization" && git push
```

We can monitor the progress of the deployments using:

```
flux get kustomizations –watch
```

After the Kustomizations have synchronized, we can view the deployments and pods using the following command:

```
kubectl get deployments,pods –n default
```

If we edit the Kustomize manifest and set the `minReplicas: 3`, and then push the manifest changes up to the repository, Flux will ensure the change is reflected on the Kubernetes cluster.

GitOps in vRealize Automation

vRealize Automation has its own pipeline solution called VMware Code Stream. Code Stream can also be used to deploy and manage a Kubernetes application in a similar manner as Flux, using the native Git and Kubernetes endpoints. The following example uses the same podInfo Kubernetes manifest as the previous example. The YAML provides the IaC definition.

Code Stream can use placeholder variables that are populated when the Pipeline runs (the IaC variables), which, combined with the YAML definition, describe the desired state.

Create a Git endpoint

In Figure 4-2, the single source of truth is a Git endpoint that provides credentials and configuration for our Git repository. We've forked the original pod info repository so that we can configure webhooks and created a Personal Access Token in the GitHub account to authenticate with the GitHub application programming interface (API). This example also uses a Code Stream Variable to securely hold our token.

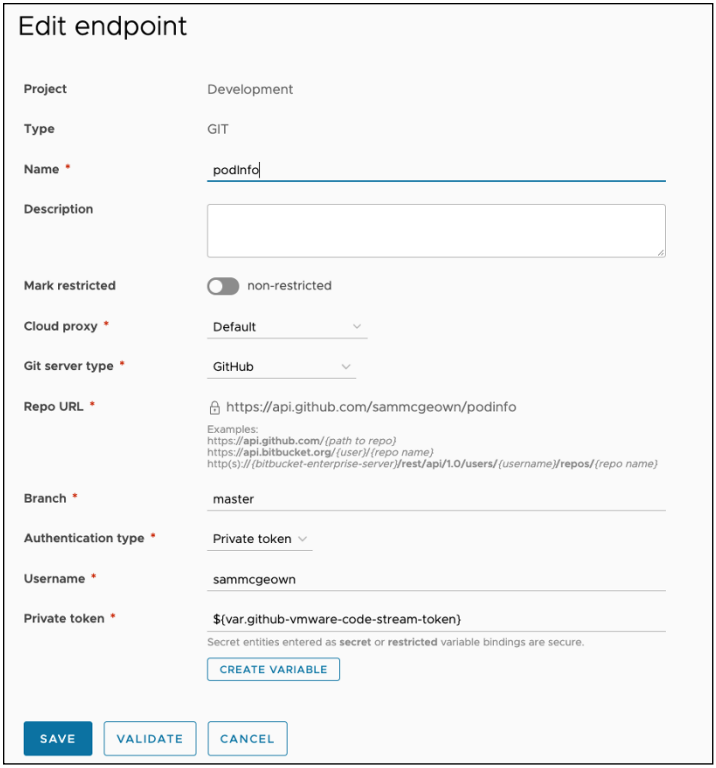

FIGURE 4-2: Adding a Git endpoint in Code Stream.

Create a Kubernetes endpoint

To perform tasks on a Kubernetes cluster, we first need to add the cluster as an endpoint. This is reasonably straightforward, because the values will match the contents of your clusters kube-conf file. Figure 4-3 uses certificate authentication, but you can use a token or username and password instead.

Configure a deployment pipeline

Next, we create a new deployment pipeline that is configured to be triggered by a Git webhook. Input variables are configured automatically by selecting the Git option on the Input tab, shown in Figure 4-4.

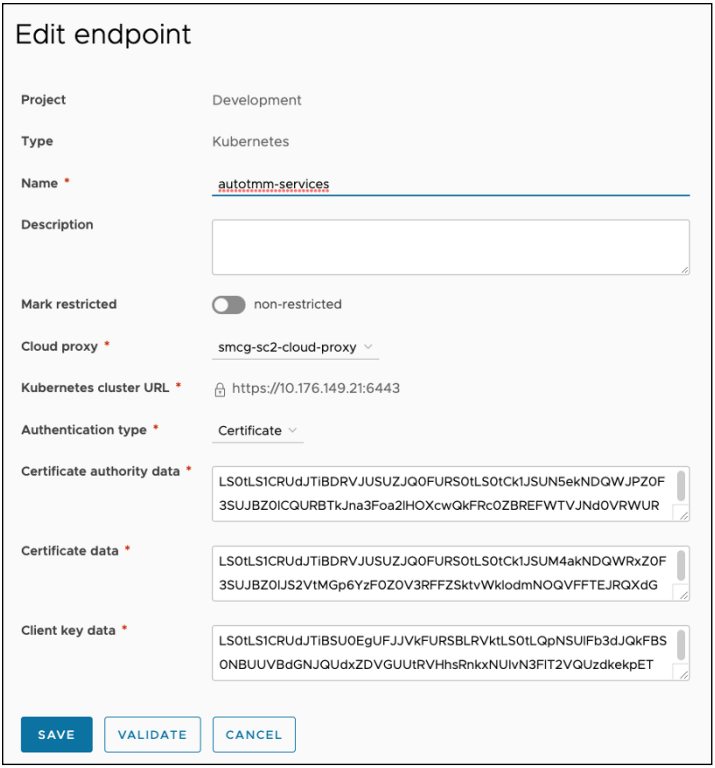

FIGURE 4-3: Adding a Kubernetes endpoint in Code Stream.

We can then add stages and tasks to build a deployment pipeline for the application. In Figure 4-5, we're using Kubernetes tasks to deploy the YAML manifest directly from the Git endpoint we created earlier. The Kubernetes tasks can be used to create, apply, or delete manifests from the Kubernetes endpoint.

With the Kubernetes tasks configured, we can run the pipeline to deploy the podInfo application, as shown in Figure 4-6.

You can view the deployed pods, service, deployment, replicaset, and autoscaler using the following command:

