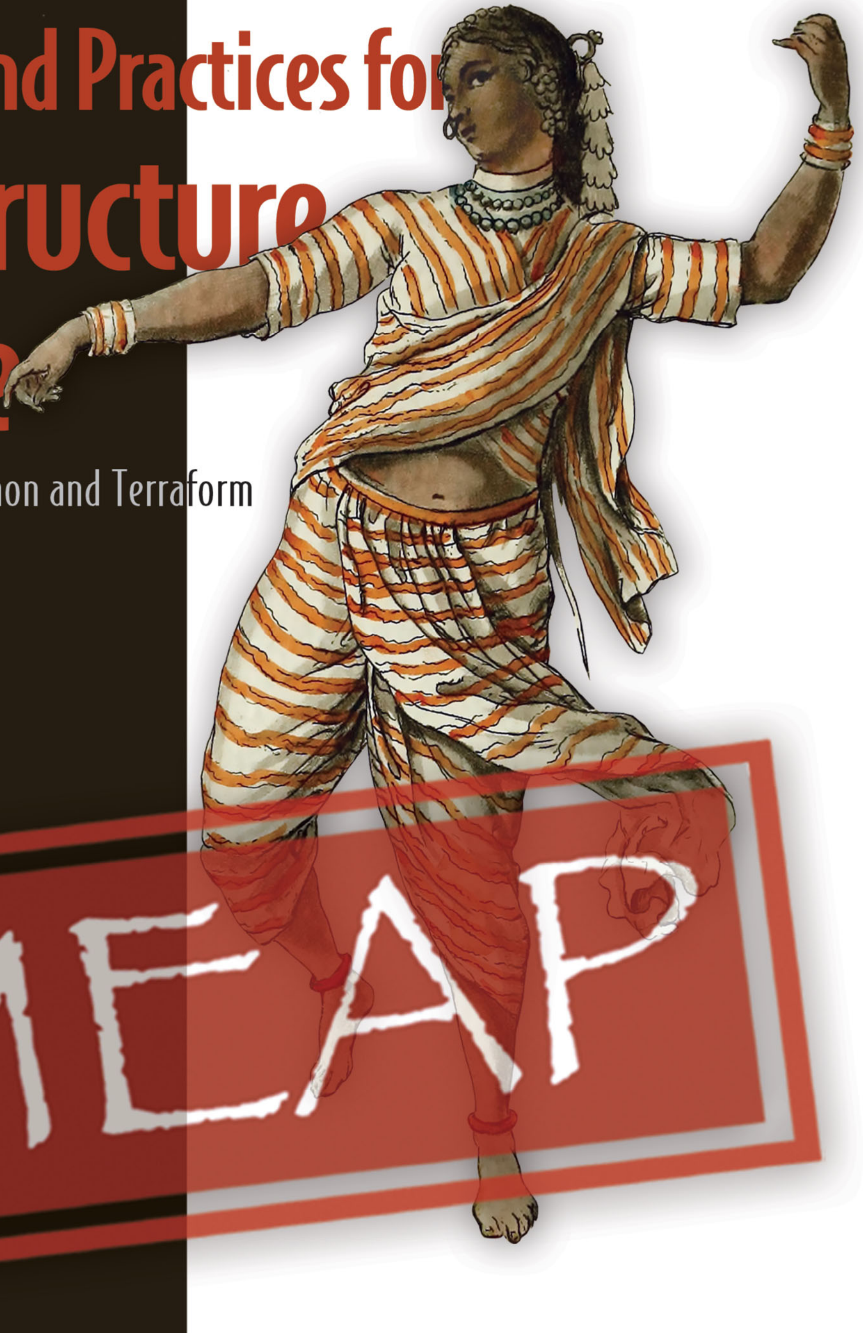
Patterns and Practices for Infrastructure as Code

With examples in Python and Terraform

Rosemary Wang



MANNING

MEAP Edition
Manning Early Access Program

Patterns and Practices for Infrastructure as Code

With examples in Python and Terraform

Version 8

welcome

Thank you for purchasing the MEAP of *Patterns and Practices for Infrastructure as Code*. Whether you're an operations, systems, DevOps, or software engineer, I hope you'll take some of the patterns and practices in this book and immediately apply them to your infrastructure configuration.

To make the most of the book, you'll want to have foundational concepts of infrastructure or learned at least one public cloud (such as Google Cloud Platform, Amazon Web Services, or Microsoft Azure). It also helps if you have some knowledge of a scripting or programming language (like Python) and got started with an infrastructure as code tool (like HashiCorp Terraform). I tried to generalize the concepts but include some hands-on examples showing implementation. You can apply the patterns to your own infrastructure, whether datacenter or cloud.

I divided the book into three parts. The first part covers how to write clean infrastructure as code. It contains a lot of patterns for modularizing and organizing your configurations. The patterns might seem a bit abstract at first but set the foundations to reduce maintenance and promote change in infrastructure systems.

The second part discuss how your team uses infrastructure as code. You'll learn team practices to scale collaboration. These involve testing, branching models, continuous delivery, and security auditing.

I'll conclude with a section on managing production complexity with infrastructure as code. You and your company can use some techniques like feature flagging and blue-green deployments of infrastructure to reduce the blast radius of failed changes. Hopefully, they'll help you continuously update infrastructure and minimize disruption.

Please leave your feedback or questions in the liveBook Discussion forum. Infrastructure as code is an art, after all! The explanations and topics can always be improved. I appreciate your comments and suggestions.

—Rosemary Wang

brief contents

1 Introducing infrastructure as code

This chapter covers

- **Defining infrastructure**
- **Defining infrastructure as code**
- **Understanding why infrastructure as code is important**

If you just started working with public cloud providers or data center infrastructure, you might feel overwhelmed with all that you have to learn. You don't know *what* you need to know to do your job! Between data center infrastructure concepts, new public cloud offerings, container orchestrators, programming languages, and software development, you have a lot to research.

Besides learning everything you can, you also have to keep up with your company's requirements to innovate and grow. Building systems to support it all gets challenging. You need a way to support more complex systems, minimize your maintenance effort, and avoid disruption to customers using your application.

What do you need to know to work with cloud computing or data center infrastructure? How can you scale your systems across a team and your organization? The answer to both questions involves infrastructure as code (IaC), the process of automating infrastructure changes in a codified manner to achieve scalability, reliability, and security.

Everyone can use infrastructure as code, from system administrators, site reliability engineers, DevOps engineers, security engineers, software developers, to quality assurance engineers. Whether you've just run your first tutorial for infrastructure as code or passed a public cloud certification (congratulations!), you can apply infrastructure as code to larger systems and teams to simplify, sustain, and scale your infrastructure.

This book offers a practical approach to infrastructure as code by applying software development practices and patterns to infrastructure management. It takes practices like testing, continuous delivery, refactoring, and design patterns with an infrastructure twist! You'll find practices and patterns to help you manage your infrastructure no matter the automation, tool, platform, or technology.

I divided this book into three parts (figure 1.1). Part one covers the practices you can apply to write infrastructure as code, while Part two describes your team's patterns and practices to collaborate on it. Part three covers some approaches to scaling infrastructure as code across your organization.

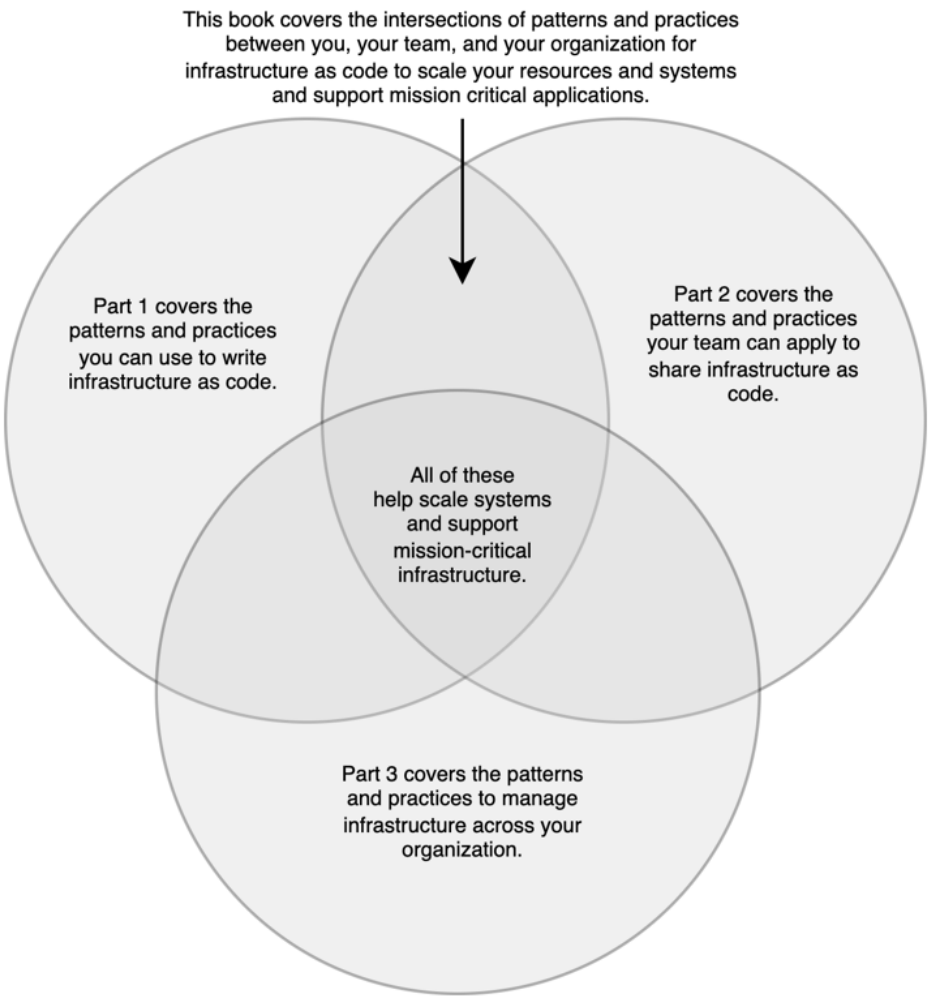

This book covers the intersections of patterns and practices between you, your team, and your organization for infrastructure as code to scale your resources and systems and support mission critical applications.
Part 1 covers the patterns and practices you can use to write infrastructure as code.
Part 2 covers the patterns and practices your team can apply to share infrastructure as code.
All of these help scale systems and support mission-critical infrastructure.
Part 3 covers the patterns and practices to manage infrastructure across your organization.

Figure 1.1. In this book, you will learn the intersections of practices between you, your team, and your organization and how they help you scale systems and support mission-critical applications.

Many of the patterns and practices in this book intersect between these interests. Writing good infrastructure as code individually helps you better share and scale it across your team and organization! Well-written infrastructure as code helps solve problems with *collaborating* on infrastructure as code, especially as more people adopt it.

Part one starts by defining infrastructure and explaining common infrastructure as code design patterns. These topics involve foundational concepts that help you scale IaC across your team. You may already be familiar with some of the material in this part, so review these chapters to establish foundations for more advanced concepts.

In parts two and three, you'll learn the patterns and practices you need to scale systems and support infrastructure for mission-critical applications. These practices extend from you to your team and organization, from creating one application metrics alert for an application to implementing a network change across a fifty thousand person organization. Many terms and concepts build on each other in these parts so you might find it helpful to read the chapters in order.

1.1 What is infrastructure?

Before I dive into infrastructure as code, let's begin with the definition of infrastructure. When I began working in a data center, the literature often defined infrastructure as hardware or devices that provide network, storage, or compute capability. Figure 1.2 shows how applications run on servers (compute), connect through a switch (network), and maintain data on disks (storage).

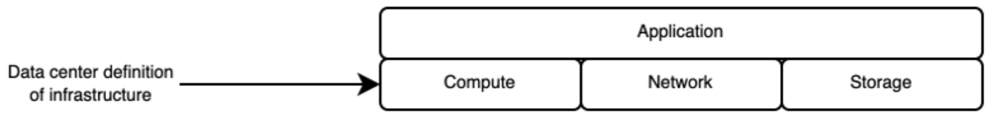


Figure 1.2. The data center definition of infrastructure includes networking, compute, or storage resources used to run an application.

These three categories matched the physical devices we managed in a data center. We made changes by scanning into a building, plugging into a device, typing commands, and hoping that everything still works. With the advent of cloud computing, we continued to use these categories to discuss virtualization of specific devices.

However, the data center definition of infrastructure doesn't quite apply to today's services and offerings. Imagine that another team requests that you help them deliver their application to production for users. You run through a checklist that includes setting up:

- Enough servers
- Network connectivity for users
- A database for storing application data

Does completing this checklist mean the team can run this application in production? Not necessarily! You do not know if you set up enough servers or the proper access to log into the application. You also need to know if network latency affects the application's database connection.

In this narrow definition of infrastructure, you omit some critical tasks necessary for production readiness, including the following:

- Monitoring application metrics
- Exporting metrics for business reporting
- Setting up alerts for teams operating the application
- Adding health checks for servers and databases
- Supporting user authentication
- Logging and aggregating application events
- Storing and rotating database passwords in a secrets manager

You need these "to-do" items to deliver an application to production that will work reliably and securely. You might think of them as "operational" requirements, but they still require infrastructure resources.

Besides infrastructure related to operations, public cloud providers abstract the management of base networking, compute, and storage and offer Platform as a Service (PaaS) offerings instead, from object stores like storage buckets to event streaming platforms like managed Apache Kafka. Providers even offer Function as a Service or Containers as a Service, additional abstractions for computing resources. The increasing marketplace of Software as a Service (SaaS) such as hosted application performance monitoring software might also be required to support an application in production and could also be infrastructure!

With so many services, you cannot describe infrastructure with just compute, network, and storage categories. We need to include operational infrastructure, PaaS, or SaaS offerings in our application delivery. Figure 1.3 adjusts the model of infrastructure to include additional service offerings like SaaS and PaaS that help us deliver applications.

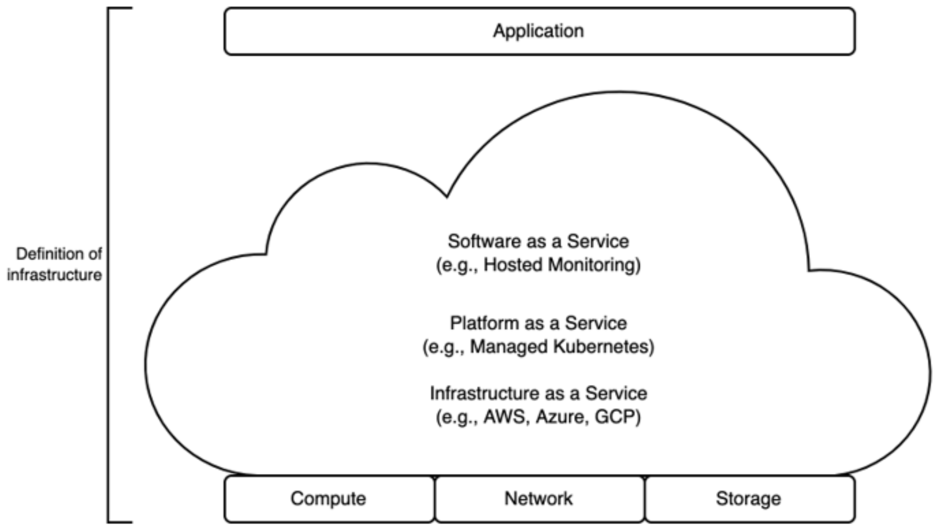

Application
Definition of infrastructure
Software as a Service (e.g., Hosted Monitoring)
Platform as a Service (e.g., Managed Kubernetes)
Infrastructure as a Service (e.g., AWS, Azure, GCP)
Compute
Network
Storage

Figure 1.3. The infrastructure for an application might even include queues on a public cloud, containers running applications, serverless functions for additional processing, or even monitoring services to check the system's health.

Due to the growing complexity, varying operating models, and user abstraction of data center management, you can't limit the definition of ***infrastructure*** to hardware or physical devices related to compute, network, or storage.

> **DEFINITION** **Infrastructure refers to software, platform, or hardware that delivers or deploys applications to production.**

A non-exhaustive list of infrastructure you might encounter includes:

- Servers
- Workload orchestration platforms (e.g., Kubernetes, HashiCorp Nomad)
- Network switches
- Load balancers
- Databases
- Object stores
- Caches
- Queues
- Event streaming platforms
- Monitoring platforms
- Data pipeline systems
- Payment platforms

Expanding the infrastructure definition provides a common language across teams managing resources for various purposes. For example, a team managing an organization's continuous integration (CI) framework utilizes infrastructure from either a continuous integration SaaS or compute resources from a public cloud. Another team builds upon the framework, thus making it critical infrastructure.

1.2 What is infrastructure as code?

Before explaining infrastructure as code, we must understand the manual infrastructure configuration. In this section, I outline the problem with infrastructure and manual configuration. Then, I will define infrastructure as code.

1.2.1 Manual configuration of infrastructure

As part of a network team, I learned to change a network switch by copying and pasting commands from a text document. I once pasted "shutdown" instead of "no shutdown," turning off a network interface! I quickly turned it back on, hoping that no one noticed and it didn't affect anything. A week later, however, I discovered that it shut off connectivity to a critical application and affected a few customer requests.

In retrospect, I ran into a few problems with my manual copy and paste of commands and infrastructure configuration. First, I had no idea what resources my change would affect (also known as the ***blast radius***). I did not know which networks or applications used the interface.

DEFINITION **The blast radius refers to the impact a failed change has on a system. A larger blast radius often affects more components or the most critical components.**

Second, the network switch accepted my command without testing its effects or checking its intent. Finally, no one else knew what affected the application's processing of customer requests, and it took a week for them to identify the root cause as my miscopied command.

How does writing infrastructure in a codified manner help catch my miscopied network switch command? I could store my configuration and automation under source control to record the commands. To catch my mistake going forward, I create a virtual switch and a test that runs my script and checks the health of the interface.

After the tests pass, I promote the change to production because the tests check for the correct command. If I apply the wrong command, I can search infrastructure configuration to determine which applications run on the affected network. You can refer to Chapter 6 for testing practices and Chapter 11 for reverting changes.

Besides the risk of misconfiguration, my development momentum sometimes slows because of manual infrastructure configuration. Once, it took nearly two months for me to test my application against a database. Throughout those two months, my team submitted over ten tickets related to creating the database, configuring new routing to connect it to the database, and opening firewall rules to allow my application. The platform team manually configured everything in a public cloud. Development teams did not have direct access due to security concerns.

In other words, manual configuration of infrastructure often does not scale as systems and teams grow. Manual changes *increase the change failure rate of systems, slow down development, and expose the system's attack surface to a potential security exploit*. You will always have the temptation to update some values into a console. However, these changes accumulate.

The next person who makes a change to the system may introduce a failure into the system that will be difficult to troubleshoot because changes haven't been audited or organized. Changes like updating a firewall to allow some traffic during the development process can inadvertently leave the system vulnerable to attack.

1.2.2 Infrastructure as code

What should you do to change infrastructure, if not manual changes? You can apply a software development lifecycle for infrastructure resources and configuration in the form of infrastructure as code. However, an infrastructure development lifecycle goes beyond configuration files and scripts.

Infrastructure needs to scale, manage failure, support rapid software development, and secure applications. A development lifecycle for infrastructure involves more specific patterns and practices to support collaboration, deployment, and testing. In figure 1.4, a simplified workflow changes infrastructure using configuration or scripts and committing them to version control. A commit automatically starts a workflow to deploy and test the changes to your infrastructure.

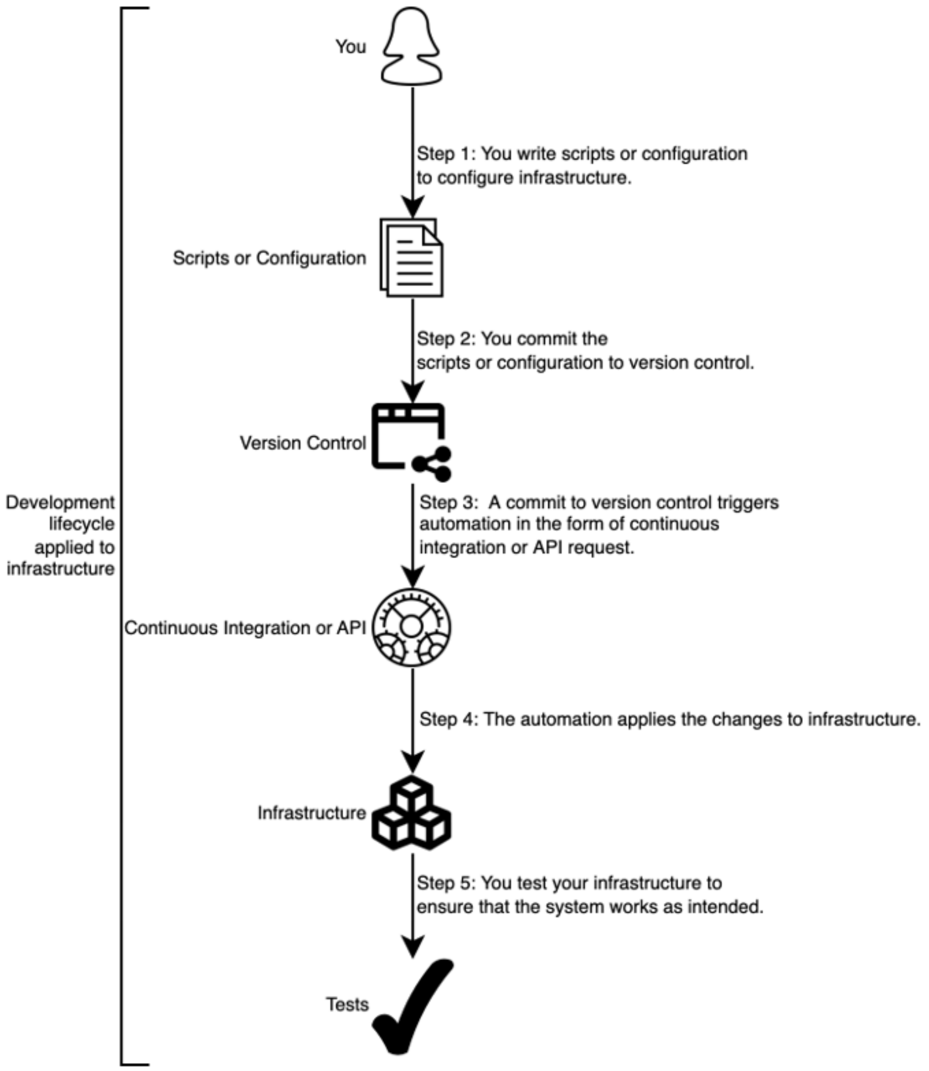

Development lifecycle applied to infrastructure
You
Step 1: You write scripts or configuration to configure infrastructure.
Scripts or Configuration
Step 2: You commit the scripts or configuration to version control.
Version Control
Step 3: A commit to version control triggers automation in the form of continuous integration or API request.
Continuous Integration or API
Step 4: The automation applies the changes to infrastructure.
Infrastructure
Step 5: You test your infrastructure to ensure that the system works as intended.
Tests

Figure 1.4. A development lifecycle for infrastructure includes writing the code as documentation, committing it to version control, applying it to infrastructure in an automated way, and testing it.

Why should you remember the development lifecycle? You can use it as a general pattern for managing changes and verifying they don’t affect your system! The lifecycle captures ***infrastructure as code***, which automates infrastructure changes in a codified manner and applies DevOps practices such as version control and continuous delivery.

> **DEFINITION** Infrastructure as code (IaC) applies DevOps practices to automating infrastructure changes in a codified manner to achieve scalability, resiliency, and security.

I often find infrastructure as code cited as a necessary practice of DevOps. It certainly addresses the automation piece of the CAMS model (Culture, Automation, Measurement, and Sharing). Figure 1.5 positions infrastructure as code as part of automation practices and philosophy in the DevOps model. The practices of code as documentation, version control, software development patterns, and continuous delivery align with the development lifecycle workflow we discussed previously!