```
kubectl get all
```

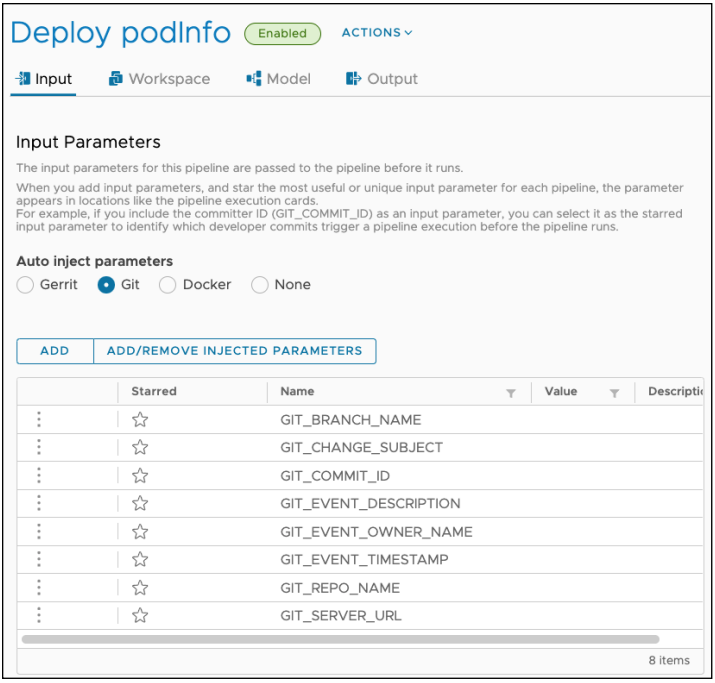

FIGURE 4-4: Configuring a pipeline for Git triggers with Git inputs.

Configure a Git trigger

With the Git endpoint and pipeline in place, you can configure a Git webhook to trigger the pipeline when a push or pull request is made to the repository, as shown in Figure 4-7.

When new code is pushed to the Git repository, a new pipeline execution should be triggered and visible under the Git Activity page. Currently the autoscaler is set to a minimum of two replicas, so we'll update that to three, which should also update the number of deployed pods.

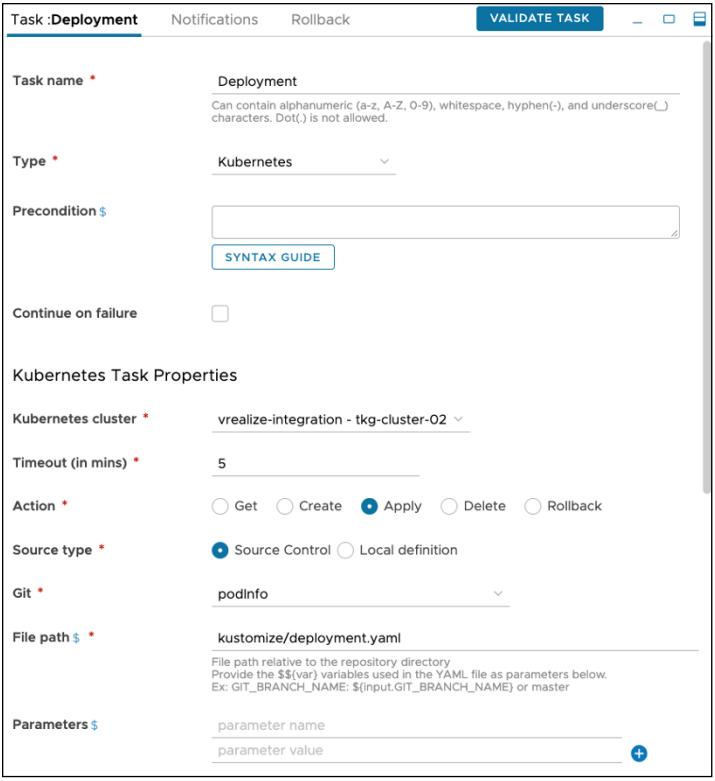

FIGURE 4-5: Configuring a Kubernetes task.

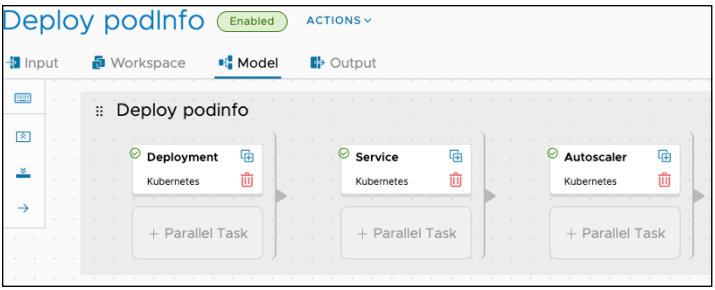

FIGURE 4-6: Kubernetes tasks in a deployment pipeline.

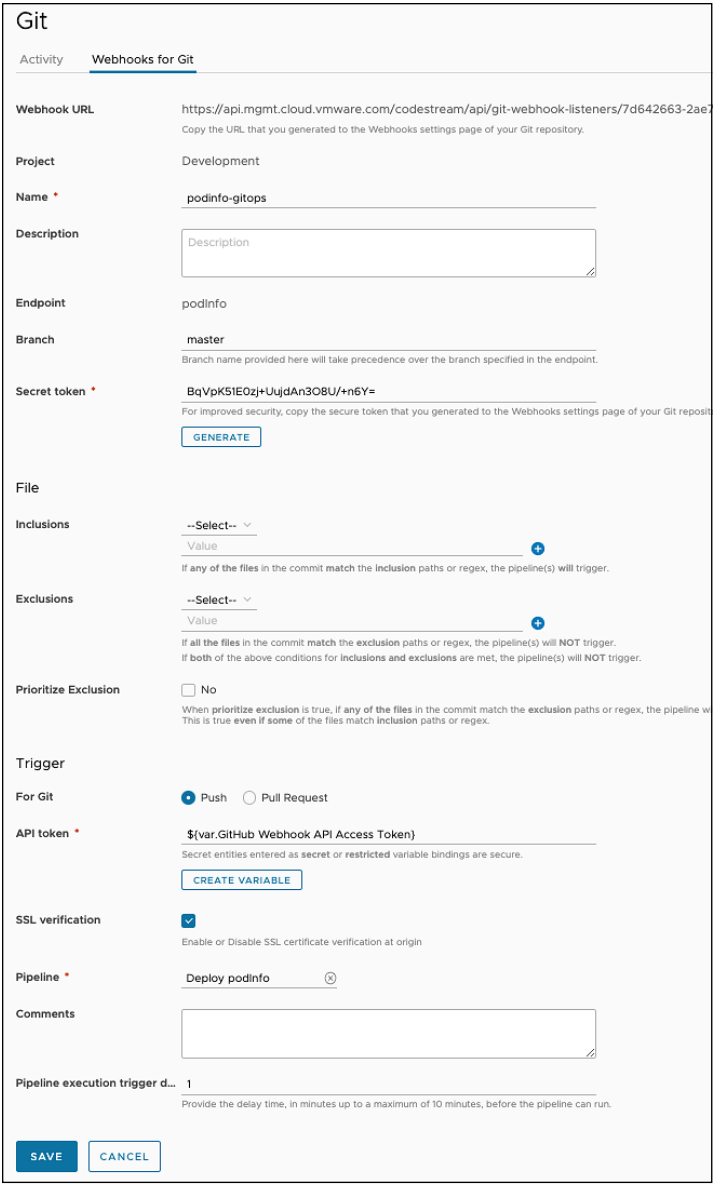

FIGURE 4-7: Configuring a Git webhook.

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: podinfo
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  minReplicas: 3
  maxReplicas: 4
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        # scale up if usage is above
        # 99% of the requested CPU (100m)
        averageUtilization: 99

git add . && git commit -m "Increase autoscaler
   minReplicas" && git push
[master 1b06052] Increase autoscaler minReplicas
 1 file changed, 1 insertion(+), 1 deletion(-)
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 16 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 371 bytes | 371.00
   KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), completed
   with 3 local objects.
To github.com:sammcgeown/podinfo.git
   627d5c4..1b06052  master -> master
```