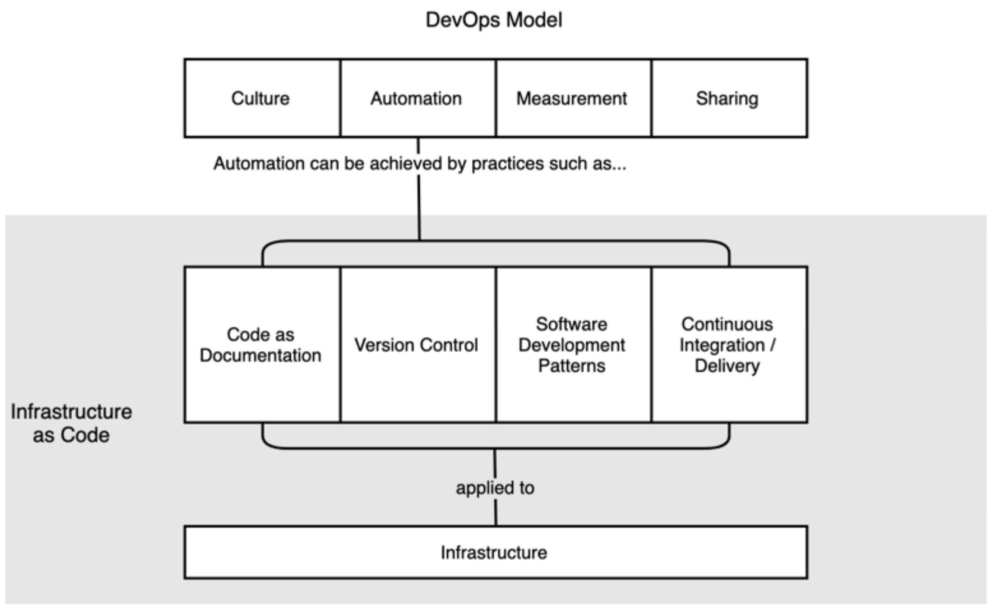


Figure 1.5. Infrastructure as code applies version control, software development patterns, continuous integration, and code as documentation to infrastructure.

Why focus on infrastructure as code as part of automation in the DevOps model? Your organization does not have to adopt DevOps to use infrastructure as code. Its benefits improve DevOps adoption and metrics but still apply to any infrastructure configuration. You can still use infrastructure as code practices to improve the process of making infrastructure changes without affecting production.

More on DevOps

You will see some DevOps practices included in this book, but I will not be focusing on its theory or principles. I recommend Accelerate by Nicole Forsgren, Ph.D., Gene Kim, and Jez Humble for a higher-level understanding of DevOps. You can also peruse Gene Kim's The Phoenix Project, which describes the fictional transformation of an organization adopting DevOps.

This book will cover some approaches for codifying infrastructure to eliminate the friction of scale while maintaining the reliability and security of infrastructure for application users, whether you use the data center or cloud. Software development practices such as version control of configuration files, continuous integration pipelines, and testing can help scale and evolve changes to infrastructure while mitigating downtime and building secure configuration.

1.2.3 What is not infrastructure as code?

Could you be doing infrastructure as code if you type out some configuration in a document? You might consider infrastructure as code to include adding configuration instructions in a change ticket. You can argue that a tutorial for building a queue or a shell script for configuring a server counts as infrastructure as code. Each of these examples *can* be forms of infrastructure as code if you can use them to:

- Reliably and accurately reproduce the infrastructure it expresses
- Revert configuration to a specific version or point in time
- Communicate and assess the blast radius of a change to the resources.

The configurations or scripts, however, are usually outdated, unversioned, or ambiguous in intent. You may even find yourself struggling to understand and change the configuration written in an IaC tool! The tool facilitates IaC workflows but not necessarily the practices and approaches that allow systems to grow while reducing operational responsibility and change failure. You need a set of principles to identify infrastructure as code.

1.3 Principles of infrastructure as code

As I mentioned, not every piece of code or configuration related to infrastructure will scale or mitigate downtime. Throughout the book, I will highlight how IaC principles apply to certain code listings or practices. You can even use these principles to assess your infrastructure as code.

While others may add or subtract to this list of principles, I remember four of the most important principles with the mnemonic, *RICE*. RICE stands for reproducibility, idempotency, composability, and evolvability. I will define and apply each principle in the following sections.

1.3.1 Reproducibility

Imagine if someone asks you to create a development environment with a queue and a server. You share a set of configuration files for your teammate. They use them to recreate a new environment for themselves in less than an hour. Figure 1.6 shows how you shared your configuration and enabled your teammate to *reproduce* a new environment. You discovered the power of reproducibility, the first principle of infrastructure as code!

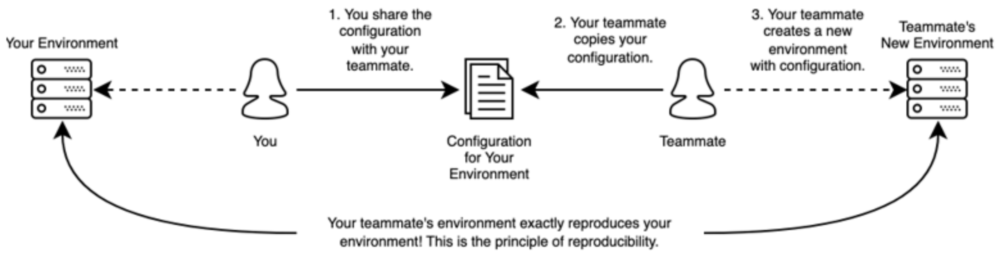

Your Environment
1. You share the configuration with your teammate.
2. Your teammate copies your configuration.
3. Your teammate creates a new environment with configuration.
Teammate's New Environment
You
Configuration for Your Environment
Teammate
Your teammate's environment exactly reproduces your environment! This is the principle of reproducibility.

Figure 1.6. Manual changes introduce drift between version control and actual state and affects reproducibility, so you instead update changes in version control.

Why should infrastructure as code conform to the principle of ***reproducibility***? The ability to copy and reuse infrastructure configuration saves you and your team's initial engineering time. You don't have to "reinvent the wheel" to create new environments or infrastructure resources!

> **DEFINITION** The principle of reproducibility means that you can use the same configuration to reproduce an environment or infrastructure resources.

However, you'll find adhering to the principle of reproducibility more complicated than copying and pasting a configuration. To demonstrate this nuance, imagine you need to reduce a network address space from "/26" to "/24." You do have infrastructure as code written that expresses the network. However, you decide to choose the easy route of logging into the cloud provider and typing "/24" into the text box.

Before you log into the cloud provider, you reflect on whether or not your change workflow adheres to reproducibility. You ask yourself the following questions:

- Will a teammate know that you've updated the network?
- If you run your configuration, will the network address space return to the "/16"?
- If you create a new environment with the configuration in version control, will it have an address space of "/24"?

You answer "no" to each of those questions! You cannot guarantee that you will reproduce the manual change successfully.

If you go ahead and type "/24" in the cloud provider's console, your network has *drifted* from its desired state expressed in IaC (figure 1.7). To conform to reproducibility, you decide to update the version control configuration to "/24" and apply the automation.

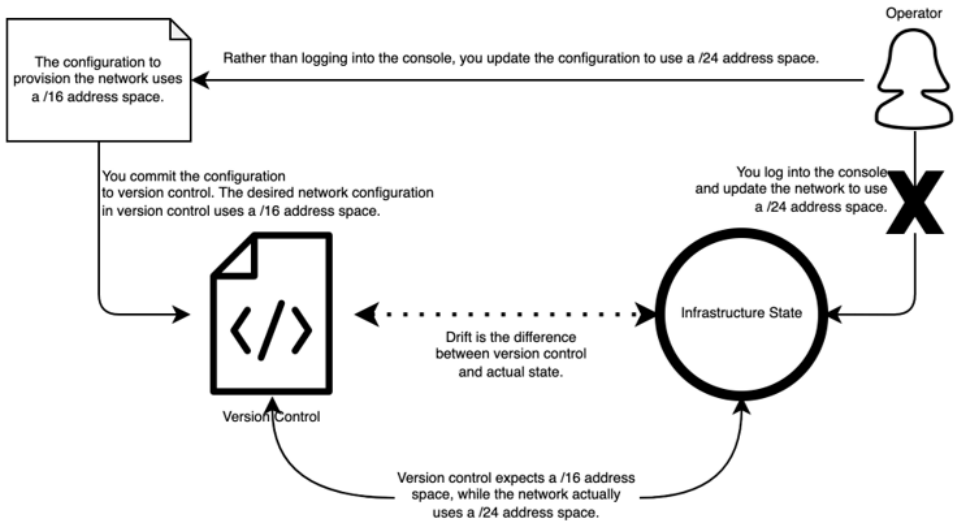

Operator
The configuration to provision the network uses a /16 address space.
Rather than logging into the console, you update the configuration to use a /24 address space.
You commit the configuration to version control. The desired network configuration in version control uses a /16 address space.
You log into the console and update the network to use a /24 address space.
X
Infrastructure State
Drift is the difference between version control and actual state.
Version Control
Version control expects a /16 address space, while the network actually uses a /24 address space.

Figure 1.7. Manual changes introduce drift between version control and actual state and affects reproducibility, so you instead update changes in version control.

This scenario demonstrates the challenge of conforming to reproducibility. You need to minimize the inconsistency between the expected and actual infrastructure configuration, also known as ***configuration drift***.

DEFINITION Configuration drift is the deviation of infrastructure configuration from the desired versus actual configuration.

As a practice, you can ensure the principle of reproducibility by placing your configuration files in version control and keeping version control as updated as possible. Maintaining the principle of reproducibility helps you collaborate better *and* manage testing environments similar to production. You'll learn more about infrastructure testing environments in Chapter 6, which benefit from reproducibility. You'll also apply reproducibility to practices and patterns in testing and upgrading infrastructure, from creating test infrastructure mirroring production to deploying new infrastructure to replace older systems (blue-green deployments).

1.3.2 Idempotency

Some infrastructure as code include *repeatability* as a principle, which means running the same automation and yielding consistent results. I pose that infrastructure as code needs a stricter requirement. Running automation should result in the same *end state* for an infrastructure resource. After all, I have one main objective when I write automation - the ability to run the automation multiple times and get the same result.

Let's understand why infrastructure as code needs a stricter requirement. Imagine you write a network script that first configures an interface and then reboots. The first time you run the script, the switch configures the interface and reboots. You save this script as version 1.

A few months later, your teammate asks you to run the script again on the switch. You run the script and the switch reboots. However, the reboot disconnected some critical applications! You already configured the network interface. Why do you need to reboot the switch?

You figure out a way to prevent the switch reboot if you already configured the network interface. In figure 1.8, you create version two of the script and add a conditional "if" statement. The statement checks if you have configured the interface already before rebooting the switch. When you run the version two script again, you do not disconnect the applications.

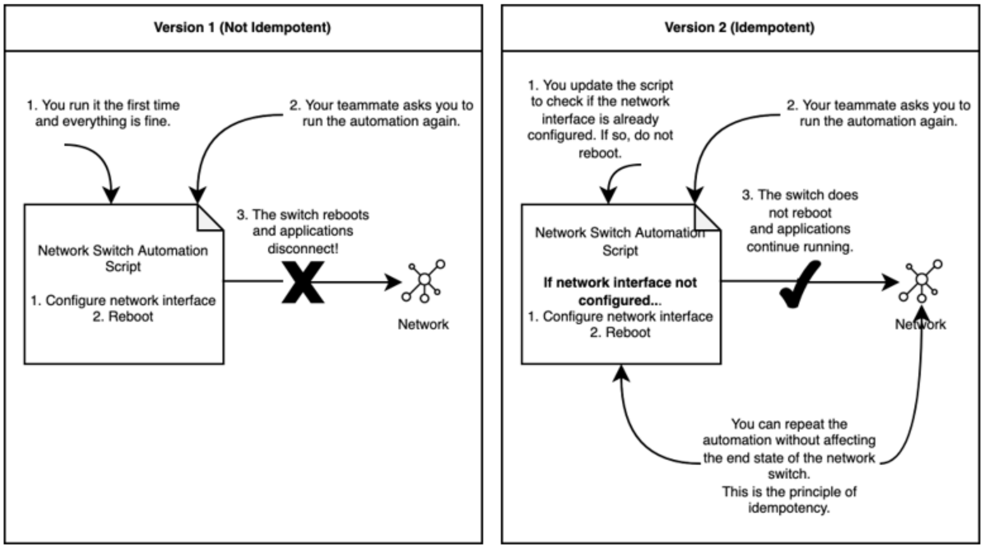

Version 1 (Not Idempotent)
1. You run it the first time and everything is fine.
2. Your teammate asks you to run the automation again.
Network Switch Automation Script
1. Configure network interface
2. Reboot
3. The switch reboots and applications disconnect!
Network
Version 2 (Idempotent)
1. You update the script to check if the network interface is already configured. If so, do not reboot.
2. Your teammate asks you to run the automation again.
Network Switch Automation Script
If network interface not configured...
1. Configure network interface
2. Reboot
3. The switch does not reboot and applications continue running.
Network
You can repeat the automation without affecting the end state of the network switch.
This is the principle of idempotency.

Figure 1.8. In version 1 of the script, you reboot the switch each time you run the script. In version 2 of the script, you check if the network interface is configured before rebooting the switch. This preserves the working state of the network.

The conditional statement conforms to the principle of idempotency. ***Idempotency*** ensures that you can repeat the automation without affecting the infrastructure unless you change the configuration or drift occurs. If an infrastructure configuration or script is idempotent, you can rerun the automation multiple times without affecting the state or operability of a resource.

> **DEFINITION** The principle of idempotency ensures that you can repeatedly run the automation on infrastructure without affecting its end state or having any side effects. You should only affect Infrastructure when you update its attributes in automation.

Why should you adhere to idempotency in your infrastructure as code, such as in the case of your network script? In the example, you want to avoid rebooting the network switch to keep the network running correctly. You already configured the network interface - why configure it again? You should only need to configure the interface if it doesn't exist or change.

Without idempotency, your automation might break accidentally. For example, you may repeat a script and create a new set of servers, doubling their numbers. More catastrophically, you might automate a database update only to remove a critical database!

You can ensure the principle of idempotency by checking the repeatability of scripts and configuration. As a general practice, include a number of *conditional statements* checking if a configuration matches the expected before running your automation. Conditional statements help apply changes when required and avoid side effects that may affect the operability of infrastructure.

Designing automation with idempotency reduces risk, as it encourages the inclusion of logic to preserve the final expected state of the system. If the automation fails once and causes an outage in the system, organizations no longer want to automate again because of its perceived risk. Idempotency becomes a guiding principle as you learn how to safely roll forward changes and preview automation changes before deployment in Chapter 11.

1.3.3 Composability

You want to mix and match any set of infrastructure components, no matter the tool or configuration. You also need to update the individual configurations without affecting the entire system. Both of these requirements promote modularity and decoupling infrastructure dependencies, something you'll learn more about in chapters 3, 4, and 5.

For example, imagine you create infrastructure for an application that someone accesses at "hello-world.com. The minimum resources you need for a secure and production-ready configuration include:

- A server
- A load balancer
- A private network for server
- A public network for a load balancer
- A routing rule to allow traffic out of the private network

- A routing rule to allow public traffic to the load balancer
- A routing rule to allow traffic from the load balancer to the server
- A domain name for "hello-world.com"

You could write this configuration from scratch. However, what if you found pre-constructed *modules* that group infrastructure components you can use to assemble the system? You now have different modules that create:

- Networks (private and public networks, gateways to route traffic from the private network, routing rules to allow traffic out of the private network)
- Servers
- Load balancers (domain name, routing rules to allow traffic from the load balancer to server)

In figure 1.9, you pick the network, server, and load balancer modules to build your production environment. Later, you realize you need a premium load balancer. You swap out the standard load balancer with a premium one so you can serve more traffic. The server and network continues running without affecting users.

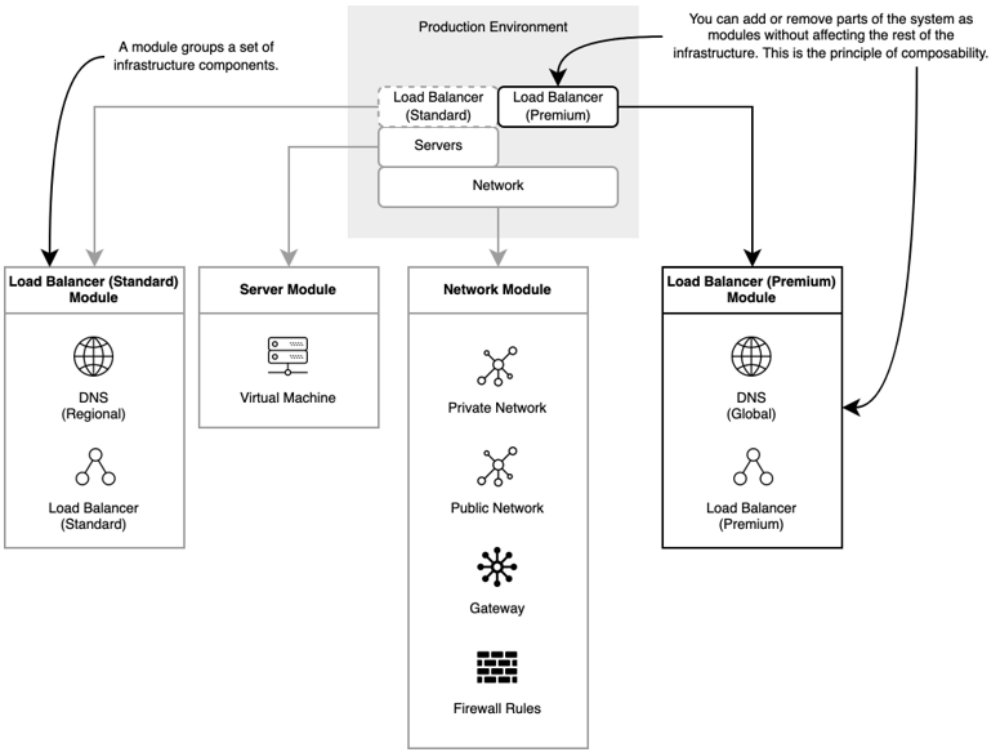

Production Environment
A module groups a set of infrastructure components.
You can add or remove parts of the system as modules without affecting the rest of the infrastructure. This is the principle of composability.
Load Balancer (Standard)
Load Balancer (Premium)
Servers
Network
Load Balancer (Standard) Module
DNS (Regional)
Load Balancer (Standard)
Server Module
Virtual Machine
Network Module
Private Network
Public Network
Gateway
Firewall Rules
Load Balancer (Premium) Module
DNS (Global)
Load Balancer (Premium)

Figure 1.9. You build your production environment with building blocks of infrastructure so you can add new resources, like the database, easily.

Your teammate can even add a database into the environment without affecting the load balancer, servers, or network. You group and select infrastructure resources in different combinations, which adheres to the principle of ***composability***.

> **DEFINITION** Composability ensures that you can assemble any combination of infrastructure resources and update each one without affecting the others.

The more composable your configuration, the easier it is to create new systems with less effort. Think of constructing your infrastructure as code with building blocks. You want to be able to update or evolve subsets of resources without toppling the entire system! If you do not consider the composability of your infrastructure as code, you run the risk of change failures due to unknown dependencies in complex infrastructure systems.

The self-service benefit of composability can help your organization scale and empower teams to interact safely with infrastructure systems. This book will examine some patterns in chapters 3, 4, and 5 that can help you approach more modular infrastructure construction and improve composability.

1.3.4 Evolvability

You want to account for the scale and growth of your system but not optimize the configuration too early and unnecessarily. Much of infrastructure configuration will change over time, including its architecture.

As a practical example, you might initially name an infrastructure resource "example." Later, you need to change the resource name to "production." You start the change by finding and replacing hundreds of tags, names, dependent resources, and more. The procedure of find-and-replace requires a high rate of effort.

You notice that you forgot to change some fields as you apply the changes and your new infrastructure changes fail. To ensure the future evolution of names, tags, and other metadata, you instead create a variable for the name, and the configuration references the variable. In figure 1.10, you update the global "NAME" variable, and the change propagates across the entire system.

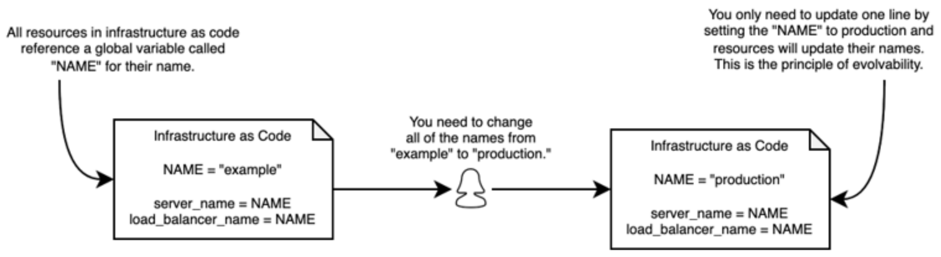

All resources in infrastructure as code reference a global variable called "NAME" for their name.
Infrastructure as Code
NAME = "example"
server_name = NAME
load_balancer_name = NAME
You need to change all of the names from "example" to "production."
Infrastructure as Code
NAME = "production"
server_name = NAME
load_balancer_name = NAME
You only need to update one line by setting the "NAME" to production and resources will update their names. This is the principle of evolvability.

Figure 1.10. Rather than finding and replacing all instances of the name, you can set a top-level variable with the name for all resources.

The example almost seems too simple. Why does changing a name matter? Infrastructure as code built with ***evolvability*** as a principle minimizes the effort (time and cost) to change a system and the risk of failure for the change.

DEFINITION **The principle of evolvability ensures that you can change your infrastructure resources to accommodate system scale or growth while minimizing effort and risk of failure.**

System evolution includes changes beyond minor ones, such as name changes. A more disruptive change in infrastructure architecture might involve a replacement of Google Cloud Platform's Bigtable with Amazon Web Service (AWS) Elastic Map Reduce. The application requiring the replacement has been future-proofed using HBase, an open-source distributed database that supports both offerings and simply requires the database endpoint.

We account for this evolution in the infrastructure as code by outputting the database endpoint to retrieve the application and completing the update behind-the-scenes by creating configurations for both offerings. After testing the AWS database, we output its endpoint for consumption by the application.

Evolvability, a complex discussion

I do not fully cover the theory in evolving your architecture in this book. I highly recommend Building Evolutionary Architectures by Neal Ford, Pat Kua, and Rebecca Parsons if you want to learn more. The book discusses how to build your infrastructure architecture to account for changes.

You might find yourself struggling to evolve your system because you have not used patterns and practices that allow it to change. Useful infrastructure as code focuses on techniques to facilitate future evolution. Many of the chapters in this book demonstrate patterns that help maintain evolvability and minimize the impact of changes to critical systems.

1.3.5 Applying the principles

Reproducibility, idempotency, composability, and evolvability seem specific in their definitions. However, they help constrain your infrastructure architecture and define the behavior of many IaC tools. Your infrastructure as code must align with all four principles to scale, collaborate, and change your company. Figure 1.11 summarizes the four most important principles (RICE) and their definitions.

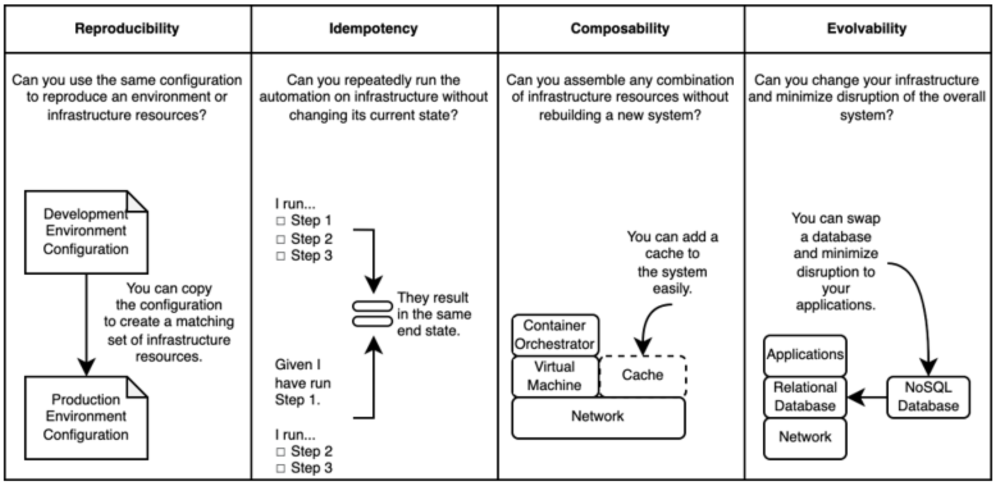

Reproducibility
Idempotency
Composability
Evolvability
Can you use the same configuration to reproduce an environment or infrastructure resources?
Can you repeatedly run the automation on infrastructure without changing its current state?
Can you assemble any combination of infrastructure resources without rebuilding a new system?
Can you change your infrastructure and minimize disruption of the overall system?
Development Environment Configuration
You can copy the configuration to create a matching set of infrastructure resources.
Production Environment Configuration
I run...
□ Step 1
□ Step 2
□ Step 3
They result in the same end state.
Given I have run Step 1.
I run...
□ Step 2
□ Step 3
You can add a cache to the system easily.
Container Orchestrator
Virtual Machine
Cache
Network
You can swap a database and minimize disruption to your applications.
Applications
Relational Database
NoSQL Database
Network

Figure 1.11. Infrastructure as code should be reproducible, idempotent, composable, and evolvable. You can ask yourself a series of questions to determine if your infrastructure as code conforms to all four principles!

As you write infrastructure as code, ask if you conform to all four principles. The principles help you write and share your infrastructure as code with less effort and ideally minimize the impact of changes to your system. A missing principle can hinder updates to infrastructure resources or increase the blast radius of potential failure!

As you practice infrastructure as code, ask if your configuration or tools align with the practices. For example, ask the following questions about your tool:

- Does the tool allow you to recreate entire environments?
- What happens when you re-run the tool to enforce configuration?
- Can you mix-and-match different configuration snippets to make a new set of infrastructure components?
- Does the tool offer capabilities to help you evolve an infrastructure resource without impacting other systems?

This book will use the principles to answer these questions and provide you with the skill to test, upgrade, and deploy infrastructure with resiliency and scalability in mind.

Exercise 1.1

Choose an infrastructure script or configuration in your organization. Assess if it adheres to the principles of infrastructure as code. Does it promote reproducibility, use idempotency, help with composability, and ease evolvability?

Find hints and answers at the end of the chapter

1.4 Why use infrastructure as code?

Infrastructure as code is typically considered a DevOps practice. However, you don't have to be applying DevOps across your entire organization to use it! You still want to manage your infrastructure in a way that reduces the change failure rate and mean time to resolution (MTTR), so you can sleep in on the weekends as an operator or spend more time writing code as a developer. There are a few reasons to use infrastructure as code, even if you think you don't need it.

1.4.1 Change management

You might experience a sinking feeling when you've applied a change to some infrastructure, only to realize that someone reported it broke something. Organizations try to prevent this with ***change management***, a set of steps and reviews to ensure your changes do not affect production. The process often includes a change review board to review the changes or change windows to block off time to execute the change.

> **DEFINITION** Change management outlines a set of steps and reviews that you take in your company to implement changes in production and prevent their failure.

However, no change is risk-free. Applying infrastructure as code practices can mitigate the risk of a change by modularizing your infrastructure (Chapter 3) and rolling forward changes (Chapter 11) to limit the blast radius.

In one regretful instance, I ignored my intuition to use infrastructure as code to mitigate the risk of a change. I had to roll out a new binary to servers, which required them to restart a set of dependent services. I wrote a script, asked my teammate to check it, and had the change review board sign off to run it. After applying and verifying the change over the weekend, I came in on Monday to several messages telling me servers supporting a reconciliation application went down overnight. My teammate traced it to an older operating system incompatibility with the dependencies in my script.

In retrospect, infrastructure as code could have mitigated the risk of the change. When I applied the RICE principles to the change, I realized that I forgot the following:

- Reproducibility. I did not reproduce my script on various test instances mimicking the various servers.
- Idempotency. I did not include logic to check the operating system before running the command.
- Composability. I did not limit the blast radius of the change to a small set of less critical servers.
- Evolvability. I did not update the servers with a newer operating system and reduce variation across the infrastructure.

Reducing variation allows for infrastructure evolution and risk mitigation because the actual configuration matches the one you expect during your automation, making changes more straightforward and reliable to apply. We'll discuss how to fit infrastructure as code into your change management process in Chapter 7!

1.4.2 Return on time investment

Infrastructure as code and time investment can be challenging to justify, especially if your devices or hardware do not have suitable automation interfaces. In addition to a lack of easy automation, it can be difficult to justify spending time automating a task that you only do once a year or even a decade. While infrastructure as code might take extra time to implement, it lowers the time to execute changes in the long term. How exactly does that work?

Imagine you need to update the same package on ten servers. You used to do this without infrastructure as code. You manually log in, update the package, verify everything works, fix errors, and move to the next one. On average, you spend 10 hours updating the servers.

Figure 1.12 shows that changes made without IaC have a constant level of effort over time. If you have additional changes, you may spend a few more hours fixing or updating the system. A failed change might mean you spend more effort over several days to fix the system.

You decide to invest time into building infrastructure as code for these servers (the solid line in figure 1.12). You reduce the servers' configuration drift, which takes about 40 hours. After your initial time investment, you spend less than five minutes updating all of the servers each time you make the change.

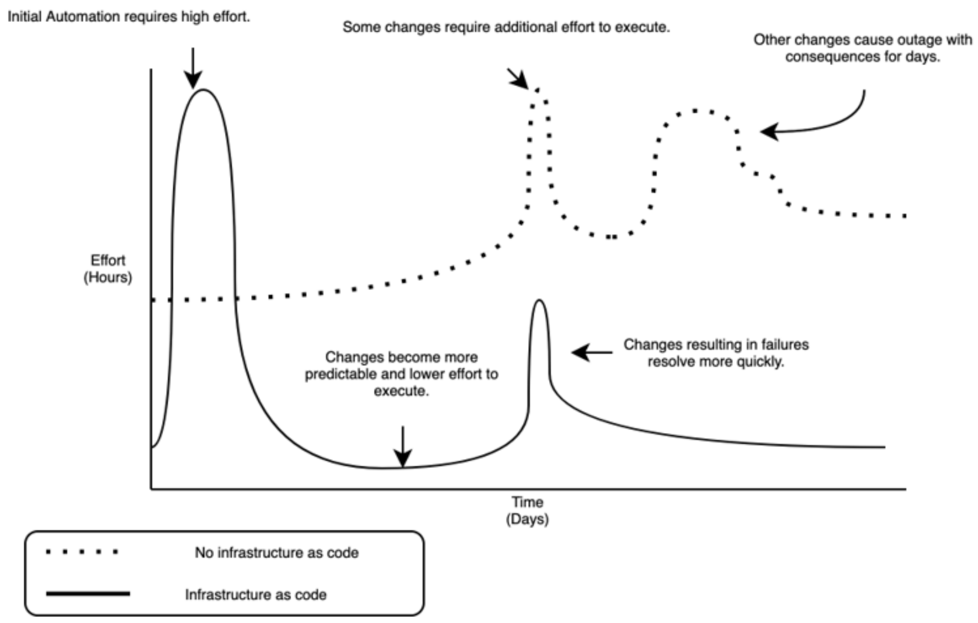


Figure 1.12. Infrastructure as code requires less effort over time after the high initial automation effort. Without it, time to execute changes can be highly variable.

Why bother understanding the relationship between time and effort of infrastructure as code? Prevention helps reduce the effort of remediation! Without infrastructure as code, you might find yourself spending weeks trying to remediate a major system outage. You typically spend those weeks trying to reverse engineer the manual changes, revert specific changes, or at worst, rebuilding a new system from scratch.

You *must* make an initial investment in writing infrastructure as code, even if it seems steep. This investment helps you over time as you spend less time debugging failed configurations or recovering broken systems in the long term. If your system completely fails one day, you can reproduce it easily by running your infrastructure as code.

The automation and tests encourage predictability and limit the blast radius of a failure change. They lower the change failure rate and MTTR of failed systems. As your infrastructure system evolves and scales, you can use the detailed testing practices covered in this book to improve the change failure rate in our system and alleviate the burden of future changes.

1.4.3 Knowledge sharing

Infrastructure as code communicates infrastructure architecture and configuration, which helps reduce human error and improve reliability. An engineer once told me, "We don't need to do infrastructure as code for a network switch for a tertiary passive data center (used for backup). It’s just me configuring stuff anyway, and we’d only need to make this change once and never touch it again.”

The engineer left the organization shortly after configuring the switch. Later, my team needed to convert the tertiary passive data center to an active data center for compliance requirements. In a panic, we scrambled to reverse engineer the configuration on the switch. It took us a good part of two months to figure out the network connection, rework its configuration, and manage it with infrastructure as code.

Even if a task seems particularly obscure or the team configuring infrastructure consists of one person, investing the time and effort to approach infrastructure configuration in an “as code” manner can help accommodate evolution, especially as infrastructure systems and teams scale. You find that you’ll spend more time remembering how you configured the obscure switch when someone reports an outage or teaching the server configuration to a new team member.

Writing a task “as code” communicates the expected state of the infrastructure and the system’s architecture, also known as ***code as documentation***.

> **DEFINITION** Code as documentation ensures that the code communicates the intent of the software or system without the need for additional reference documentation.

Someone unfamiliar with the system should examine the infrastructure configuration and understand its intent. You cannot expect all code to serve as documentation from a practical standpoint! However, the code should reflect most of your infrastructure architecture and system expectations.

1.4.4 Security

Auditing and checking for insecure configurations in infrastructure as code can highlight security concerns earlier in the development process. You hear this as “shifting security left.” If you incorporate security checks earlier in the process, you find fewer vulnerabilities when the system configuration runs in production. You’ll learn more about security patterns and practices in Chapter 8.

For example, you might temporarily increase access to an object store such that anyone can write and read from it in development. You push it to production. However, some of the objects in the store allow everyone to write to and read from them! While this seems like a simple mistake, the configuration has grave implications if the store contains customer data.

Insecure infrastructure in the news

For more examples of insecure infrastructure configuration, you can search the news for a misconfigured object store that exposed driver's license information or even a database with a default password that revealed millions of consumer credit cards. Some security breaches involve legitimate vulnerabilities, but many involve insecure configuration. Organizations that prevent these misconfigurations in the first place can usually quickly examine configuration, audit access control, assess the blast radius, and remediate the breach.

Infrastructure as code simplifies access control by expressing it in a single configuration. With infrastructure as code, you can test the configuration to ensure the object store does not allow public access. Furthermore, you can include a production check to verify your policy only allows read access to specific objects. Even security policies in a data center, such as firewall rules, can be expressed in infrastructure as code and audited to ensure its rules only allow inbound connections from known sources.

If you experience a security breach, infrastructure as code allows you to examine configuration, audit access control quickly, assess the blast radius, and remediate the breach. You can use the same IaC practices to make all sorts of changes. You'll discover some practices in this book to help audit and secure your infrastructure in adherence to infrastructure as code principles.

1.5 Tools

Infrastructure as code tooling varies quite widely because it applies to various resources. Most tools fall into one of three use cases, all of which address very different functions and vary widely in behavior, including the following:

1. Provisioning
2. Configuration Management
3. Image Building

In this book, I primarily focus on tools used for provisioning, which deploy and manage sets of infrastructure resources. I will include some sidebars and examples in configuration management and image building to highlight some differences in approach.

1.5.1 Examples in this book

In this book, I faced a challenge to build concrete examples agnostic of tools or platforms. As a patterns and practices book, I needed to find a way to express the concepts in a general programming language without rewriting the logic of a tool.

Figure 1.13 outlines the workflow of the code listings and examples, and Appendix A details the technical implementation. I wrote the code listings in Python to create a JSON file consumed by HashiCorp Terraform, a provisioning tool with various integrations across public clouds and other infrastructure providers.

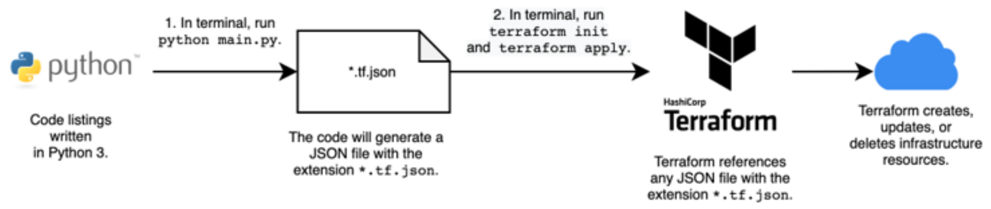

python
Code listings written in Python 3.
1. In terminal, run python main.py.
*.tf.json
The code will generate a JSON file with the extension *.tf.json.
2. In terminal, run terraform init and terraform apply.
HashiCorp Terraform
Terraform references any JSON file with the extension *.tf.json.
Terraform creates, updates, or deletes infrastructure resources.

Figure 1.13. This book uses Python to create a JSON file that HashiCorp Terraform can consume.

When you run the Python script using `python run.py`, the code creates a JSON file with the file extension `*.tf.json`. The JSON file uses syntax specific to Terraform. Then, you can go into the directory with the `*.tf.json` file and run `terraform init` and `terraform apply` to create the resources. While the Python code seems to add unnecessary abstraction, it ensures that I can offer concrete examples agnostic of platform and tool.

I recognize the complexity of this workflow seems nonsensical. However, it serves two purposes. The Python files provide a generalized implementation of patterns and practices with a programming language. The JSON configuration allows you to run and create the resources with a tool instead of me writing in abstractions.

Code repository for this book

You can find the complete code examples at this book's code repository: https://github.com/joatmon08/manning-book.

You do not have to know Python or Terraform in-depth to understand the code examples. If you would like to run the examples and create the resources, I recommend reviewing an introductory tutorial to Terraform or Python for syntax and commands.

For more on HashiCorp Terraform and Python

You'll find a number of resources on HashiCorp Terraform and Python. Check out Terraform in Action by Scott Winkler, The Quick Python Book by Naomi Cedar, or Python Workout by Reuven M. Learner.

The examples create infrastructure in Google Cloud Platform (GCP). While Amazon Web Services (AWS) or Azure may be more popular choices at publication, too many code examples become quickly outdated. Much to my concern, cloud provider APIs change very quickly. I use GCP as the primary cloud provider for two main reasons.

First, if you create a new account with GCP, you will receive a free trial of up to $300 (at time of publication). You can use offerings in the free usage tier if you have an existing account. I will note if an example requires resources that do not qualify for the free usage tier at the time of publication. Furthermore, GCP's billing project structure isolates any resources you create with these examples so you can delete the project and all of its resources in one click.

Using Google Cloud Platform

For more information on GCP's free program, refer to their program webpage at https://cloud.google.com/free.

I recommend creating a separate GCP project to run all of the examples. A separate project will isolate your resources. When you finish this book, you can delete the project and its resources. Review the tutorial for creating GCP projects at https://cloud.google.com/resource-manager/docs/creating-managing-projects.

Second, I use GCP because its offerings use more straightforward naming and generic infrastructure terminology. If you work in a data center, you'll still be able to recognize what the services create. For example, Google's Cloud SQL creates SQL databases.

If you run the examples, you will use the following resources in GCP:

- Networking (networks, load balancers, and firewalls)
- Compute
- Managed queues (PubSub)
- Storage (Cloud Storage)
- Identity and access management (IAM)
- Kubernetes offerings (Kubernetes Engine and Cloud Run)
- Database (Cloud SQL)

In general, you *do not* have to know the details of each service. I use them to demonstrate dependency management between infrastructure resources. Each example includes notations on equivalent AWS and Azure service offerings to further solidify specific patterns and techniques.

To update the examples for the cloud provider of your choice, you may need to do some language replacements and change some of your dependencies. For example, you can create GCP networks with built-in gateways while you must explicitly build them in AWS networks. I avoided using specialized services different for each cloud platform, such as machine learning.

1.5.2 Provisioning

Provisioning tools create and manage sets of infrastructure resources for a given provider, whether it be a public cloud, data center, or hosted monitoring solution. A provider refers to a data center, IaaS, PaaS, or SaaS responsible for providing infrastructure resources.

DEFINITION A provisioning tool creates and manages a set of infrastructure resources for a public cloud, data center, or hosted monitoring solution.

Some provisioning tools only work with a specific provider, while others integrate with multiple providers (table 1.1).

Table 1.1. Examples of provisioning tools & providers

Tool	Provider
Amazon Web Services CloudFormation	Amazon Web Services
Google Cloud Platform Deployment Manager	Google Cloud Platform
Azure Resource Manager	Microsoft Azure
HashiCorp Terraform	Various[1]
Pulumi SDK	Various[2]
AWS Cloud Development Kit	Amazon Web Services
Kubernetes Manifests	Kubernetes (Container Orchestrator)

Most provisioning tools can preview changes to a system and express dependencies between infrastructure resources, also known as a ***dry run***.

> **DEFINITION** A dry run analyzes and outputs expected changes to infrastructure before you apply the changes to resources.

For example, you can express a dependency between a network and a server. If you change the network, the provisioning tool will show that the server may also change.

1.5.3 Configuration management

Configuration management tools ensure that servers and computer systems run in the desired state. Most configuration management tools excel at device configuration, such as server installation and maintenance.

> **DEFINITION** A configuration management tool configures a set of servers or resources for the packages and attributes hosted on them.

[1] For a complete list, see https://www.terraform.io/docs/providers/index.html.
[2] For a complete list, see https://www.pulumi.com/docs/intro/cloud-providers/.

For example, if you have 10,000 servers in your data center, how do you ensure that all of them run a specific version of the packages your security team approved? It does not scale for you to log into 10,000 servers and manually type commands to review. If you configure the servers with a configuration management tool, you can run a single command to review all 10,000 servers and execute the packages' updates.

A non-exhaustive list of configuration management tools that address this problem space includes:

- Chef
- Puppet
- Ansible
- SaltStack
- CFEngine

While this book focuses on provisioning tools and managing multi-provider systems, I will address some configuration management practices related to infrastructure testing, updating infrastructure, and security. Configuration management can help tune your server and network infrastructure.

More on configuration management

I recommend books and tutorials for a configuration management tool of your choice for additional information. They will provide a more detailed guide specific to their design approach.

To add to the confusion, you might notice that some configuration management tools offer integrations to the data center and cloud providers. As a result, you might think of using your configuration tool as your provisioning tool. While possible, this approach may not be ideal because provisioning tools often have a different design approach for specifically addressing dependencies between infrastructure resources. I will explore this nuance in the next chapter.

1.5.4 Image building

When you create a server, you must specify a machine image with an operating system. ***Image building*** tools create images used for application runtime, whether a container or server.

> **DEFINITION** An image building tool builds machine images for application runtimes, such as containers or servers.

Most image builders allow you to specify a runtime environment and build targets. Table 1.2 outlines a few tools, their supported runtime environments, and their build targets and platforms.

Table 1.2. Examples of image builders & providers

Tool	Runtime Environment	Build Target
HashiCorp Packer	Containers & Servers	Various[3]
Docker	Containers	Container Registries
Amazon EC2 Image Builder	Servers	Amazon Web Services
Azure VM Image Builder	Servers	Microsoft Azure

I will not include detailed discussions on image building in this book. However, the patterns in Chapters 6-8 apply to image building workflows. In the next chapter, you'll learn about immutability, a critical paradigm that informs the approach of image builders.

Figure 1.14 shows how image building, configuration management, and provisioning tools work together. The process of deploying a new server configuration often starts with a configuration management tool, where you start building the foundations and testing if your server configuration is correct.

After establishing the server configuration you want, you use an image builder to preserve the server's image with its versioning and runtime. Finally, your provisioning tool references the image builder's snapshot to create new production servers with the configuration you want.

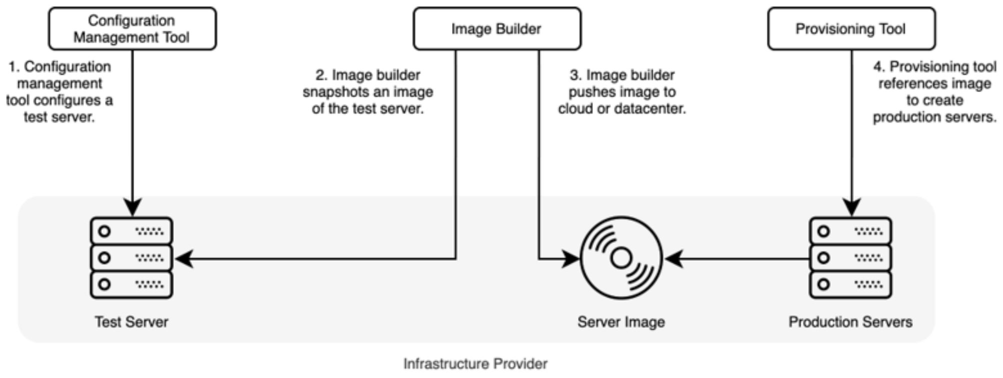

Configuration Management Tool
Image Builder
Provisioning Tool
1. Configuration management tool configures a test server.
2. Image builder snapshots an image of the test server.
3. Image builder pushes image to cloud or datacenter.
4. Provisioning tool references image to create production servers.
Test Server
Server Image
Production Servers
Infrastructure Provider

Figure 1.14. Each type of infrastructure as a code tool contributes to the lifecycle of a server infrastructure resource, from configuration to image capture and deployment.

[3] For a complete list, see https://www.packer.io/docs/builders.

This workflow represents the ideal end-to-end approach for managing and deploying servers with infrastructure as code tooling. However, as you will learn, infrastructure can be complex, and this workflow may not apply to every use case. Various infrastructure systems and their dependencies complicate provisioning, which I chose to be the primary focus of this book and why the examples use provisioning tools.

1.6 Exercises and Solutions

Exercise 1.1

Choose an infrastructure script or configuration in your organization. Assess if it adheres to the principles of infrastructure as code. Does it promote reproducibility, use idempotency, help with composability, and ease evolvability?

Answer:

You can use the following steps to identify if your script of configuration follows infrastructure as code principles:

- Reproducibility: Copy and paste the script or configuration, share it with someone else, and ask them to create the resources without modifying or changing the configuration.
- Idempotency: Run the script a few times. It should not change your infrastructure.
- Composability: Copy a portion of the configuration and build it on top of other infrastructure resources.
- Evolvability: Add a new infrastructure resource to the configuration and verify you can make changes without affecting other resources.

1.7 Summary

- Infrastructure can be software, platform, or hardware that delivers or deploys applications to production.
- Infrastructure as code is a DevOps practice of automating infrastructure to achieve reliability, scalability, and security.
- The principles of infrastructure as code are reproducibility, idempotency, composability, and evolvability.
- By following the principles of infrastructure as code, you can improve change management processes, lower time spent on fixing failed systems in the long term, better share knowledge and context, and build security into your infrastructure.
- Infrastructure as code tools include provisioning, configuration management, and image building tools.

2 Writing infrastructure as code

This chapter covers

- **How the current infrastructure state affects the reproducibility of infrastructure**
- **Detecting and remediating infrastructure drift due to mutable changes**
- **Implementing best practices for writing reproducible infrastructure as code**

Imagine you've created a development environment for a "hello-world" application. You built it organically, adding new components as you needed them. Eventually, you need to reproduce the configuration for production use, which people can publicly access. You also need to scale production across three geographic regions for high availability.

To do this, you must create and update firewalls, load balancers, servers, and databases in new networks for the production environment. Figure 2.1 shows the complexity of the development environment with the firewall, load balancer, server, and database and the components you need to reproduce in production. However, the figure outlines the differences between development and production. The production configuration needs three servers for high availability, expanded firewall rules to allow all HTTP traffic, and stricter firewall rules for the servers to connect to the database. After reviewing all of the differences, you might have a lot of questions about the best and easiest way to make the changes!