The pipeline will be triggered, as shown in Figure 4-8.

The number of pods will be updated to the new setting:

```
kubectl get all
```

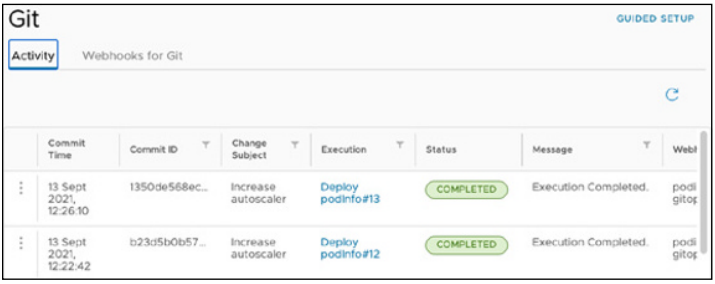

FIGURE 4-8: Git Activity showing a triggered webhook.

IN THIS CHAPTER

- Gathering support for your project
- Planning for a successful roll-out
- Learning from your mistakes

Chapter 5
Ten Tips for DevOps for Infrastructure

There is no step-by-step recipe to succeed in implementing DevOps, or DevOps for Infrastructure, but there certainly are many ways you can fail. In this last chapter, we've collected ten tips that will help you avoid common pitfalls.