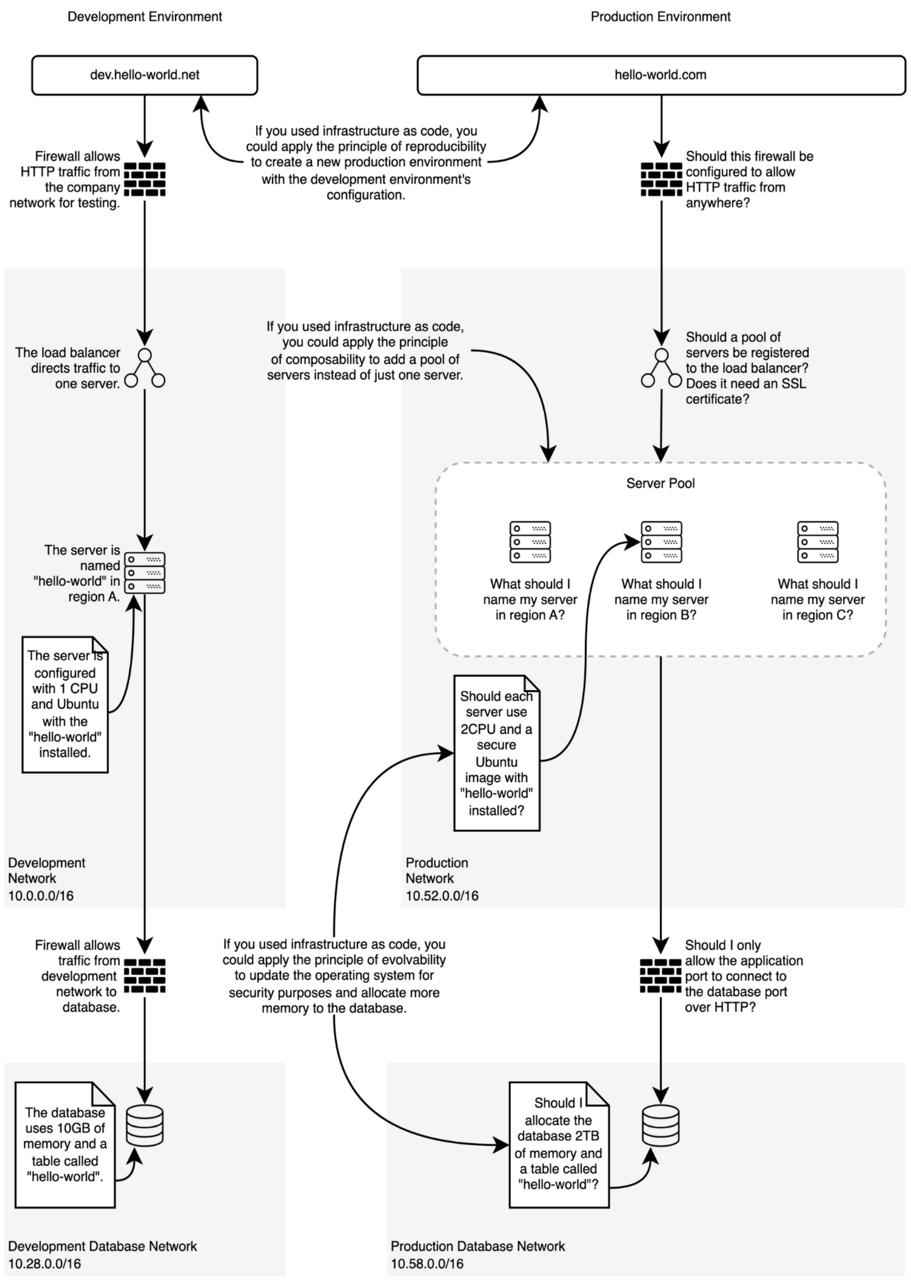

Development Environment
Production Environment
dev.hello-world.net
hello-world.com
If you used infrastructure as code, you could apply the principle of reproducibility to create a new production environment with the development environment's configuration.
Firewall allows HTTP traffic from the company network for testing.
Should this firewall be configured to allow HTTP traffic from anywhere?
If you used infrastructure as code, you could apply the principle of composability to add a pool of servers instead of just one server.
The load balancer directs traffic to one server.
Should a pool of servers be registered to the load balancer? Does it need an SSL certificate?
Server Pool
The server is named "hello-world" in region A.
What should I name my server in region A?
What should I name my server in region B?
What should I name my server in region C?
The server is configured with 1 CPU and Ubuntu with the "hello-world" installed.
Should each server use 2CPU and a secure Ubuntu image with "hello-world" installed?
Development Network 10.0.0.0/16
Production Network 10.52.0.0/16
Firewall allows traffic from development network to database.
If you used infrastructure as code, you could apply the principle of evolvability to update the operating system for security purposes and allocate more memory to the database.
Should I only allow the application port to connect to the database port over HTTP?
The database uses 10GB of memory and a table called "hello-world".
Should I allocate the database 2TB of memory and a table called "hello-world"?
Development Database Network 10.28.0.0/16
Production Database Network 10.58.0.0/16

Figure 2.1. When you create a production environment based on the development, you must answer many questions about configurations for new infrastructure and reverse-engineer the functionality of the development environment.

You might wonder, for example, why the lack of infrastructure as code for the *development* environment affects your ability to create the *production* one? The first reason is that you cannot easily *reproduce* the infrastructure resources. You have to reverse engineer a weeks' worth of manual configuration! With infrastructure as code, you can instead copy and paste some configuration and modify it for the production environment.

Second, you cannot easily *compose* the infrastructure resources with new ones. You need a pool of servers for production instead of a single server. If you built an infrastructure module, you could use that building block to create multiple servers without updating the configuration from scratch.

Finally, you cannot easily *evolve* the production environment with its specific requirements. The production environment requires some different infrastructure resources, like secure operating systems and a larger database. You'll have to manually tweak configuration that you've never run in the development environment!

You can solve these challenges and improve reproducibility, composability, and evolvability in two ways. First, you need a way to migrate manually configured infrastructure to infrastructure as code. Second, you need to write clean infrastructure as code to promote reproducibility and evolvability.

The first part of this chapter outlines some fundamental concepts for writing infrastructure as code (IaC) and migrating existing infrastructure to code. The second part of this chapter applies code hygiene practices to infrastructure. The combination of these practices will help you write reproducible infrastructure as code and set the stage for future composition and evolution of your system.

2.1 Expressing infrastructure change

I mentioned in Chapter 1 that infrastructure as code automates changes. It turns out that reproducing and automating many changes over time takes effort. For example, if you want to provision and manage a server named on Google Cloud Platform (GCP), you'll usually make the following changes over time:

1. Create the server in GCP using the console, terminal, or code.
2. Read the server in GCP to check that you created the server with the correct specifications, like Ubuntu 18.04 as the operating system.
3. Update the server in GCP with a publicly accessible network address to log into it.
4. Delete the server in GCP because you no longer require it.

To make more complex updates or reproduce the server in another environment, you take the following steps:

- Create the server.
- Check if it exists by a "read" command.
- Update it if you need to log in.
- Delete the server if you no longer need it.

No matter which resource you automate, you can always break down your changes to create, read, update, and delete (CRUD). You create an infrastructure resource, search for its metadata, update its properties, and delete it when you no longer need it.

> **NOTE** You wouldn't usually have a change record that explicitly states "read the server." It usually implies a "read" step to verify that a resource is created or updated.

Create, read, update, and delete allow you to automate your infrastructure step-by-step in a specific order. This approach, called the ***imperative style***, describes step-by-step how to configure infrastructure. You can think of it as an instruction manual.

> **DEFINITION** The imperative style of infrastructure as code describes how to configure an infrastructure resource step-by-step.

While it seems intuitive, the imperative style does not scale as you make more changes to the system. Once, I had to create a new database environment based on a development environment. I started reconstructing the 200 change requests submitted to the development environment over two years. Each change request became a series of steps creating, updating, and deleting resources. It took me a month and a half to complete an environment that still didn't match the existing development one!

Rather than painstakingly recreate every step, I wished I could just describe the new database environment based on the running state of the development environment and let a tool figure out how to achieve the state. With most infrastructure as code, you will find it easier to reproduce environments and make changes in the declarative style. The ***declarative style*** describes the desired end state of an infrastructure resource. The tool decides the steps it needs to take to configure the infrastructure resource.

> **DEFINITION** The declarative style of infrastructure as code describes the desired end state of an infrastructure resource. Automation and tooling decide how to achieve the end state without your knowledge.

This process with infrastructure as code takes a few steps. First, I need to search inventory sources for information on the database servers. Next, I get the database IP addresses. Finally, I write a configuration based on the information I've collected.

Your configuration in version control becomes the infrastructure ***source of truth***. You declare the new database environment's desired state instead of describing a set of steps that may not end in the same result.

> **DEFINITION** An infrastructure source of truth structures information about the state of your infrastructure system consistently and singularly.

You make all changes on the infrastructure source of truth. However, even in ideal circumstances (such as with GitOps in Chapter 7), you probably have some configuration drift from manual changes over time. If you use the declarative style and create a source of truth, you can use immutability to change infrastructure and lower the risk of failure.

Exercise 2.1

Given:

```
if __name__ == "__main__":
  update_packages()
  read_ssh_keys()
  update_users()
  if enable_secure_configuration:
    update_ip_tables()
```

Does it use the imperative or declarative style of configuring infrastructure?

A. Imperative style
B. Declarative style

Find answers and explanations at the end of the chapter.

2.2 Understanding immutability

How do you prevent configuration drift and quickly reproduce your infrastructure? It starts with changing how you think about change. Imagine I create a server with Python version 2. I could update my scripts to log into the server and upgrade Python without restarting the server. I can treat the server as ***mutable infrastructure*** because I update the server in place without restarting it.

DEFINITION **You can update mutable infrastructure in place without recreation or restart.**

However, treating the server as mutable infrastructure raises an issue. Other packages on the server do not work with Python 3! Rather than update every other package and break the server, I can change my update scripts to create a *new* server with Python version 3 and compatible dependencies. Then I can delete the old server with Python 2. Figure 2.2 shows how to do this.

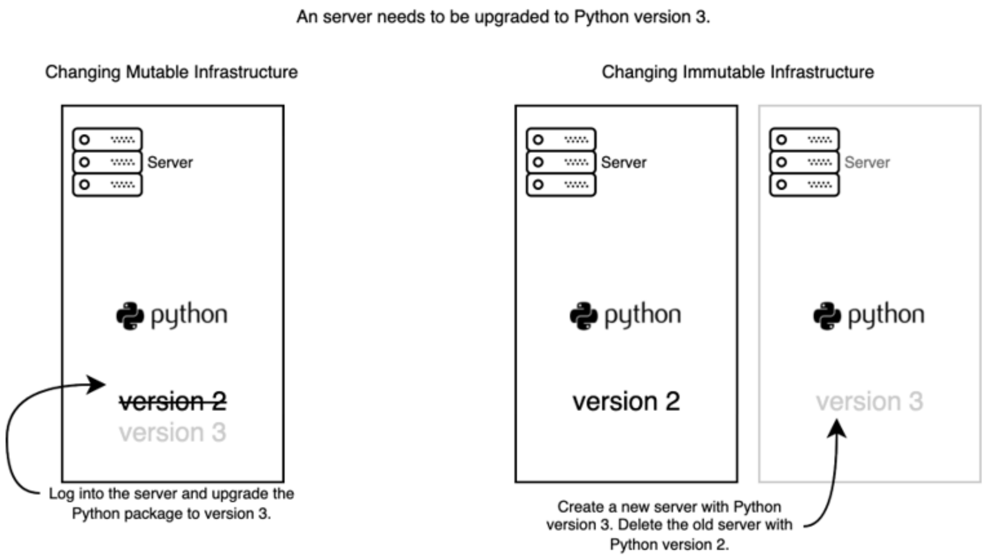

An server needs to be upgraded to Python version 3.
Changing Mutable Infrastructure
Server
python
~~version 2~~
version 3
Log into the server and upgrade the Python package to version 3.
Changing Immutable Infrastructure
Server
python
version 2
Server
python
version 3
Create a new server with Python version 3. Delete the old server with Python version 2.

Figure 2.2. You treat the server mutably by logging in and updating the Python package version. By comparison, you treat the server immutably by replacing the old server with a new one upgraded to Python 3.

My new scripts treat the server as ***immutable infrastructure***, in which I can replace the existing infrastructure with changes. I do not update the infrastructure in place. Immutability means that once you create a resource, you do not change its configuration.

> **DEFINITION** Immutable infrastructure means you must create a new resource for any changes to infrastructure configuration. You do not modify the resource after creating it.

Why treat the server's update in two different ways? Some changes will break the resource if you do them mutably. To mitigate the risk of failure, you can create a whole new resource with the updates and remove the old one with immutability.

Immutability relies on a series of creation and deletion changes. Creating a new resource alleviates drift (difference in actual versus expected configuration) because the new resource aligns with the infrastructure as code you use to create it. You can expand this beyond server resources to even serverless functions or entire infrastructure clusters. You choose to create a new resource with changes instead of updating the existing one.

Immutable image building

Machine image builders work with the concept of immutable infrastructure. Any updates to a server require a new machine image, which the builder generates and provides. Modifications to the server, such as IP address registration, should be passed as parameters to a startup script defined by the image builder.

The enforcement of immutability affects how you make changes. Creating a new resource requires the principle of reproducibility. As a result, infrastructure as code lends well to enforcing immutability as you make changes. For example, you might create a new firewall each time you need to update it. The new firewall overrides any manual rules someone added outside of infrastructure as code, facilitating security and reducing drift.

Immutability also promotes system availability and mitigates any failures to mission-critical applications. Instead of updating an existing resource in place, creating a new one isolates changes to the new resource, limiting the blast radius if something goes wrong. I'll discuss more on this in Chapter 9.

However, immutability sometimes comes at the cost of time and effort. Figure 2.3 compares the effect of mutable versus immutable infrastructure. When I treat the server as a mutable resource, I localize the effect of the Python in-place update. Updating Python affects a small part of the server's overall state. When I treat a server immutably, I replace the *entire* server's state, affecting any resources dependent on that server.

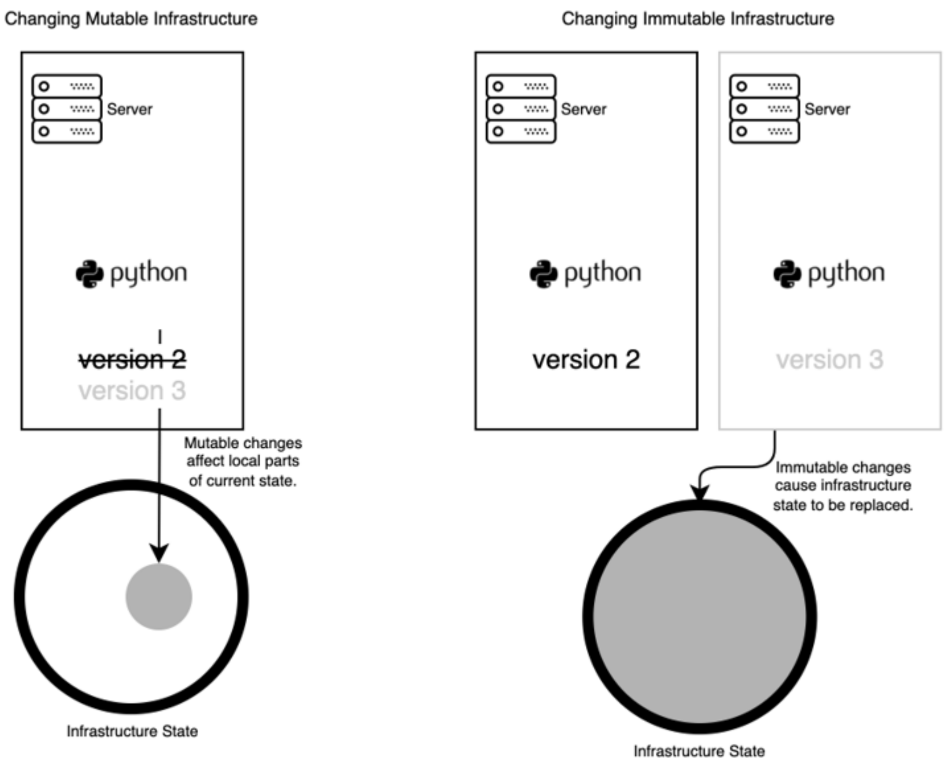

Changing Mutable Infrastructure
Server
python
~~version 2~~
version 3
Mutable changes affect local parts of current state.
Infrastructure State
Changing Immutable Infrastructure
Server
python
version 2
Server
python
version 3
Immutable changes cause infrastructure state to be replaced.
Infrastructure State

Figure 2.3. Changes to mutable resources affect a small portion of the infrastructure state, while immutable resources replace the entire resource state.

Here, replacing the entire state for immutability can take *longer* than changing to mutable infrastructure! You cannot expect to treat all infrastructure immutably all the time. If I change tens of thousands of servers immutably, I spend a few days recreating all of them. An in-place update may only take a day if I don't break anything.

You'll find that you will switch between treating infrastructure as mutable *and* immutable, depending on the circumstance. Immutable infrastructure helps mitigate potential risk of failure across your system, while mutable infrastructure facilitates faster changes. You often treat infrastructure as mutable when you need to fix the system. How do you migrate between mutable and immutable infrastructure?

2.2.1 Remediating out-of-band changes

You cannot expect to deploy a new resource every time you make a change. Sometimes, changes might seem minor in scope and impact. As a result, you decide to make the change mutably.

Imagine you and your friend meet at a coffee shop. Your friend orders a cappuccino with a non-dairy alternative. However, the barista adds milk. They need to make a new cappuccino for your friend because the milk affects the whole cup. Your friend waits for another five to ten minutes. You get a cup of coffee instead and add milk and sugar to taste. If you don't have enough sugar, you just add more.

You take less time to change your mutable coffee than your friend changing their immutable cappuccino. Similarly, it takes far less time, effort, and cost to execute changes to a mutable resource.

When you temporarily treat infrastructure as a mutable resource, you make an ***out-of-band*** change.

> **DEFINITION** An out-of-band change involves a quickly implemented change that temporarily treats immutable infrastructure as mutable infrastructure.

When you break immutability with an out-of-band change, you reduce the change time but increase the risk of affecting another change in the future. After you make the out-of-band change, you need to update your source of truth to return to immutable infrastructure. How do you start this remediation process?

You must reconcile the actual state and desired configuration when making an out-of-band change. I'll apply this to my server example in figure 2.4. First, I log into the server and upgrade Python to version 3. Then, I change the configuration in version control, so new servers install Python version 3. The configuration matches the server's state with the source of truth in version control.

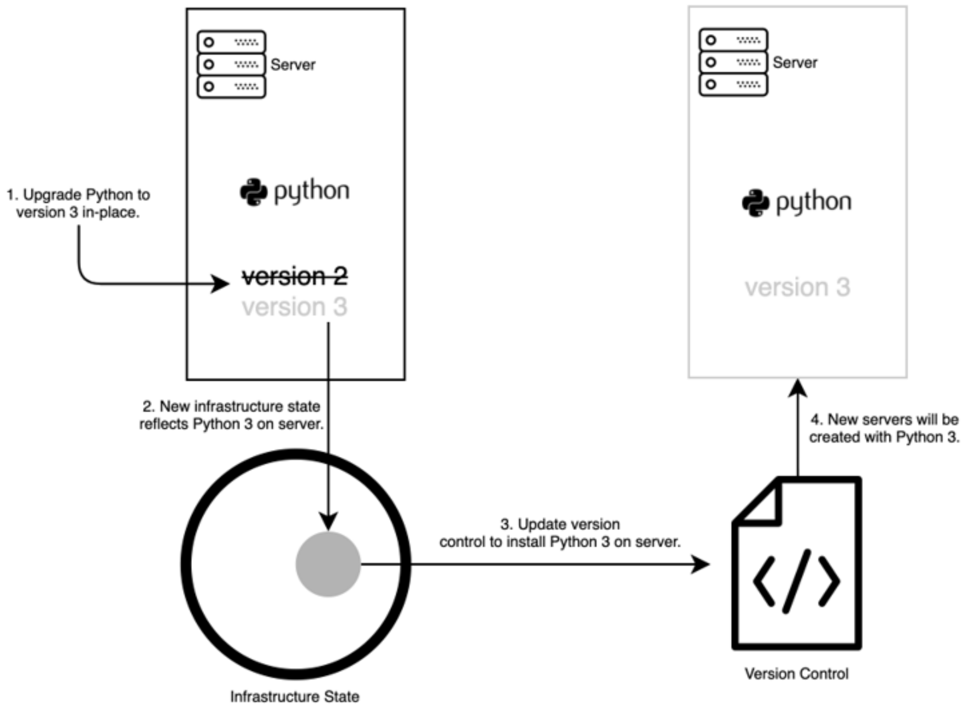

Server
python
1. Upgrade Python to version 3 in-place.
~~version 2~~
version 3
2. New infrastructure state reflects Python 3 on server.
Infrastructure State
3. Update version control to install Python 3 on server.
Version Control
4. New servers will be created with Python 3.
Server
python
version 3

Figure 2.4. After updating a mutable resource, you need to update version control to account for the out-of-band change.

Why should you update infrastructure as code for the out-of-band change? Remember from Chapter 1 that manual changes may affect reproducibility. Making sure that you transition a change made to mutable infrastructure to future immutable infrastructure preserves reproducibility. After remediating the out–of-band change and adding it to infrastructure as code, I can re-deploy changes repeatedly to my server and nothing should change. This behavior conforms to idempotency!

You will continuously reconcile state and source of truth if you make many mutable changes. You should *prioritize immutability* to promote reproducibility. A barista can always replace a drink in my coffee example, even if you spill the sugar container into your mutable coffee. I recommend using your organization’s change procedures to limit out-of-band changes and ensure the updates align with the configuration in infrastructure as code. You can always use the immutable infrastructure configuration to fix a failed mutable change.

Exercise 2.2

Which of the following changes benefit from the principle of immutability? (Choose all that apply.)

A. Reducing a network to have fewer IP addresses
B. Adding a column to a relational database
C. Adding a new IP address to an existing DNS entry
D. Updating a server's packages to backward-incompatible versions
E. Migrating infrastructure resources to another region

Find answers and explanations at the end of the chapter.

2.2.2 Migrating to infrastructure as code

Immutability through infrastructure as code allows version control to manage infrastructure configuration as a source of truth and facilitate future reproduction. In fact, conforming to immutability means that you create new resources all the time. It works very well for *greenfield* environments, which do not have active resources.

However, most organizations have *brownfield* environments, an existing environment with active servers, load balancers, and networks. Recall that the chapter example includes a brownfield development environment called "hello-world." You went into your infrastructure provider and manually created a set of resources.

In general, a brownfield environment treats infrastructure as mutable. You need a way to change your practice of manually changing mutable infrastructure to automatically updating immutable infrastructure as code. How do you migrate the environment's infrastructure resources to immutability?

Let's migrate the "hello-world" development environment to immutable infrastructure as code. Before I begin, I make a list of infrastructure resources in the environment. It contains networks, servers, load balancers, firewalls, and DNS entries.

BASE INFRASTRUCTURE

To start, I find the base infrastructure resource that other resources need to use. For example, every infrastructure resource depends on the network in the development environment. I start writing infrastructure as code for the database and development networks because the server, load balancer, and database run on it. I cannot reconstruct any resources that run on the networks until the networks exist as code.

In figure 2.5, I use my terminal to access the infrastructure provider API. My terminal command prints out the name and IP address range (CIDR block) of the development database and development networks. I reconstruct each network by copying the name and CIDR block of each network into infrastructure as code.

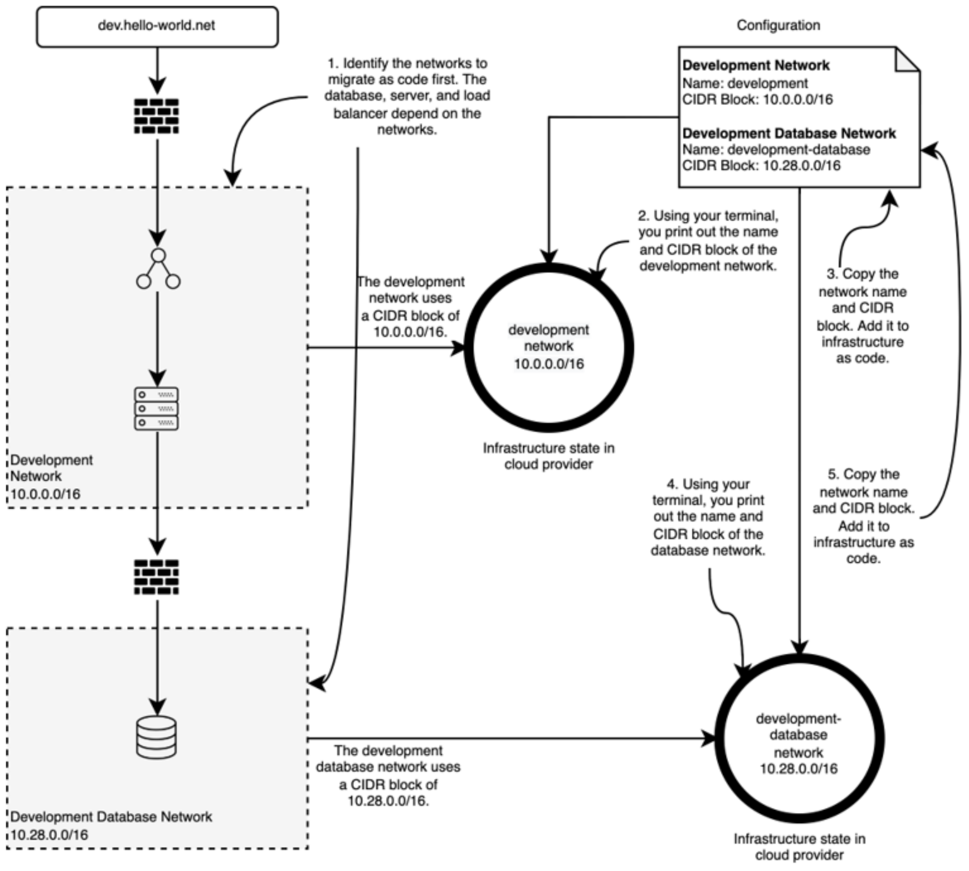

dev.hello-world.net
1. Identify the networks to migrate as code first. The database, server, and load balancer depend on the networks.
Configuration
Development Network
Name: development
CIDR Block: 10.0.0.0/16
Development Database Network
Name: development-database
CIDR Block: 10.28.0.0/16
2. Using your terminal, you print out the name and CIDR block of the development network.
3. Copy the network name and CIDR block. Add it to infrastructure as code.
The development network uses a CIDR block of 10.0.0.0/16.
development network 10.0.0.0/16
Infrastructure state in cloud provider
Development Network 10.0.0.0/16
4. Using your terminal, you print out the name and CIDR block of the database network.
5. Copy the network name and CIDR block. Add it to infrastructure as code.
development-database network 10.28.0.0/16
The development database network uses a CIDR block of 10.28.0.0/16.
Development Database Network 10.28.0.0/16
Infrastructure state in cloud provider

Figure 2.5. Reverse engineer the networks for the database and server first and write their configurations as code.

Why reverse engineer and reproduce the network in infrastructure as code? You must match the infrastructure as code with the network's actual resource state exactly. If you have a mismatch, called drift, you'll find that your infrastructure as code may break your network (and anything on it) by accident!

If possible, import the resource into infrastructure as code state. The resource already exists, and you need your provisioning tool to recognize that. The import step migrates the existing resource to infrastructure as code management! To complete the network resource migration, run the infrastructure as code again and check that you don't have drift.

Many provisioning tools have a function for importing resources. For example, AWS CloudFormation uses the `resource import` command. Similarly, HashiCorp Terraform offers `terraform import`.

If you write infrastructure as code without a provisioning tool, you do not need a direct import capability. Instead, you write code to create a new resource. Sometimes, it's easier to use reproducibility to create a whole new resource. If you cannot easily create a new resource, write code with conditional statements that check if the resources exist.

Figure 2.6 captures the entire decision workflow of reconstructing the network and whether or not you can use a provisioning tool to migrate your resource to immutability. It includes the considerations for creating new resources or writing conditional statements for existing resources. As you migrate, you run your infrastructure as code multiple times to check for drift.

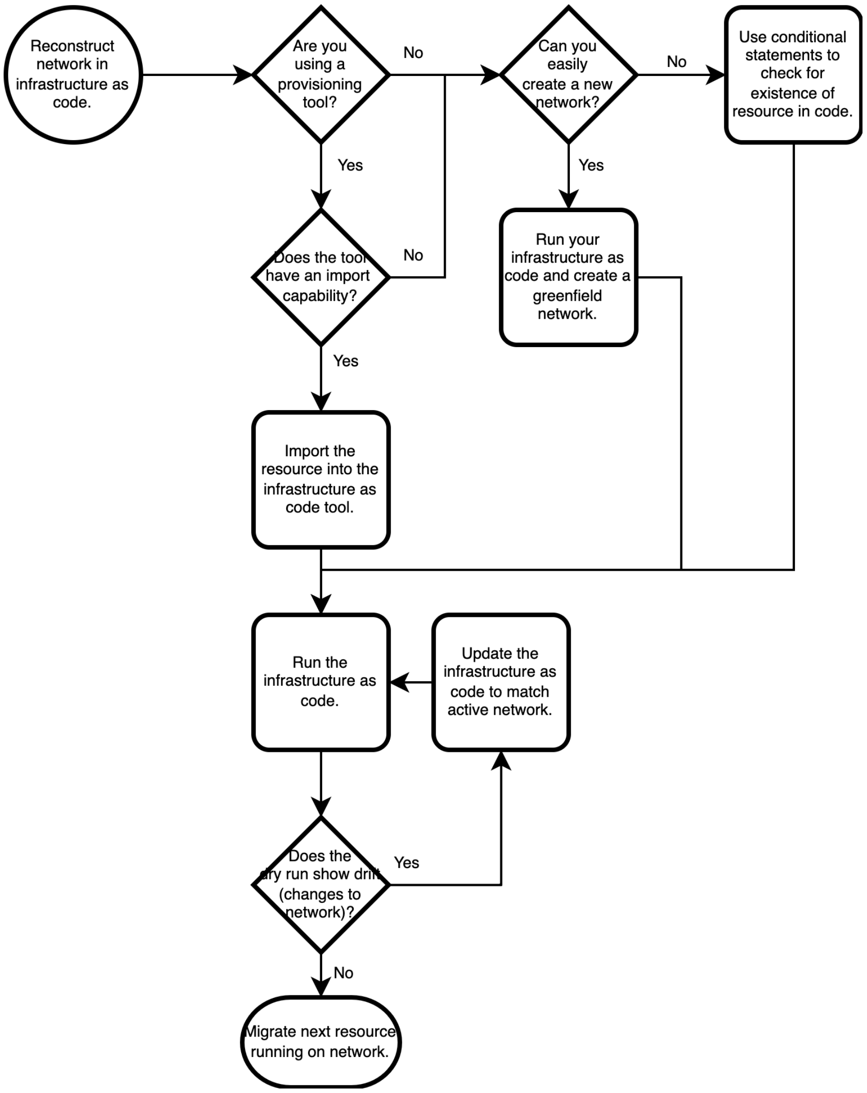

Reconstruct network in infrastructure as code.
Are you using a provisioning tool?
No
Can you easily create a new network?
No
Use conditional statements to check for existence of resource in code.
Yes
Yes
Does the tool have an import capability?
No
Run your infrastructure as code and create a greenfield network.
Yes
Import the resource into the infrastructure as code tool.
Run the infrastructure as code.
Update the infrastructure as code to match active network.
Does the dry run show drift (changes to network)?
Yes
No
Migrate next resource running on network.

Figure 2.6. The decision workflow for migration helps you decide how to import your infrastructure with a provisioning tool, recreate a resource, or build conditional statements to check for resource existence. No matter which option, you must re-run your infrastructure as code and reconcile any drift.

Why do you have so many decision workflows for migrating to immutability? All of these practices adhere to the principles of reproducibility, idempotency, and composability. You want to reproduce the resources in infrastructure as code as accurately as possible. If you can’t import the resource, you can at least reproduce a new one.

Furthermore, re-running the code uses the principle of idempotency, which ensures that you don't recreate the resource (unless necessary). If you reconcile drift, idempotency should not change the active network. Similarly, composability allows you to migrate each resource separately to avoid disrupting the system.

As you work on other resources, keep the decision workflow in mind. You can apply it to each resource you migrate to infrastructure as code until you complete the migration. You'll revisit parts of this decision workflow when you refactor infrastructure as code in Chapter 10.

RESOURCES DEPENDENT ON BASE INFRASTRUCTURE

After reconstructing the base network infrastructure, I can work on the servers and other components. Once again, I use my terminal to print out attributes for the "hello-world" server. It runs in region A with an Ubuntu operating system and 1 CPU. I write the server specification in its configuration, making note of its dependency on the development network. Similarly, I use my terminal to learn that the database uses 10GB of memory. I copy this into infrastructure as code and record its use of the development database network. Figure 2.7 shows the process of migrating the server and database to code.

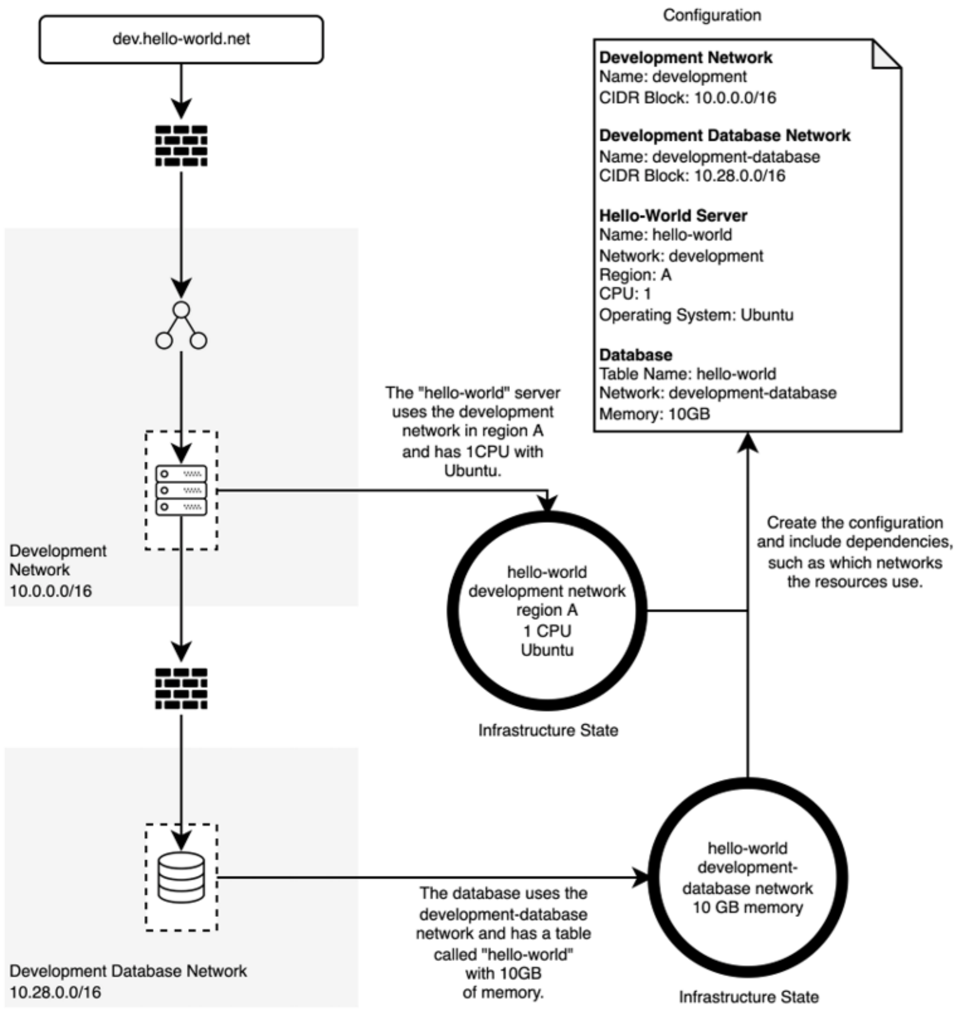

dev.hello-world.net
Configuration
Development Network
Name: development
CIDR Block: 10.0.0.0/16
Development Database Network
Name: development-database
CIDR Block: 10.28.0.0/16
Hello-World Server
Name: hello-world
Network: development
Region: A
CPU: 1
Operating System: Ubuntu
Database
Table Name: hello-world
Network: development-database
Memory: 10GB
The "hello-world" server uses the development network in region A and has 1CPU with Ubuntu.
Create the configuration and include dependencies, such as which networks the resources use.
Development Network 10.0.0.0/16
hello-world development network region A 1 CPU Ubuntu
Infrastructure State
hello-world development-database network 10 GB memory
The database uses the development-database network and has a table called "hello-world" with 10GB of memory.
Development Database Network 10.28.0.0/16
Infrastructure State

Figure 2.7. After migrating base infrastructure such as networks, migrate the server and database resources. They depend on base infrastructure but do not depend on each other.

You want to migrate the second set of resources that use the network. Use composability to isolate these infrastructure resources and make iterative updates. Small changes to next levels of infrastructure help prevent larger system failures. Later in chapter 7, you'll learn more about deploying small changes to infrastructure.

Before you move to the next set of resources, complete the cycle of migration by running your infrastructure as code and checking for drift. Ensure that the network, server, and database do not show changes in their infrastructure as code. After you reconcile any new drift, you can move onto the remaining resources (DNS, firewall rules, and load balancers).

Finally, figure 2.8 rebuilds the remaining configuration for DNS, firewall rules, and load balancers. They depend on the existing configuration of servers and databases. No other resources depend on them.

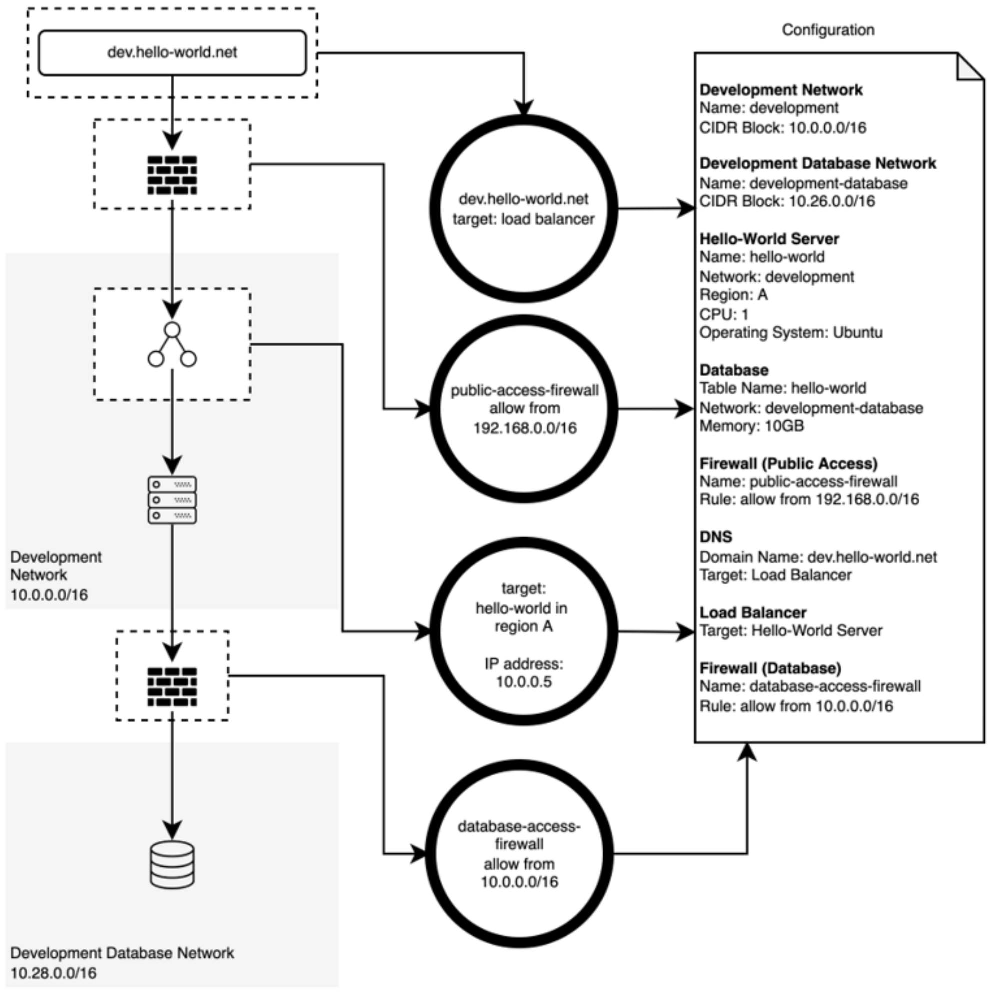


Figure 2.8. Finally, migrate resources with the fewest dependencies or require server and database configuration.

Why go through the painstaking process of reconstructing various levels of infrastructure? Your brownfield environment did not have a consistent *source of truth*, so you need to build one. When I finish adding the infrastructure resources to the configuration, I reconstruct a source of truth for the environment. A source of truth with infrastructure as code allows me to treat the brownfield environment as immutable infrastructure.

Outside of the example, you'll always migrate to immutability from base to top-level resources. Identify resources that others heavily depend upon when you start the migration. Write these low-level resources, such as networks, accounts or projects, and identity and access management, into infrastructure as code.

Next, choose resources such as servers, queues, or databases. Firewalls, load balancers, DNS, and alerts depend on the existence of servers, queues, and databases. You can migrate resources with the fewest dependencies at the end of the process. We'll discuss more about infrastructure dependencies in Chapter 4.

Dependency graphing

A dependency graph represents the dependencies between infrastructure resources. Infrastructure as code tools, such as HashiCorp Terraform, use this concept to apply changes in a structured way. When you migrate resources, you reconstruct the dependency graph. You can investigate tooling that will map the live infrastructure state for you and highlight dependencies to make this easier.

MIGRATION STEPS

I usually follow general steps to assess dependencies and structure migrating existing resources to infrastructure as code.

1. First, migrate **initial login, accounts, and provider resource isolation constructs**. For example, I write the configuration for a cloud provider's account or project and my initial service account for automation.
2. Migrate **networks, subnetworks, routing, and root DNS** configuration, if applicable. The root DNS configuration can include SSL certificates. For example, I created the root domain "hello-world.net" and its SSL certificate to prepare for subdomains such as "dev.hello-world.net."
3. Migrate **computing resources** such as application servers or databases.
4. Migrate the **compute orchestration platform and its components if you use a compute orchestration platform**. For example, I migrate my Kubernetes cluster to schedule workloads across servers.
5. If you use a compute orchestration platform, migrate the **application deployments to the compute orchestration platform**. For example, I backport the configuration of the "hello-world" application deployed on Kubernetes.

6. Migrate **messaging queues, caches, or event streaming platforms**. These services have application dependencies before you can reconstruct them. For example, I write the configuration for a messaging queue to communicate between "hello-world" and another application.
7. Migrate **DNS subdomains, load balancers, and firewalls**. For example, I recreate a configuration for a firewall rule between my "hello-world" application and its database.
8. Migrate **alerts or monitoring** related to resources. For example, I reconstruct my configuration to notify me if the "hello-world" application fails.
9. Finally, migrate **software-as-a-service (SaaS) resources**, such as data processing or repositories, that do not depend on applications. For example, this could be a data transform job on GCP that has a singular dependency on a database.

Between each of these steps, make sure you *test* that you've correctly migrated the initial resources by re-running the configuration. You rarely get all the parameters and dependencies you need on the first try.

NOTE Re-running the migrated configuration should *not* change existing infrastructure because of idempotency. You should re-apply the configuration and check the dry run. Changes in your dry run mean your configuration has not accurately captured the actual state of the resource.

If you run the configuration and it outputs some changes, you must correct your configuration! The process requires trial and error. As a result, I recommend you test and verify each set of resources.

Migrating to immutability becomes an exercise in reducing drift. This process shows an extreme circumstance where the configuration has drifted very far from the state. You work to reconcile the source of truth by updating its configuration in version control. The process of importing existing resources to a new source of truth applies to refactoring infrastructure as code, something we'll discuss in Chapter 10.

2.3 Clean infrastructure as code

Besides using immutability, you can also promote reproducibility by writing configuration cleanly. ***Code hygiene*** refers to a set of practices to enhance the readability and structure of code.

DEFINITION Code hygiene is a set of practices and styles to enhance the readability and maintainability of code.

Infrastructure as code hygiene helps save time when you need to reuse the configuration. I often find infrastructure configuration copied, pasted, and edited with hard-coded values. Hard-coded values reduce readability and reproducibility. While many of these practices come from software development, I suggest some practices specific to infrastructure.

2.3.1 Version control communicates context

How do you use version control effectively to enable reproducibility? Structured practices around version control help you quickly reproduce configuration and make informed changes. For example, you might update a firewall rule in development that allows traffic from "app-network" to "shared-services-network." You add the following commit message to describe why you added the allowance.

```
$ git commit -m "Allow app to connect to queues
      app-network needs to connect to shared-services-network because the application uses
      queues. All ports need to be open."
```

A few weeks later, you reproduce the network in production. However, you forgot why you added the allowance. When you examine the commit history, you remember your descriptive message. You now have information that the application needs to access queues.

When you write commit messages for infrastructure as code, you do not need to explain the configuration. The change already captures what the configuration will be. Instead, use the commit message to explain *why* you want to make the change and *how* it will affect other infrastructure.

More on version control

I address version control practices specific to infrastructure as code. To learn about version control, check out a tutorial (https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control). Check out the practices for more on writing good commit messages (https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project).

You also might have an audit requirement to prefix an issue number or ticket number to the front of the commit message for traceability. For example, I might work on a ticket numbered TICKET-002. It contains a request to allow traffic between the application and shared services. To correlate the ticket to my commit, I add the ticket to the start of the commit message.

```
$ git commit -m "TICKET-002 Allow app to connect to queues
      app-network needs to connect to shared-services-network because the application uses
      queues. All ports need to be open."
```

Adding the work item or ticket information to commit messages makes it easier to track changes. Configuration becomes change documentation because it is the source of truth for infrastructure resources. Version control also becomes a mechanism for documenting changes. You can reconstruct the history of changes and reproduce environments by examining version control and configuration.

2.3.2 Linting & formatting

Before you commit your code, you want to format and lint it. Infrastructure as code often will not execute because you missed a space (or two) or used the wrong field name. The wrong field name could lead to an error. Misaligned code can often cause you to misread or skip a configuration line.

Imagine you configure a server, and it needs a field called "ip_address." Instead, you name the field "ip" and later realize the server cannot create. How can you make sure you've written the field as "ip_address"?

You can use ***linting*** to analyze your code and verify non-standard or incorrect configurations. Most tools offer a way to lint the configuration or code. Linting for "ip_address" catches the wrong field name of "ip" early in development.

DEFINITION **Linting automatically checks the style of your code for non-standard configuration.**

Why check for non-standard or incorrect configuration? You want to make sure you write the proper configuration and you don't miss critical syntax. If the tool does not have a linting feature, you can always find a community extension or write your own linting rules with a programming language. You should include some linting rules that address security standards, such as no secrets committed to version control (Chapter 8).

Besides linting, you can use ***formatting*** to check for spacing and configuration formats. Formatting might seem obvious as a software development practice, but it becomes more critical in infrastructure as code.

DEFINITION **Formatting automatically aligns your code for correct spacing and configuration formats.**

Most tools use domain-specific languages (DSL) that offer a higher level of abstraction for a programming language. A DSL provides a lower barrier to entry if you don't know a programming language. These languages use YAML or JSON data formats with particular format requirements. It helps to have tools to check the formatting, such as whether or not you missed a space in your YAML file!

You can also add version control hooks to run formatting checks before committing your code. For example, you might create my infrastructure resources with Amazon Web Services (AWS) CloudFormation in YAML data format. To validate the infrastructure resource fields and values, you use the AWS CloudFormation Linter (https://github.com/aws-cloudformation/cfn-python-lint). You also format the YAML file with the AWS CloudFormation Template Formatter (https://github.com/awslabs/aws-cloudformation-template-formatter).

Rather than remember to type these commands each time, you can add the commands as a pre-commit Git hook. Each time you run `git commit`, the command checks for a proper configuration and format before pushing them to a repository. You can also add them to a continuous delivery workflow, something I cover in Chapter 7.

2.3.3 Naming resources

When your infrastructure as code becomes documentation, your resources, configuration, and variables need descriptive names. I once created a firewall rule to test something and called it “firewall-rule-1”. Two weeks later, when I wanted to reproduce it into production, I did not remember why I created the rule in development.

In retrospect, I should have named the firewall rule something more descriptive. I spent another thirty minutes tracking down the rule’s IP addresses and allowances. Naming can affect the time you spend deciphering what the infrastructure does and how it differs in another environment.

Resource names should include the **environment**, the infrastructure **resource type**, and its **purpose**. Figure 2.9 names the firewall rule “dev-firewall-rule-allow-hello-world-to-database,” which includes the environment (“dev”), the resource type (“firewall-rule”), and the purpose (“allow-hello-world-to-database”).

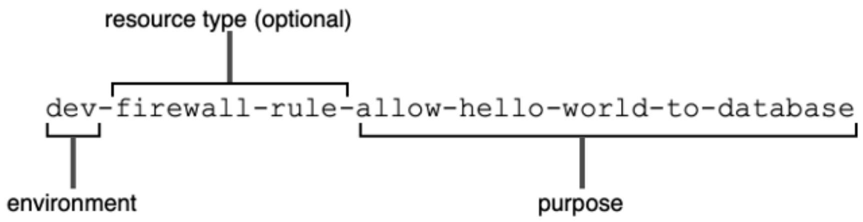


Figure 2.9. The resource name should include the environment, type, and purpose.

Why should names involve so much detail? You want to identify the resource quickly for troubleshooting, sharing, and auditing. Noticing the environment at-a-glance ensures you configure the right one (and not production by accident). The purpose tells others and reminds yourself what the resource does.

Optionally, you can include the resource type. I usually omit resource type from the name because I identify it from resource metadata. Omitting the resource type allows you to conform to your cloud provider’s character limit. If you want to include more information about the purpose or type of your resource, you can always include it in the resource’s tags (chapter 8).

Describe the resource to someone else

When I name a resource, I try to describe it to someone else. If another person understands the resource based on the name, I know it is good. However, I know it needs more information if someone needs to ask additional questions about the environment or resource type. This exercise can make the names a little long, but I err on the side of being more descriptive. Recognizing the resource’s purpose based on its name saves valuable time reconstructing the environment.

Besides resource names, you also want to make variables and configurations as descriptive as possible. Most infrastructure providers have specific resource attribute naming. Amazon Web Services refers to a network's IP address as the "CidrBlock" while Microsoft Azure refers to it as an "address_space."