Find Your Executive Sponsor

Projects succeed when there's someone near the top of the organization who wants them to succeed and clears the obstacles for that to happen. This is true for almost any project, especially in larger and more hierarchical organizations, but it's even more important for a project that will require significant time, resources, and patience without an easily quantifiable return on investment (ROI).

Find Your Team of Champions

Being the hero is awesome, but DevOps is a team matter. Do the groundwork to get enough buy-in from engineers and other parties; you'll need their precious time and collaboration.

Roll Out Gradually

Be bold but kind. Follow a phased approach with small, tangible goals and improve *continuously.*

Be Pragmatic and Flexible

DevOps requires both cultural change and process change, and those kinds of change take time. Remind yourself and your team that establishing habits requires persistence, iteration, and sometimes pivoting.

Observe Everything

Openness is critical to foster a collaborative environment. Collaboration is a key DevOps pillar, and enabling anyone to access information across the process will help establish trust, resolve issues faster, and focus efforts on the right areas. Meanwhile, be intentional about metrics that you report and measure success by. Velocity (for example, deployment frequency) should not be the only target outcome. In infrastructure use cases, time to recovery, resilience, and change failure rate are more important metrics.

Automate Everything

Automate it all! Or at least what's worth automating. Reducing labor will provide some great early wins and boost morale because it's *so* satisfying!

Curate a Tool List

DevOps is not about the tooling, but the right solutions for specific areas can help catalyze adoption. It's important to not be overly opinionated. After all, DevOps is very much about enabling choice. However, for critical functions that cut across teams, select tools that bias for extensibility and provide accessible interfaces and common languages. Provide a process and guidelines for adding new tools, and be intentionally inclusive.

Set Common Goals and Separate Concerns

Setting common goals, tracking them, and documenting them through service-level objectives (SLOs) and service-level agreements (SLAs) are the keys to a healthy DevOps culture.

REMEMBER

Collaboration does not mean everyone does everything. Be explicit about team ownership, and assign responsibility to the subject matter experts (SMEs).

WARNING

Avoid assigning blame to individuals, and focus on what went wrong rather than who made a mistake. DevOps processes should be resilient to individual mistakes. And, besides, blaming the system is always more fun!

Embrace Your Failures

Fail fast, and bounce back faster. DevOps won't prevent you from failing, but it will provide pillows to make the fall softer and the tools to recover more quickly — and in some cases even instantly. Failing is the best way to learn, so don't be afraid of it, but make sure to seize the opportunity to improve.

Don’t Stop Learning

DevOps is about community. Share your questions, your successes, and your failures, either online (like on Reddit) or in person (like with DevOps Days). Learning is one of the most important and enjoyable parts of the process!

Make DevOps work for you

DevOps is a set of principles designed to bridge the gap between application development and IT operations. Breaking down the Dev and Ops silos can shorten delivery times, reduce risks, encourage creativity, and increase customer satisfaction. Sounds good, right? But how to get that ball rolling isn't always obvious. This handy book offers a concise, fun-to-read overview of DevOps from the infrastructure point of view, explaining how DevOps came to be, what it offers modern organizations, and what steps to take when you're ready to get started.

Inside...

- Find out how DevOps came to be
- Discover the key principles of DevOps
- Understand how infrastructure as code works
- Get familiar with infrastructure pipelines
- Understand the GitOps process
- Get tips and tricks that ensure a successful rollout

vmware®

Sam McGeown is a technical architect working for VMware and based in the UK, specialising in cloud automation and modern applications. **Lefteris Marakas** is a product marketer working for VMware and aspiring to make technology more accessible and inclusive. He strives to simplify concepts and trends around digital transformation, platforms, and organizations, focusing on areas such as DevOps and cloud automation.

ISBN: 978-1-119-83780-0
Not For Resale

for dummies®
A Wiley Brand

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.