I lean toward the provider's specific naming to look up documentation for the provider for later changes and reproduction. If I rename the configuration for Microsoft Azure to "cidr_block," I have to remember to translate the parameter to "address_space" for Azure to consume it. You need to remember to translate a more generic field name for variables or configurations to another provider or environment.

2.3.4 Variables & constants

Besides naming variables, how do you know which values should be variables? Let's say the "hello-world" application always serves on port 8080. You don't plan on changing the port often, so you set it to "application_port = 8080" at the beginning of your configuration. However, you hard-code "hello-world" directly into the "name" attributes of your infrastructure resources.

One year later, you reproduce the environment for a new version of "hello-world" on port 3000. You need to call it "hello-world-v2". You update the "application_port" at the beginning of your configuration to 3000. Putting the port in a variable allows you to reference the "application_port" throughout your configuration and store the value in one place. You congratulate yourself on not needing to find and replace instances of "8080" in your configuration. However, spend an hour searching for all instances of "hello-world" in your infrastructure configuration to change its name.

In this example, you have two types of inputs. A ***variable*** stores a value referenced by infrastructure configuration. Most infrastructure values are best stored in variables and referenced by the configuration.

DEFINITION A variable stores a value referenced by infrastructure configuration. You expect to change the value of a variable any time you create a new resource or environment.

You should set the application's name, "hello-world," as a variable because it will change depending on the environment, version, or purpose. However, the port does not change based on environment or purpose. A ***constant*** variable sets a common value across a set of resources and rarely changes with environment or purpose.

DEFINITION A constant variable establishes a common value across infrastructure configuration. You do not change constants often.

When deciding when to make a configuration value a variable or constant, consider the impact and security implications of changing the value. The frequency of change matters less. If changing the value affects infrastructure dependencies or compromises sensitive information, set it as a variable. You should always set names or environments as variables.

Unlike software development, which pushes for fewer constants, infrastructure as code *prioritizes constants over variables*. Avoid setting too many variables because they make the configuration challenging to maintain. Instead, you can set a constant by defining a local variable with a static configuration.

For example, HashiCorp Terraform uses local values (https://www.terraform.io/docs/language/values/locals.html) to store constants. Commonly defined constants include operating systems, tags, account identifiers, or domain names. Standardized values on infrastructure providers such as "internal" or "external" to describe a type of network can also be constant.

2.3.5 Parametrize dependencies

When you create a server, you need to specify the network it needs to use. You initially express this by hard-coding the name of the network you want, specifically "development." When you read the configuration, you know precisely which network the server uses.

However, you realize that when you have to reproduce this for production, you need to search and replace any reference of "development" with "production." Problematically, you have multiple references to "development"! Your search and replace mission becomes a few tedious hours.

CODE EXAMPLE

You decide to parametrize the Google Cloud Platform (GCP) network as a variable so you can reproduce a server in another environment with a different network. When you pass the network name as a variable, you change the network for any server referencing it. Let's pass the name of the network as a variable in code.

Listing 2.1 Parametrize the network as a variable

```
import json


def hello_server(name, network):    #A
   return {
       'resource': [
           {
               'google_compute_instance': [    #B
                   {
                       name: [
                           {
                               'allow_stopping_for_update': True,
                               'zone': 'us-central1-a',
                               'boot_disk': [
                                   {
                                       'initialize_params': [
                                           {
                                               'image': 'ubuntu-1804-lts'
                                           }
                                       ]
                                   }
                               ],
                               'machine_type': 'f1-micro',
                               'name': name,
                               'network_interface': [
                                   {
                                       'network': network    #C
                                   }
                               ],
                               'labels': {
                                   'name': name,
                                   'purpose': 'manning-infrastructure-as-code'
                               }
                           }
                       ]
                   }
               ]
           }
       ]
   }


if __name__ == "__main__":
   config = hello_server(name='hello-world', network='default')    #D

   with open('main.tf.json', 'w') as outfile:     #E
       json.dump(config, outfile, sort_keys=True, indent=4)    #E
```

#A Pass the name and network as parameters to the configuration.
#B Use Terraform's "google_compute_instance" resource to configure a server.
#C Set the network using the "network" variable.
#D Set the network dependency as the "default" network when you run the script.
#E Create a JSON file with the server object and run it with Terraform.

AWS and Azure equivalent

The AWS equivalent of the example would involve deploying an Amazon EC2 instance to the default Virtual Private Cloud (VPC). For Azure, you need to create a virtual network, subnet, network security group, routing table, and virtual machine.

Why pass the name and a network as variables? You often change the name and network depending on the environment. Parametrizing these values helps with reproducibility and composability. You can create new resources on different networks *and* build multiple resources without worrying about conflicts.

Running the example

I'll run the example step-by-step to celebrate our first "hello-world" server. Refer to Chapter 1 for more information on the tools required for the examples and Appendix A for detailed usage instructions.

Run the script in Python by entering the command in the terminal.

```
$ python main.py
```

The command will create a file with the extension `*.tf.json`. Terraform will automatically search for this file extension to create the resources. Check the file exists by listing files in the terminal.

```
$ ls *.tf.json
The output should be as follows:
main.tf.json
```

Authenticate to GCP in the terminal.

```
$ gcloud auth login
```

Set the GCP project you want to use as the `CLOUDSDK_CORE_PROJECT` environment variable.

```
$ export CLOUDSDK_CORE_PROJECT=<your GCP project>
```

Initialize Terraform to retrieve the GCP plugin in the terminal.

```
$ terraform init
```

The output should include the following:

```
Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/google...
- Installing hashicorp/google v3.58.0...
- Installed hashicorp/google v3.58.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Apply the Terraform configuration in the terminal. Ensure you enter "yes" to apply the changes and create the instance.

```
$ terraform apply
```

Your output should include the configuration and name of the server instance.

```
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

google_compute_instance.hello-world: Creating...
google_compute_instance.hello-world: Still creating... [10s elapsed]
google_compute_instance.hello-world: Still creating... [20s elapsed]
google_compute_instance.hello-world: Creation complete after 24s
       [id=projects/infrastructure-as-code-book/zones/us-central1-a/instances/hello-world]

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

For more about Terraform

I will not go through all of the nuances of Terraform. For detailed information on getting started with Terraform, check out https://learn.hashicorp.com/terraform. You can find additional documentation on how Terraform works with Google Cloud Platform at https://registry.terraform.io/providers/hashicorp/google/latest/docs.

You can examine the GCP console for the server's network and metadata. Otherwise, you can use the Cloud SDK CLI to check the network in the terminal. Enter the command to filter out the "hello-world" server.

```
$ gcloud compute instances list --filter="name=( 'hello-world' )" \
  --format="table(name,networkInterfaces.network)"
```

The output should include the GCP URL of the network.

```
NAME         NETWORK
hello-world  ['https://www.googleapis.com/compute/v1/projects/<your GCP
       project>/global/networks/default']
```

The GCP server uses the "default" network, which I passed as a variable to my example. If I want to change the network, I update the new variable. My infrastructure-as-code tooling will pick up the changes and create a new server.

To destroy the server, you can use Terraform in the terminal. Make sure you enter "yes" to remove the server altogether.

```
$ terraform destroy
```

AWS and Azure equivalent

In AWS, you would use the "aws_instance" Terraform resource with a reference to the default network (https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance). For a complete example, refer to the code repository at https://github.com/joatmon08/manning-book.

In Azure, you would use the "azurerm_linux_virtual_machine" Terraform resource with references to the virtual network and subnets you created (https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/linux_virtual_machine).

When you define dependencies as variables, you loosely couple the two infrastructure resources. Chapter 4 will cover specific patterns you can use to decouple infrastructure resources and dependencies further. If possible, you should avoid hard-coding dependencies and pass them in as parameters.

2.3.6 Keeping it a Secret

Infrastructure as code often needs to use ***secrets*** such as tokens, passwords, or keys to execute changes to a provider.

DEFINITION A secret is a piece of sensitive information such as a password, token, or key.

When I create servers in Google Cloud Platform (GCP), I need a service account key or token that accesses the project and server resources. To ensure I can create resources, I maintain the secrets as part of the infrastructure configuration. Secrets in configuration can be problematic. If someone can read my secret, they can use it to access my GCP account to create resources and access restricted data!

You might also need to pass secrets as part of the configuration. For example, you use infrastructure as code to set the SSL certificate for a load balancer. The SSL certificate expires in two years. You recreate the environment two years later. However, you discover that the encrypted string of the certificate has expired. You cannot decrypt it and must now issue a new certificate.

Figure 2.10 shows how to best secure your certificate but improve its evolvability in the future. You pass the certificate as an input variable to have different ones for each environment. Then, you put the new certificates in a secrets manager, which stores and manages the certificate for you.

Any time the certificate changes, you update it in the secrets manager. Your IaC updates its configuration when it reads the certificate from the secrets manager. Separating concerns for certificate management from configuration mitigates any problems you have later with certificate expiration.

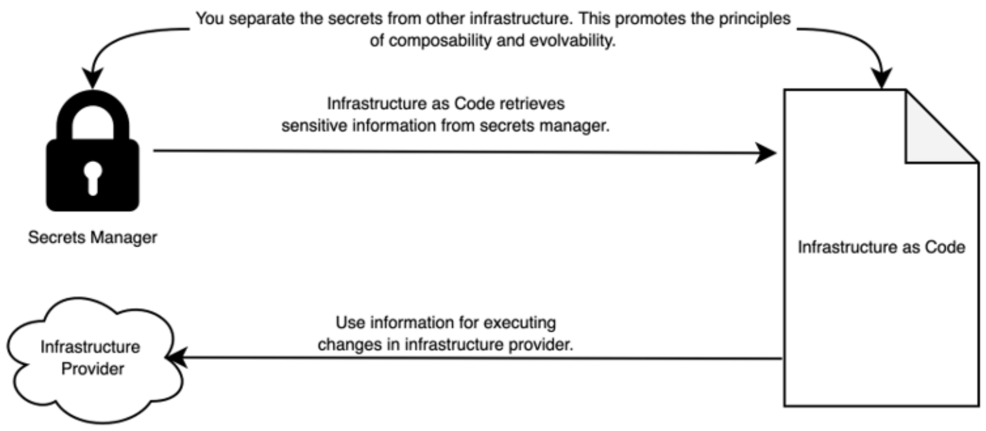

You separate the secrets from other infrastructure. This promotes the principles of composability and evolvability.
Infrastructure as Code retrieves sensitive information from secrets manager.
Secrets Manager
Infrastructure as Code
Use information for executing changes in infrastructure provider.
Infrastructure Provider

Figure 2.10. Retrieve sensitive information from a secrets manager to change resources with an infrastructure provider.

Why store secrets outside of infrastructure as code? You just applied the principles of composability and evolvability to separating secrets from other infrastructure resources. This separation ensures that someone can't examine your infrastructure as code to get a password or username. You also minimize the impact of failure when you rotate a secret by re-running the infrastructure as code.

Always pass secrets as variables into infrastructure as code and use it in memory. These include SSH keys, certificates, private keys, Application Programming Interface (API) tokens, passwords, and other login information. A separate entity should store and manage sensitive authentication data, such as a secrets manager. Separate secrets management facilitates reproduction, especially when you want different passwords and tokens for each environment. You should never hard-code or commit secrets to version control in plaintext.

2.4 Exercises and Solutions

Exercise 2.1

Given:

```
if __name__ == "__main__":
  update_packages()
  read_ssh_keys()
  update_users()
  if enable_secure_configuration:
     update_ip_tables()
```

Does it use the imperative or declarative style of configuring infrastructure?

A. Imperative style
B. Declarative style

Answer:

The code snippet uses the imperative style (A) of configuring infrastructure. It defines how to configure a server *step-by-step* in a specific sequence rather than declaring a specific target configuration.

Exercise 2.2

Which of the following changes benefit from the principle of immutability? (Choose all that apply.)

A. Reducing a network to have fewer IP addresses
B. Adding a column to a relational database
C. Adding a new IP address to an existing DNS entry
D. Updating a server's packages to backward-incompatible versions
E. Migrating infrastructure resources to another region

Answer:

The correct answers are A, D, and E. In each change, they benefit from a new set of resources that implement the change. If you try to make the changes in place, you may accidentally bring down the existing system. For example, reducing a network to have fewer IP addresses may displace any resources using the network.

Updating packages to backward-incompatible versions means that a broken package update can affect the server's ability to handle user requests. Migrating infrastructure resources to another region takes time. Not all cloud provider regions support every type of resource. Creating new resources helps mitigate problems with migration. You can make changes for answers B and C mutably, likely without affecting the system

2.5 Summary

- Prioritizing immutability reduces configuration drift, maintains a source of truth, and improves reproducibility.
- To conform to immutability, changes to a resource create an entirely new resource and replace its state.
- If you make mutable changes, you must reconcile the localized changes in the infrastructure state with your configuration.
- When writing infrastructure as code, use commits in version control to communicate changes and context and format the code for readability.
- Parametrize names, environments, and dependencies to other infrastructure. If you scope the configuration attributes to a resource, you can set it as a constant.
- Secrets should always be passed as variables and never hard-coded or committed to version control in plaintext.
- When writing scripts, always simplify actions to create, read, update, and delete commands to reproduce resources.

3 Patterns for infrastructure modules

This chapter covers

- **Grouping infrastructure resources into composable modules based on function**
- **Building infrastructure modules with software development design patterns**
- **Applying module patterns to common infrastructure use cases**

In the last chapter, I covered the fundamental practices for infrastructure as code (IaC). Even though I knew the fundamental practices, my first Python automation script grouped code into one file with messy functions. Years later, I learned software design patterns. They provided a standard set of patterns that made it easier for me to change the script and hand it over to another teammate for maintenance.

In the following two chapters, I will show how to apply design patterns to infrastructure as code configuration and dependencies. Software ***design patterns*** help you identify common problems and build reusable, object-oriented solutions.

DEFINITION **A design pattern is a repeatable solution to a common problem in software.**

Applying software design patterns to IaC has its pitfalls. Infrastructure as code has reusable objects (as infrastructure resources). However, its opinionated behaviors and domain-specific languages do not map directly to software design patterns.

Infrastructure as code offers an immutable layer of abstraction, which is why this chapter borrows both creational (used for creating objects) and structural (for structuring objects) design patterns to make approximations to infrastructure. Most infrastructure as code focuses on immutability, which automatically creates a new resource upon changes. As a result, the design patterns that rely on mutability do not apply.

For more on software design patterns

I adapted many patterns from Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides to infrastructure as code. If you would like to learn more about the original software design patterns, I recommend referencing their book!

I include Python code listings that create Terraform JSON files. They reference Google Cloud Platform (GCP) resources. You can extend the patterns to domain-specific languages (DSLs), such as HashiCorp Terraform, AWS CloudFormation, or Azure Bicep. Depending on the domain-specific language and tool you choose, it might use different mechanisms or features. When possible, I will note limitations for DSLs and equivalents for AWS and Azure.

3.1 Singleton

Imagine you need to create a set of database servers in Google Cloud Platform (GCP) from scratch. The database system needs a GCP project, custom database network, server template, and server group. The server template installs the packages on each server, while the server group describes how many database servers you need.

Figure 3.1 shows how to add the project, database network, server template, and server group to one directory. You determine the attributes for the GCP project name and its organization. Next, you figure out the network should have the name “development-database-network” with an IP address range of 10.0.3.0/16. Finally, you express that the database should have three servers that use templates for MySQL. You write all these attributes as code into one configuration file.

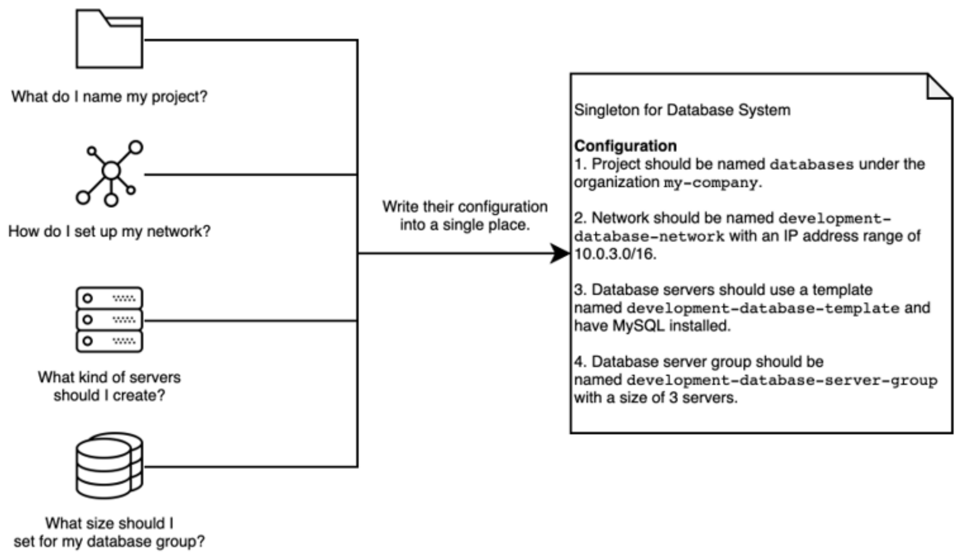

What do I name my project?
How do I set up my network?
What kind of servers should I create?
What size should I set for my database group?
Write their configuration into a single place.
Singleton for Database System
Configuration
1. Project should be named databases under the organization my-company.
2. Network should be named development-database-network with an IP address range of 10.0.3.0/16.
3. Database servers should use a template named development-database-template and have MySQL installed.
4. Database server group should be named development-database-server-group with a size of 3 servers.

Figure 3.1. A singleton expresses the configuration of an initial set of resources in one file, such as a project, network, and database. It captures the relationships between resources in one place.

The database system configuration uses the ***singleton pattern***, which declares a set of resources as singular instances in a system.

> **DEFINITION** The singleton pattern declares a set of resources as singular instances in a system. It deploys all the resources with a single command.

Why do we call it a singleton pattern? You create one file or directory with a static configuration. Furthermore, you define several parameters inline to create all infrastructure resources with a single command. That configuration expresses resources unique and specific to the environment created by the configuration.

The singleton pattern simplifies writing infrastructure as code because you put everything in one configuration! When you express every infrastructure resource in a single configuration, you get a single reference to debug and troubleshoot the provisioning order and required parameters.

However, starting with the singleton pattern often leads to challenges later. When I started using the singleton pattern, I treated infrastructure configuration like a junk drawer. A junk drawer stores random items that don't have a place otherwise. The drawer becomes the first place you look if you cannot find something (figure 3.2).

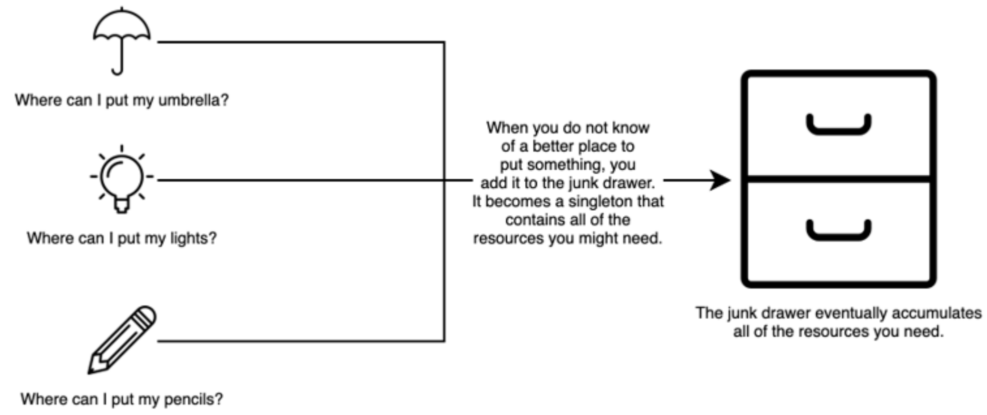

Where can I put my umbrella?
Where can I put my lights?
Where can I put my pencils?
When you do not know of a better place to put something, you add it to the junk drawer. It becomes a singleton that contains all of the resources you might need.
The junk drawer eventually accumulates all of the resources you need.

Figure 3.2. If you do not know where to put an object, add it to the junk drawer, which uses the singleton pattern to aggregate all resources.

Why do I call it a junk drawer? I didn't know where else to configure infrastructure resources, so I just added them to one file. Eventually, the singleton pattern became messy like a junk drawer! I had to search the singleton for an infrastructure resource. In addition, the number of infrastructure resources in the singleton means it takes time to identify, change, and create resources.

The singleton pattern challenges reproducibility as your system grows with more resources. Producing the configuration for production means copying and pasting it into a new file. Copying and pasting configuration does not scale when you have more resources to update! Singletons come at the cost of scalability and composability. It works for a few infrastructure resources but doesn't scale with complex systems.

When should you use a singleton? It works best when you have a resource that only requires a single instance and rarely changes, such as the GCP project. The network, database server template, and server group must go into another configuration.

All GCP projects must have a unique identifier, which makes it ideal for the singleton pattern. The project can only have a single instance. For example, you could create a project named "databases" and generate a unique identifier based on your current system username. Here's the code to implement the singleton pattern for creating a GCP project with your system username.

Listing 3.1 Creating a project in GCP using a singleton pattern

```
import json
import os

class DatabaseGoogleProject:
   def __init__(self):    #A
       self.name = 'databases'  #B
       self.organization = os.environ.get('USER')  #C
       self.project_id = f'{self.name}-{self.organization}'  #D
       self.resource = self._build()    #E

   def _build(self):
       return {
           'resource': [
               {
                   'google_project': [    #B
                       {
                           'databases': [
                               {
                                   'name': self.name,    #B
                                   'project_id': self.project_id    #D
                               }
                           ]
                       }
                   ]
               }
           ]
       }


if __name__ == "__main__":
   project = DatabaseGoogleProject()  #E

   with open('main.tf.json', 'w') as outfile:  #F
       json.dump(project.resource, outfile, sort_keys=True, indent=4)  #F
```

#A Create an object for the database Google project.
#B Set up the Google project using a Terraform resource with the name “databases”.
#C Get the operating system user and set it to the organization variable.
#D Make a unique project ID based on the project name and operating system user so GCP can create the project.
#E Create a DatabaseGoogleProject to generate the JSON configuration for the project.
#F Write it out to a JSON file to be executed by Terraform later.

AWS and Azure equivalent

You can equate a GCP project to an AWS account. To automate the creation of AWS accounts, you will need to use AWS Organizations (https://aws.amazon.com/organizations/).

In Azure, you would use a resource group. You can create a resource group using the “azurerm_resource_group” Terraform resource (https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/resource_group).

Imagine I want to create a server in my “database” project. I can call the `DatabaseGoogleProject` singleton and extract the project identifier from the JSON configuration. A singleton contains unique resources that you can reference with a call to the module. For example, if I reference my “database” project, I will always get the correct project and not a different one.

You use a singleton for a GCP project because you create it *once* and change it *infrequently*. You can apply the singleton pattern to rarely changed *global* resources such as provider accounts, projects, domain name registrations, or root SSL certificates. It also works for static environments, such as low-use datacenter environments.

3.2 Composite

Instead of expressing the database system in one singleton, you can organize the components into modules. A ***module*** groups infrastructure resources that share a function or business domain. Modules allow you to change the automation of the parts without affecting the whole.

> **DEFINITION** A module organizes infrastructure resources by function or business domain. Other tools or resources may refer to them as infrastructure stacks or sets.

You can use modules like building blocks to construct your system. Other teams can use modules as building blocks for their unique infrastructure system.

Figure 3.3 shows how your company might organize itself into teams and create a reporting structure. Each team or manager reports to another manager, leading to the executive level. The company uses composition of teams to achieve a common objective!

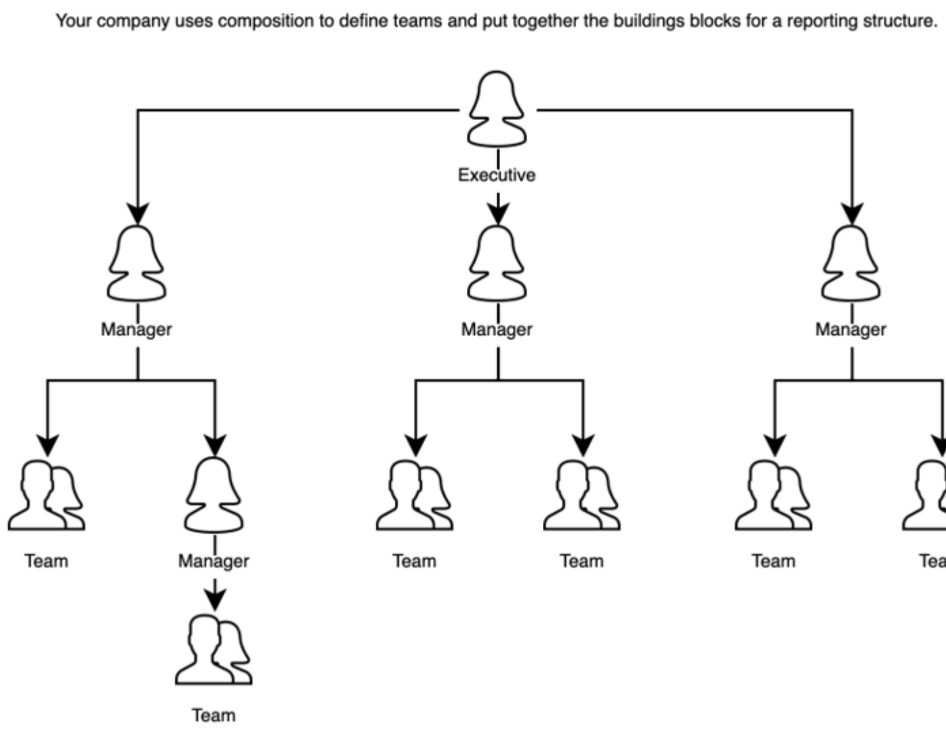

Your company uses composition to define teams and put together the buildings blocks for a reporting structure.
Executive
Manager
Manager
Manager
Team
Manager
Team
Team
Team
Team
Team

Figure 3.3. Your company might use the composite pattern to group employees into a reporting structure, thus allowing managers to organize teams and their goals.

Why would your company break out reporting structures into modules? The pattern ensures new initiatives or business opportunities have a team to support them. It promotes composability and evolvability as the company grows.

In a similar manner, most infrastructure as code depends on the ***composite pattern*** to group, rank, and structure a set of modules. You'll often find the composite pattern classified as a structural pattern because it structures objects in a hierarchy.

> **DEFINITION** The composite pattern treats an infrastructure module as a single instance and allows you to assemble, group, and rank different modules in a system.

Tools usually have their own modularization feature. HashiCorp Terraform or Azure Bicep use their own module frameworks to nest and organize modules. You can use nested stacks or StackSets in AWS CloudFormation to reuse templates (modules) or create stacks across regions. A configuration management tool like Ansible lets you build top-level playbooks that import other tasks.

How do you implement a module? Imagine you need to set up a network for the database servers. However, the server needs a subnetwork (subnet). I composite the network and subnet into a module in figure 3.4. I determine how I set up my network first and write that into my module. Then, I write down the configuration for the subnetwork and add it to the module.

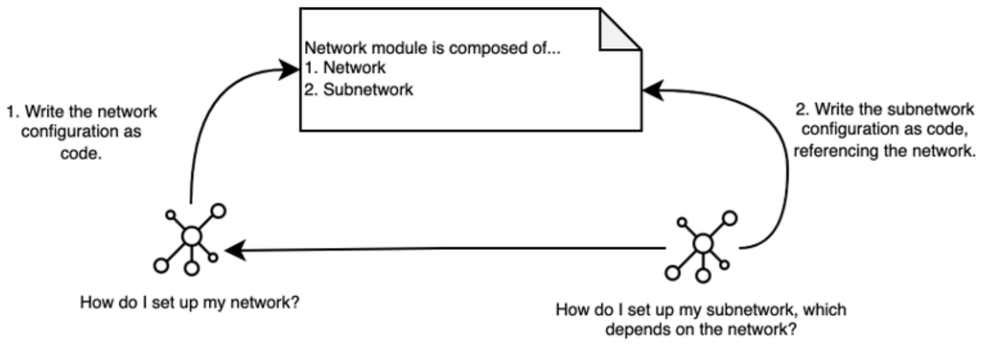

Network module is composed of...
1. Network
2. Subnetwork
1. Write the network configuration as code.
2. Write the subnetwork configuration as code, referencing the network.
How do I set up my network?
How do I set up my subnetwork, which depends on the network?

Figure 3.4. A network module can use the composite pattern to group the network and subnetwork resources.

The module contains the configuration for both the network and the subnetwork. If I need to reproduce the network system in my production environment, I can always copy and paste the entire module to create a new network and subnetwork. A composite pattern for the network ensures that I can always reproduce a group of resources that depend on each other.

You can implement the composite pattern for the network configuration in a module. As listing 3.2 shows, the module creates a network and subnet. You pass the CIDR ranges for the network and subnet, and the module generates a standardized name for the network.

Listing 3.2 Creating a network and subnetwork

```
import json


class Network:    #A
   def __init__(self, region='us-central1'):    #E
       self._network_name = 'my-network'    #B
       self._subnet_name = f'{self._network_name}-subnet'    #C
       self._subnet_cidr = '10.0.0.0/28'    #D
       self._region = region    #E
       self.resource = self._build()    #G

   def _build(self):    #G
       return {
           'resource': [
               {
                   'google_compute_network': [    #B
                       {
                           f'{self._network_name}': [    #B
                               {
                                   'name': self._network_name    #B
                               }
                           ]
                       }
                   ]
               },
               {
                   'google_compute_subnetwork': [    #F
                       {
                           f'{self._subnet_name}': [    #C
                               {
                                   'name': self._subnet_name,    #C
                                   'ip_cidr_range': self._subnet_cidr,    #D
                                   'region': self._region,    #E
                                   'network':
       f'${{google_compute_network.{self._network_name}.name}}'    #F
                               }
                           ]
                       }
                   ]
               }
           ]
       }


if __name__ == "__main__":
   network = Network()    #G

   with open('main.tf.json', 'w') as outfile:    #H
       json.dump(network.resource, outfile, sort_keys=True, indent=4)    #H
```

#A Create a module for the network, which uses the composition pattern to bundle the network and subnet together.
#B Set up the Google network using a Terraform resource with the name “my-network”. GCP does not require a network CIDR block to be defined.
#C Set up the Google subnetwork using a Terraform resource with the name “my-network-subnet”.
#D Set the subnet’s CIDR block as “10.0.0.0/28”.
#E Set the region to the default region, “us-central1”.

#F Create the Google subnetwork on the network using a Terraform variable. Terraform dynamically references the network ID and inserts it to the subnetwork's configuration for you.
#G Use the module to create the JSON configuration for the network and subnetwork.
#H Write it out to a JSON file to be executed by Terraform later.

AWS and Azure equivalent

You can equate a GCP network and subnets to an AWS Virtual Private Cloud (VPC) and subnets or Azure virtual network and subnets. However, in AWS and Azure, you will need to define gateways and routing tables in each subnet. GCP automatically defines these for you when you create the network.

Why compose the module with the network and subnetwork? You can't use the GCP network unless you have a subnet! Composition allows you to create a set of resources together. Bundling the required resources together helps your teammate who might not know much about networking. The composite pattern improves the principle of composability because it groups and organizes common resources that you must deploy as one unit.

The composite pattern works well for infrastructure because infrastructure resources have a hierarchy. A module following this pattern reflects relationships between resources and facilitates their management. If I need to update routing, I update the network composite configuration. I can refer to the network configuration to determine subnet CIDR ranges and calculate network address space.

You apply the composite pattern by grouping resources based on function, business unit, or operational responsibility. As you draft the initial module, you may add variables to allow more flexible parameters or distribute the configuration to other teams. I'll discuss how to share modules in Chapter 5. However, you can apply other patterns outside of a general composite pattern to further improve the reproducibility of infrastructure as code.

3.3 Factory

Previously, you used the singleton pattern to create a GCP project for the database system. Then, you apply the composite pattern and build the network in a different module. However, you realize you need a database network divided into three subnets! Rather than copying and pasting three subnets, you want to create a configuration that accepts inputs with the subnet name and IP address range.

A configuration to create three subnets and a network requires many parameters, which can be tedious to include and maintain. What if you had something like a factory to manufacture all of the resources with opinionated defaults? Figure 3.5 shows that you can create a network factory to stamp out similar networks. You can reduce the required parameters to two inputs and other configurations to default values.

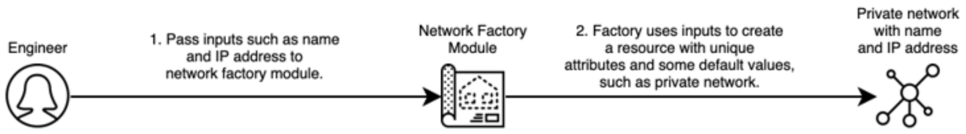

Engineer
1. Pass inputs such as name and IP address to network factory module.
Network Factory Module
2. Factory uses inputs to create a resource with unique attributes and some default values, such as private network.
Private network with name and IP address

Figure 3.5. A factory pattern module includes a set of defaults for a minimal resource configuration and enables customization by accepting input variables.

When you know the network has specific default attributes, you can minimize the inputs and produce multiple resources with less effort. I call this approach the ***factory pattern***. A module that uses the factory pattern takes a set of inputs, such as name and IP address range, and creates a set of infrastructure resources based on the inputs.

> **DEFINITION** The factory pattern takes a set of input variables and creates a set of infrastructure resources based on input variables and default constants.

You want to offer just enough flexibility to make changes, such as the subnet's IP address and name. In general, you need to find a balance in your module between offering just enough customization and opinionated default attributes. After all, you want to promote the principle of reproducibility but maintain the resource's evolvability. We'll discuss more about sharing standard practices for resources in chapters 5 and 8.

Let's return to our example. How do you create three subnets without passing a list of names? The module can automatically name the subnets so you avoid hard-coding them! Figure 3.6 shows how to add some logic into the network factory module to standardize the name of the subnet based on the network address.

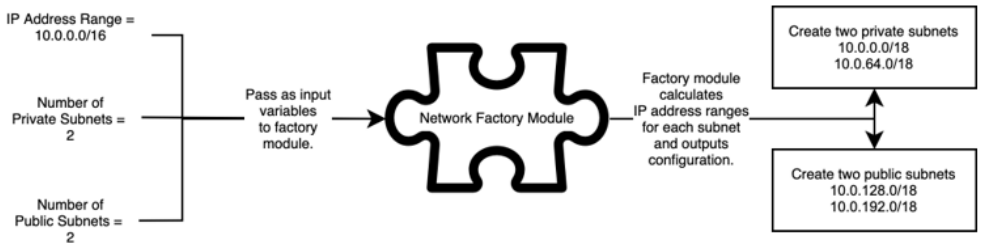

IP Address Range = 10.0.0.0/16
Number of Private Subnets = 2
Number of Public Subnets = 2
Pass as input variables to factory module.
Network Factory Module
Factory module calculates IP address ranges for each subnet and outputs configuration.
Create two private subnets 10.0.0.0/18 10.0.64.0/18
Create two public subnets 10.0.128.0/18 10.0.192.0/18

Figure 3.6. A network factory module can include transformations that calculate subnet addressing and create multiple subnet resources for a network.

Modules using factory patterns can transform input variables into standardized templates, a common practice to generate names or identifiers. When you implement the network module with the factory pattern in code, you add a `SubnetFactory` module. Listing 3.3 builds a factory module to generate names for the subnets.

Listing 3.3 Creating three subnets in GCP using a factory pattern

```
import json
import ipaddress


def _generate_subnet_name(address):    #B
   address_identifier = format(ipaddress.ip_network(    #B
       address).network_address).replace('.', '-')    #B
   return f'network-{address_identifier}'    #B


class SubnetFactory:    #A
   def __init__(self, address, region):
       self.name = _generate_subnet_name(address)    #B
       self.address = address    #C
       self.region = region     #D
       self.network = 'default'    #E
       self.resource = self._build()    #H

   def _build(self):    #H
       return {
           'resource': [
               {
                   'google_compute_subnetwork': [    #F
                       {    #F
                           f'{self.name}': [    #F
                               {    #F
                                   'name': self.name,    #F
                                   'ip_cidr_range': self.address,    #F
                                   'region': self.region,    #F
                                   'network': self.network    #F
                               }    #F
                           ]    #F
                       }    #F
                   ]    #F
               }
           ]
       }


if __name__ == "__main__":
   subnets_and_regions = {    #G
       '10.0.0.0/24': 'us-central1',    #G
       '10.0.1.0/24': 'us-west1',    #G
       '10.0.2.0/24': 'us-east1',    #G
   }    #G

   for address, region in subnets_and_regions.items():    #G

       subnetwork = SubnetFactory(address, region)    #H

       with open(f'{_generate_subnet_name(address)}.tf.json', 'w') as outfile:   #I
           json.dump(subnetwork.resource, outfile, sort_keys=True, indent=4)   #I
```

#A Create a module for the subnet, which uses the factory pattern to generate any number of subnets.
#B For a given subnet, generate the subnet name by dash-delimiting the IP address range and appending it to the "network".
#C Pass the subnet's address to the factory.

#D Pass the subnet’s region to the factory.
#E Create the subnets on the “default” network in this example.
#F Create the Google subnetwork using a Terraform resource based on the name, address, region, and network.
#G For each subnet defined with its IP address range and region, create a subnet using the factory module.
#H Use the module to create the JSON configuration for the network and subnetwork.
#I Write it out to a JSON file to be executed by Terraform later.

Why do I separate the subnets into their own factory module? Creating a separate module for subnets promotes the principle of evolvability! I can change the logic to generate names for any number of subnets. I can also update the name format without affecting the network.

Most factory modules include *transformation or dynamic generation* of attributes. For example, you can modify the network factory module to calculate the IP address ranges for subnets. The calculations automatically build the correct number of private or public subnets.

However, I recommend minimizing the transformations you add to factory modules when possible. They can add complexity to resource configuration. The more complex the logic for your transformation, the more you need tests to check for transformations. I’ll cover how to test modules and infrastructure configuration in Chapter 6.

The factory pattern balances reproducibility and evolvability of infrastructure resources. It manufactures similar infrastructure with minor differences for names, sizes, or other attributes. You’ll want to build a factory module if the configuration applies to commonly built resources, such as networks or servers.

Any time you run a factory module, you can expect to get the specific set of resources you request. The module does not contain much logic to determine *which* resources to build. Instead, a factory module focuses on setting attributes for resources.

I frequently write factory modules with *many default constants* and *few input variables*. This way, I reduce the overhead of maintaining and validating the inputs. Infrastructure that commonly uses factory modules includes networks and subnetworks, clusters of servers, managed databases, managed queues, or managed caches, and more.

3.4 Prototype

You can create the database servers now that you have created a module to build database networks. However, you must tag all resources in the database system with customer name, business unit, and cost center. The auditing team also asks you to include `automated=true` to identify the automated resources.

Ideally, tags (or labels in GCP) must be consistent across all your resources. Your automation should copy them to every resource if you update the tags. You’ll learn more about the importance of tagging in Chapter 8.

What if you could put all the tags in one place and update them at once? Figure 3.7 shows that you can put all of your tags into a module. The database server references the common module for tags and applies the static values to the server.

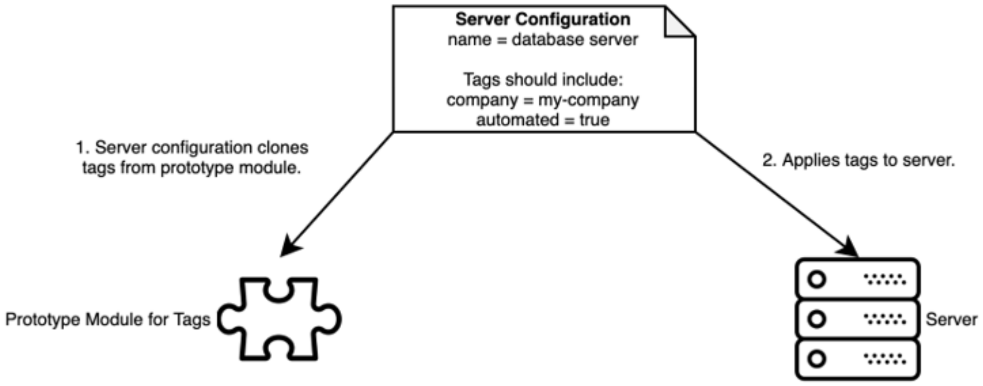

Server Configuration
name = database server
Tags should include:
company = my-company
automated = true
1. Server configuration clones tags from prototype module.
2. Applies tags to server.
Prototype Module for Tags
Server

Figure 3.7. A module with the prototype pattern returns a copy of static values, such as tags, for consumption by other infrastructure resources.

Rather than hard-coding every tag, you created a module implementing the ***prototype*** pattern to express a set of static defaults for consumption by other modules. Prototype modules produce a copy of the configuration to append to other resources.

> **DEFINITION** The prototype pattern takes a set of input variables to build a set of static defaults for consumption by other modules. They usually do not directly create infrastructure resources but export configuration values.

You can think of a prototype like a dictionary that stores words and definitions (figure 3.8). The dictionary's creators change the words and definitions. You can reference it and update your text or vocabulary.

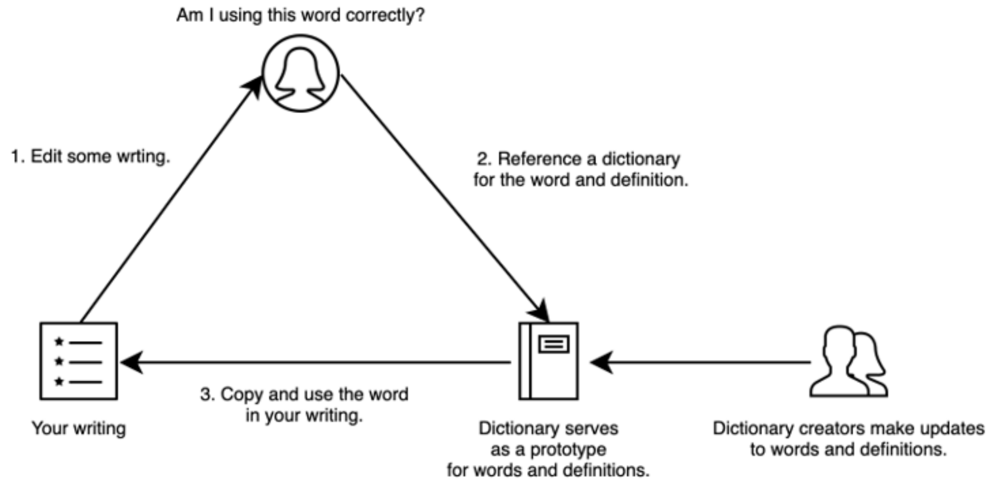

Am I using this word correctly?
1. Edit some wrting.
2. Reference a dictionary for the word and definition.
3. Copy and use the word in your writing.
Your writing
Dictionary serves as a prototype for words and definitions.
Dictionary creators make updates to words and definitions.

Figure 3.8. You use the dictionary as a prototype to reference words and definitions and update them in your writing.

Why use a prototype module to reference common metadata? The prototype pattern promotes the principle of evolvability and reproducibility. It ensures consistent configuration across resources and eases the evolution of common configuration. You don't have to find and replace strings in your files!

Let's implement the tag module with the prototype pattern. I create a module using the prototype pattern that returns a set of standard tags. In subsequent infrastructure resources, I reference the module for `StandardTags` for any tags that I need to include. The module does not create tag resources. Instead, it returns a copy of pre-defined tags.

Listing 3.4 Creating a tagging module using the prototype pattern

```
import json


class StandardTags():    #A
   def __init__(self):    #A
       self.resource = {    #A
           'customer': 'my-company',    #A
           'automated': True,    #A
           'cost_center': 123456,    #A
           'business_unit': 'ecommerce'    #A
       }    #A


class ServerFactory:    #B
   def __init__(self, name, network, zone='us-central1-a', tags={}):    #C
       self.name = name
       self.network = network
```

```
        self.zone = zone
        self.tags = tags    #C
        self.resource = self._build()    #G

    def _build(self):
        return {
            'resource': [
                {
                    'google_compute_instance': [    #D
                        {
                            self.name: [
                                {
                                    'allow_stopping_for_update': True,
                                    'boot_disk': [
                                        {
                                            'initialize_params': [
                                                {
                                                    'image': 'ubuntu-1804-lts'
                                                }
                                            ]
                                        }
                                    ],
                                    'machine_type': 'f1-micro',
                                    'name': self.name,
                                    'network_interface': [
                                        {
                                            'network': self.network
                                        }
                                    ],
                                    'zone': self.zone,
                                    'labels': self.tags    #F
                                }
                            ]
                        }
                    ]
                }
            ]
        }


if __name__ == "__main__":
    config = ServerFactory(    #G
        name='database-server', network='default',    #G
        tags=StandardTags().resource)    #H

    with open('main.tf.json', 'w') as outfile:     #I
        json.dump(config.resource, outfile,     #I
                  sort_keys=True, indent=4)     #I
```

#A Create a module using the prototype pattern that returns a copy of standard tags, such as customer, cost center, and business unit.
#B Create a module using the factory pattern to create a Google computer instance (server) based on name, network, and tags.
#C Pass tags as a variable to the server module.
#D Create the Google compute instance (server) using a Terraform resource.
#F Add the tags stored in the variable to the Google compute instance resource.
#G Use the module to create the JSON configuration for a server on the “default” network.
#H Use the standard tags module to add tags to the server.

#I Write it out to a JSON file to be executed by Terraform later.

Let’s run the Python script to create the server configuration. When you examine the JSON output for the server, you’ll notice that the server includes a set of labels. Those labels match the standard tags from your prototype module!

Listing 3.5 Server configuration with tags from the module

```
{
   "resource": [
       {
           "google_compute_instance": [    #A
               {
                   "database-server": [    #B
                       {
                           "allow_stopping_for_update": true,
                           "boot_disk": [
                               {
                                   "initialize_params": [
                                       {
                                           "image": "ubuntu-1804-lts"
                                       }
                                   ]
                               }
                           ],
                           "labels": {     #C
                               "automated": true,     #C
                               "business_unit": "ecommerce",     #C
                               "cost_center": 123456,     #C
                               "customer": "my-company"     #C
                           },     #A
                           "machine_type": "f1-micro",
                           "name": "database-server",    #B
                           "network_interface": [
                               {
                                   "network": "default"
                               }
                           ],
                           "zone": "us-central1-a"    #D
                       }
                   ]
               }
           ]
       }
   ]
}
```

#A The JSON file defines a Google compute instance using a Terraform resource.
#B Terraform identifies the resource as a database server. The JSON configuration matches what you defined in the server factory module using Python.
#C Add tags from the standard tags prototype module to the labels field in the server configuration.
#D The JSON configuration retrieves the zone variable and populates it into the JSON file.

You’ll notice that your server configuration contains a lot of hard-coded values, like operating system and machine type. The values behave as global defaults. Over time, you’ll keep adding global defaults to your factory module and find they overrun the module!

To detangle and organize the global defaults, you can define them in a prototype module. The module makes it easier to evolve the default over time and compose it with other values. The prototype becomes the static, well-defined default for a resource.

In one such situation, I started by writing a factory module to create a set of alerts on infrastructure. Initially, I passed the environment names and metrics thresholds to parametrize the alerts and their configuration. I discovered alerts did not need the environment names, and the metrics thresholds did not change across environments.

As a result, I converted this module into a prototype. Teams that needed to add metrics into their systems imported the module. The module added pre-defined alert resources to their configuration.

Domain-specific languages

Domain-specific languages for tools like HashiCorp Terraform, Kubernetes, AWS CloudFormation, or Azure Bicep do not have "global constants" like programming languages. However, they do support module referencing and object structures. You can use the same pattern for DSLs and programming languages by creating the prototype as an object.

A prototype makes creating a standard set of resources or configurations easier. It eliminates the uncertainty in setting input values. However, you will have exceptions to standard values. As a solution, you can override or add configurations based on the resource. For example, I usually merge custom tags unique to a resource with the list of standard tags.

Besides tags, I commonly use prototype modules for regions, availability zones, or account identifiers. I create modules with the prototype pattern when I have *static configurations with many global defaults or complex transformations*. For example, you might have a server initialization script that runs when you use SSL. You can create a prototype module to template the script based on whether or not you use SSL.

3.5 Builder

You learned to apply the singleton pattern to create a project, the factory to create the networks, and the prototype to set the tags for the database server. Next, you'll build a load balancer that connects you to the database.

You run into a challenging requirement. The module must allow you to create private or public load balancers! A private load balancer requires different server and network configuration. You must build a module that offers the flexibility to choose a private or public load balancer and configures the server and networks based on your choice.

Figure 3.8 demonstrates a module that chooses a firewall and server configuration based on your load balancer type. I can use the same module to create an external or internal load balancer. The module handles the correct configuration for the load balancer and its required firewall rules.

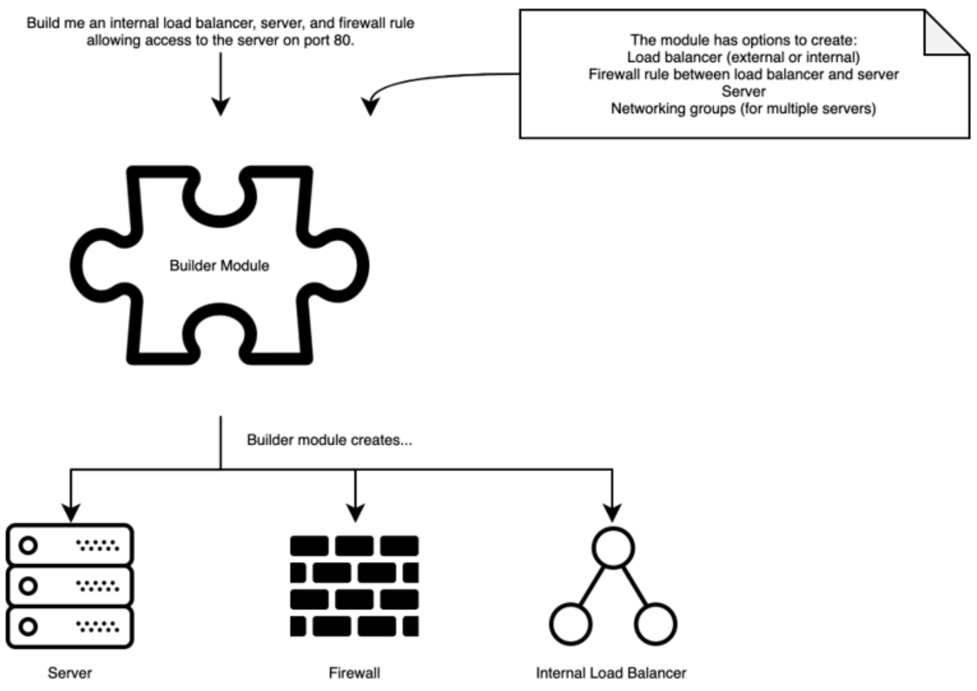

Build me an internal load balancer, server, and firewall rule allowing access to the server on port 80.
The module has options to create:
Load balancer (external or internal)
Firewall rule between load balancer and server
Server
Networking groups (for multiple servers)
Builder Module
Builder module creates...
Server
Firewall
Internal Load Balancer

Figure 3.9. A builder module for the database would include parameters to select the type of load balancer and firewall rules it must create.

The module gives you the choice to build the system you want, helping with evolvability and composability. The module follows the ***builder pattern***, which captures some default values but allows you to compose the system you need. The builder pattern organizes a set of related resources you enable or disable for your desired system.

> **DEFINITION** **The builder pattern assembles a set of infrastructure resources that you can enable or disable to achieve your desired configuration.**

Implementing the builder pattern in your database module allows you to generate a combination of resources based on your selection.

A builder pattern uses inputs to decide *which* resources it needs to build, while a factory module configures resource attributes based on input variables. The pattern works like building houses for a real estate development. You choose from preset blueprints and tell the builder which changes you want for the layout (figure 3.10). For example, some builders might add an extra room by removing the garage.

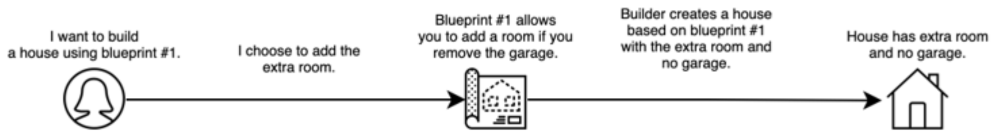

I want to build a house using blueprint #1.
I choose to add the extra room.
Blueprint #1 allows you to add a room if you remove the garage.
Builder creates a house based on blueprint #1 with the extra room and no garage.
House has extra room and no garage.

Figure 3.10. A builder module constructs a house with a preset blueprint that allows for some layout changes, such as an extra room.

Let’s start implementing the builder pattern. First, I define a load balancer module using the factory pattern. I use the factory pattern to customize the load balancer (also known as compute forwarding rule in GCP). The module sets the scheme for the load balancer as external or internal.

Listing 3.6 Factory module for load balancer

```
class LoadBalancerFactory:    #A
   def __init__(self, name, region='us-central1', external=False):    #A
       self.name = name
       self.region = region
       self.external = external    #A
       self.resources = self._build()    #F

   def _build(self):
       scheme = 'EXTERNAL' if self.external else 'INTERNAL'    #B
       resources = []
       resources.append({
           'google_compute_forwarding_rule': [{    #C
               'db': [
                   {
                       'name': self.name,
                       'target': r'${google_compute_target_pool.db.id}',    #D
                       'port_range': '3306',     #E
                       'region': self.region,
                       'load_balancing_scheme': scheme,
                       'network_tier': 'STANDARD'
                   }
               ]
           }
           ]
       })
       return resources
```

#A Create a module for the load balancer, which uses the factory pattern to generate an internal or external load balancer.
#B Set scheme to internal or external load balancing. The load balancer defaults to an internal configuration.
#C Create the Google compute forwarding rule using a Terraform resource. This is GCP’s equivalent of load balancing.
#D Set the load balancer’s target to the database server group. This uses Terraform’s built-in variable interpolation feature to dynamically resolve the database server group’s ID.
#E Allow traffic to port 3306, the MySQL database port.
#F Use the module to create the JSON configuration for the load balancer.

AWS and Azure equivalent

You can equate a GCP compute forwarding rule to an AWS Elastic Load Balancer or an Azure Load Balancer. Similarly, an AWS security group or Azure network security group roughly equals a GCP firewall rule. For a complete example, refer to the code repository at https://github.com/joatmon08/manning-book.

However, an external load balancer requires additional configuration in the form of a firewall rule. You must allow traffic from external sources to the database ports. Let's define a module using the factory pattern for another firewall rule to allow traffic from external sources.

Listing 3.7 Factory module for the firewall rule

```
class FirewallFactory:    #A
   def __init__(self, name, network='default'):
       self.name = name
       self.network = network
       self.resources = self._build()    #D

   def _build(self):
       resources = []
       resources.append({
           'google_compute_firewall': [{    #B
               'db': [
                   {
                       'allow': [    #C
                           {    #C
                               'protocol': 'tcp',    #C
                               'ports': ['3306']    #C
                           }    #C
                       ],    #C
                       'name': self.name,
                       'network': self.network
                   }
               ]
           }
           ]
       })
       return resources
```

#A Create a module for the firewall, which uses the factory pattern to generate a firewall rule.
#B Create the Google compute firewall using a Terraform resource. This is GCP's equivalent of a firewall rule.
#C Firewall rule should allow TCP traffic to port 3306 by default.
#D Use the module to create the JSON configuration for the load balancer.

Thanks to the principle of composability, I put the load balancer and factory module into the database builder module. The module needs a variable that helps you choose the type of load balancer and whether or not you should include a firewall rule to allow traffic to the load balancer.

When you implement the database builder module, you set it to create the database server group and networks by default. Then, the builder accepts two options: an internal or external load balancer and the extra firewall rule.

Listing 3.8 Constructing a database with the builder pattern

```
import json
from server import DatabaseServerFactory    #A
from loadbalancer import LoadBalancerFactory    #A
from firewall import FirewallFactory    #A


class DatabaseModule:     #B
   def __init__(self, name):
       self._resources = []
       self._name = name
       self._resources = DatabaseServerFactory(self._name).resources    #C

   def add_internal_load_balancer(self):     #D
       self._resources.extend(
           LoadBalancerFactory(
               self._name, external=False).resources)

   def add_external_load_balancer(self):     #D
       self._resources.extend(
           LoadBalancerFactory(
               self._name, external=True).resources)

   def add_google_firewall_rule(self):     #F
       self._resources.extend(
           FirewallFactory(
               self._name).resources)

   def build(self):     #G
       return {     #G
           'resource': self._resources     #G
       }     #G


if __name__ == "__main__":
   database_module = DatabaseModule('development-database')    #H
   database_module.add_external_load_balancer()    #H
   database_module.add_google_firewall_rule()    #H

   with open('main.tf.json', 'w') as outfile:   #I
       json.dump(database_module.build(), outfile,   #I
                 sort_keys=True, indent=4)   #I
```

#A Import factory modules to create the database server group, load balancer, and firewall.
#B Create a module for the database, which uses the builder pattern to generate the required database server group, networks, load balancers, and firewalls.
#C Always create a database server group and networks using the factory module. The builder module needs the database server group.
#D Add a method so you can choose to build an internal load balancer.
#E Add a method so you can choose to build an external load balancer.
#F Add a method so you can choose to build a firewall rule to allow traffic to the databases.
#G Use the builder module to return the JSON configuration for your customized database resources.
#H Use the database builder module to create a database server group with external access (load balancer and firewall rule).
#I Write it out to a JSON file to be executed by Terraform later.

After running the Python script, you will find a lengthy JSON configuration with the instance templates, server group, server group manager, external load balancer, and firewall rule. The builder generates all of the resources you need to build an externally accessible database. Note that the listing omits the other components for clarity.

Listing 3.9 Truncated database system configuration

```
[
  {
    "google_compute_forwarding_rule": [    #A
      {
        "db": [
          {
            "load_balancing_scheme": "EXTERNAL",    #B
            "name": "development-database",
            "network_tier": "STANDARD",
            "port_range": "3306",    #D
            "region": "us-central1",
            "target": "${google_compute_target_pool.db.id}"    #C
          }
        ]
      }
    ]
  },
  {
    "google_compute_firewall": [    #A
      {
        "db": [
          {
            "allow": [    #D
              {    #D
                "ports": [    #D
                  "3306"    #D
                ],    #D
                "protocol": "tcp"    #D
              }    #D
            ],
            "name": "development-database",
            "network": "default"
          }
        ]
      }
    ]
  }
]
```

#A The JSON file defines a Google compute forwarding rule and firewall using a Terraform resource. It omits the instance templates, server group, and server group for clarity.
#B Create a load balancer with the “EXTERNAL” scheme, which makes it accessible from external sources.
#C Set the load balancer’s target to the database server group. This uses Terraform’s built-in variable interpolation feature to dynamically resolve the database server group’s ID.
#D Create a firewall that allows TCP traffic on port 3306, the MySQL database port.

The builder pattern helps you adhere to the principle of evolvability. You get to choose the set of resources you need. A module with this pattern takes away the challenge of configuring the right combination of attributes and resources.

Furthermore, you can use the builder pattern to wrap a generic interface around a cloud provider's resources. The Python example offers builder methods to "add_external_load_balancer," which wraps around the GCP compute forwarding rule. When you use the module, the option describes the intent of creating a generic "load balancer" and not a GCP forwarding rule.

Domain-specific languages

Some domain-specific languages (DSLs) offer an if-else (conditional) statement or loop (iteration) that you can use for the builder pattern. HashiCorp Terraform offers the "count" argument to create a set number of resources based on a conditional statement. AWS CloudFormation supports conditionals for user inputs that can select stacks. Azure Bicep uses deploy conditions. For Ansible, you can use conditional imports to select tasks or playbooks.

For example, you could set up a boolean variable called "add_external_load_balancer." If you pass "true" to the variable, the DSL adds a conditional statement to build an external load balancer resource. Otherwise, it creates an internal load balancer.

Some DSLs do not offer conditional statements. You will need some code similar to my Python example to template the DSL. For example, you can use Helm to template and release Kubernetes YAML files.

The builder pattern best applies to modules that create multiple resources. These use cases include configuration for container orchestrators such as Kubernetes, platforms with cluster architectures, dashboards for application and system metrics, and more. A builder module for these use cases allows me to select the resource I want without passing specific input attributes.

However, builder modules can be complex because they reference other modules and multiple resources. The risk of module misconfiguration can be very high. Chapter 6 will cover testing strategies to ensure a builder module's functionality and stability.

3.6 Choosing a Pattern

Throughout the chapter, I showed how to group some resources for a database system into different module patterns throughout the chapter. How would you choose which module pattern to use? What about the other resources in the database system I did not mention?

You can create separate modules for new infrastructure resources with different business functions and purposes. In the database example throughout this chapter, I separated the configuration of the Google project (singleton), network (factory), and database cluster (factory) into modules. They each evolve as different resources with different input variables and defaults.

My example applies to the composite pattern to combine all of the different module patterns in the system. I choose the factory pattern for network, load balancer, and database cluster modules because I want to pass attributes and customize each resource. Tags often use the prototype pattern because they involve consistent metadata copied to other resources. I use the factory and prototype patterns for most of the modules I write because they offer composability, reproducibility, and evolvability.

By contrast, I build the Google project as a singleton because no one else will change attributes for the single instance of my project. The project does not change much so I use a less complex pattern. However, I solved the complex problem of creating a database system with the builder pattern. The builder module allows me to select the specific resources to create.

Figure 3.11 offers a decision tree for identifying which pattern to use. I ask myself a series of questions on the purpose, reuse and update frequency, and composition of multiple resources. Based on these criteria, I create a module with a specific pattern.

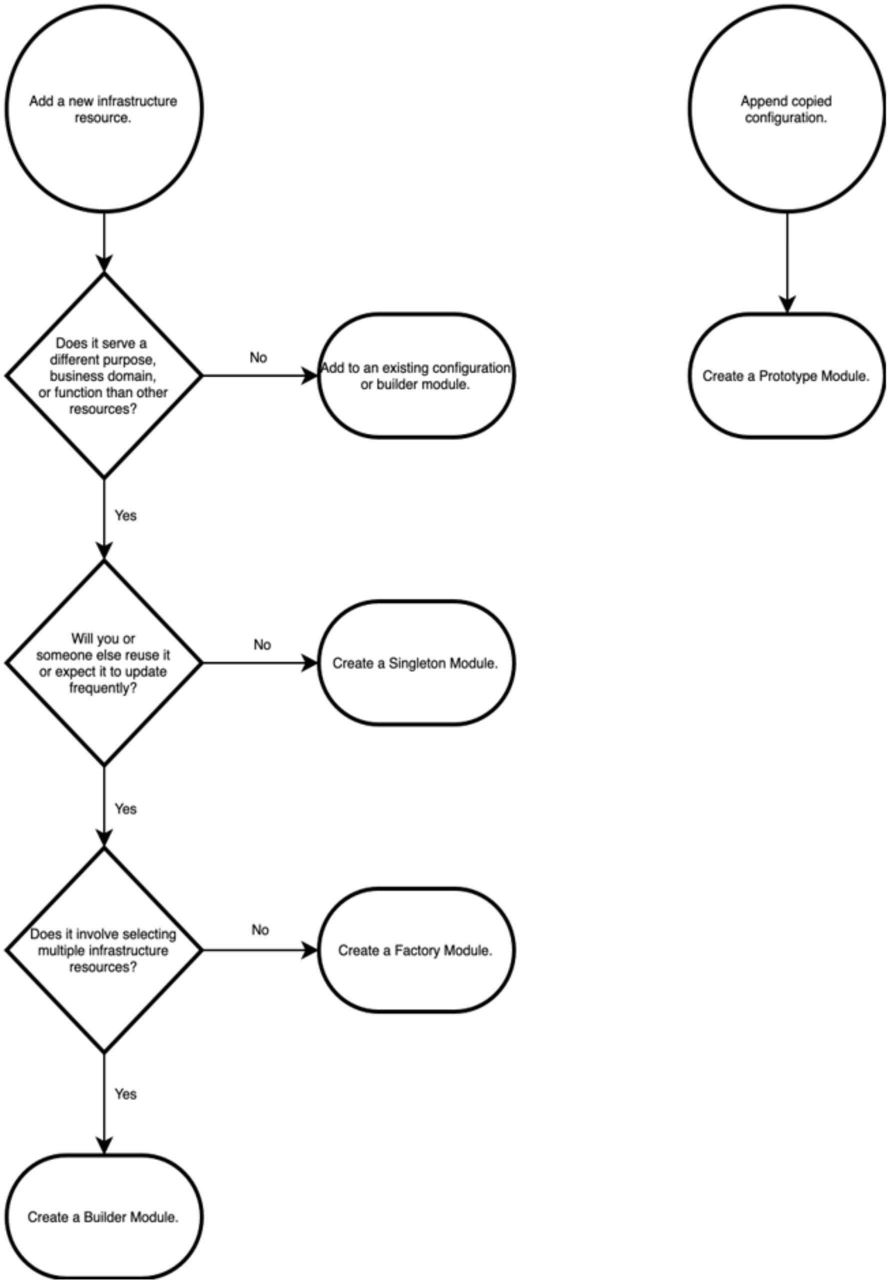

Add a new infrastructure resource.
Does it serve a different purpose, business domain, or function than other resources?
No
Add to an existing configuration or builder module.
Yes
Will you or someone else reuse it or expect it to update frequently?
No
Create a Singleton Module.
Yes
Does it involve selecting multiple infrastructure resources?
No
Create a Factory Module.
Yes
Create a Builder Module.
Append copied configuration.
Create a Prototype Module.

Figure 3.11. To decide which module pattern you want to use, you must assess how you use the resource and its behavior.

Following the decision tree helps build more composable and evolvable modules. You want to balance the predictability of standard attributes with the flexibility to override configuration for specific resources. However, keep an open mind. Modules can outgrow and change in function. Just because you build a module with one pattern doesn't mean you convert it to a different one in the future!

Exercise 3.1

Given:

```
if __name__ == "__main__":
  environment = 'development'
  name = f'{environment}-hello-world'
  cidr_block = '10.0.0.0/16'

  # NetworkModule returns a subnet and network
  network = NetworkModule(name, cidr_block)

  # Tags returns a default list of tags
  tags = TagsModule()

  # ServerModule returns a single server
  server = ServerModule(name, network, tags)
```

Which of the following module patterns does the infrastructure as code apply? (Choose all that apply.)

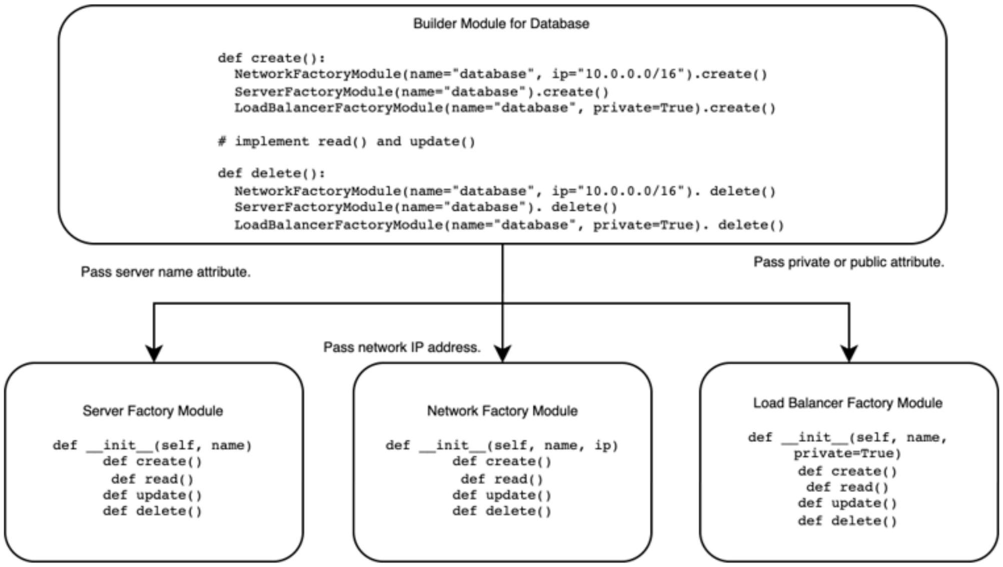
A. Factory
B. Singleton
C. Prototype
D. Builder
E. Composite

Find answers and explanations at the end of the chapter.

Note that many of the patterns in this chapter focus on building modules with IaC tooling. Sometimes, you might write automation in a programming language because you cannot find infrastructure as code support. This situation happens most with legacy infrastructure! For example, imagine you need to create the database system in GCP. However, you do not have an IaC tool and can only access the GCP API directly.

To create the database system using the GCP API, you separate each infrastructure resource into a factory module with four functions: create, read, update, and delete. Changes to the resource use the composition of these functions. You can check for errors in each function depending on the action you want to execute for each resource.

Figure 3.12 implements factory modules for the server, network, and load balancer. You can create, read, update, and delete each module. A builder module for the database uses the composite pattern to create, read, update, and delete the network, server, and load balancer.


Builder Module for Database
Pass server name attribute.
Pass private or public attribute.
Pass network IP address.
Server Factory Module
Network Factory Module
Load Balancer Factory Module

```
def create():
  NetworkFactoryModule(name="database", ip="10.0.0.0/16").create()
  ServerFactoryModule(name="database").create()
  LoadBalancerFactoryModule(name="database", private=True).create()

# implement read() and update()

def delete():
  NetworkFactoryModule(name="database", ip="10.0.0.0/16"). delete()
  ServerFactoryModule(name="database"). delete()
  LoadBalancerFactoryModule(name="database", private=True). delete()
```

```
def __init__(self, name)
    def create()
    def read()
    def update()
    def delete()
```

```
def __init__(self, name, ip)
    def create()
    def read()
    def update()
    def delete()
```

```
def __init__(self, name,
    private=True)
    def create()
    def read()
    def update()
    def delete()
```

Figure 3.12. If you need to write scripts for automation, create factory modules for individual resources and build functions to create, read, update, and delete resources.

Breaking down the updates you'll make to resources into four functions organizes the automation. Even the builder pattern uses the create, read, update, and delete functions. The functions define the automation behavior you want to use to configure resources. However, you want to test each function for the principle of idempotency. Any time you run the function, it should result in the same configuration.

You can apply the module patterns in this chapter to automating and implementing infrastructure as code on any infrastructure. As you develop infrastructure as code, identify where you can break down your infrastructure system into modules. When deciding when and what to modularize, consider the following:

- Is the resource shared?
- What business domain does it serve?
- Which environment uses the resource?
- Which team manages the infrastructure?
- Does the resource use a different tool?
- How do you change a resource without affecting something else in the module?

By assessing which resources align with different business units, teams, or functions, you build smaller sets of infrastructure. As a general practice, write modules with as few resources as possible. Modules with fewer resources speed up the provisioning and minimize the blast radius of a failure. More importantly, you can deploy, test, and debug smaller modules before applying them to a more extensive system.

Grouping resources into modules offers a few benefits for you, your team, and your company. For you, modules improve the scalability and resiliency of infrastructure resources. You minimize the blast radius of changes to modules to improve the resiliency of the overall system.

For your team, a module provides the benefit of a self-service mechanism for other team members to reproduce your module and create infrastructure. Your teammate can use the module and pass the variables they want to customize instead of finding and replacing attributes. You'll learn more about sharing modules in chapter 5.

For your organization, a module helps you standardize better infrastructure and security practices across resources. You can use the same configurations to stamp out similar load balancers and restricted firewall rules. Modules also help your security team audit and enforce these practices across different teams, covered in chapter 8.

3.7 Exercises and Solutions

Exercise 3.1

Given:

```
if __name__ == "__main__":
 environment = 'development'
 name = f'{environment}-hello-world'
 cidr_block = '10.0.0.0/16'

 # NetworkModule returns a subnet and network
 network = NetworkModule(name, cidr_block)

 # Tags returns a default list of tags
 tags = TagsModule()

 # ServerModule returns a single server
 server = ServerModule(name, network, tags)
```

Which of the following module patterns does the infrastructure as code apply? (Choose all that apply.)

A. Factory
B. Singleton
C. Prototype
D. Builder
E. Composite

Answer:

The infrastructure as code applies A, C, and E. The network module uses the composite pattern to compose a subnet and a network. The tag module uses the prototype pattern to return status metadata. The server module uses the factory pattern to return a single server based on name, network, and tags.

The code does not use singleton or builder patterns. It does not create singular global resources or internal logic to build specific resources.

3.8 Summary

- Apply module patterns such as singleton, factory, prototype, and builder so you can construct composable infrastructure configurations.
- Use the composite pattern to group infrastructure resources into a hierarchy and structure them for automation.
- You can use the singleton pattern to manage single instances of infrastructure, which applies to rarely changed infrastructure resources.
- Use the prototype pattern for copying and applying global configuration parameters, such as tags or common configuration.
- Factory modules take inputs to construct infrastructure resources with a specific configuration.
- Builder modules take inputs to decide which resources to create. Builder modules can be composed of factory modules.
- When deciding what and how to modularize, assess which functions or business domain the infrastructure configuration serves.
- If you write scripts to automate infrastructure, construct factory modules with capabilities to create, read, update, and delete. Reference them in builder modules.

4 Patterns for infrastructure dependencies

This chapter covers

- **Writing loosely coupled infrastructure modules using dependency patterns**
- **Identifying ways to decouple infrastructure dependencies**
- **Recognizing infrastructure use cases for dependency patterns**

An infrastructure system involves a set of resources that depend on each other. For example, a server depends on the existence of a network. How do you know the network exists before creating a server? You can express this with an ***infrastructure dependency***. An infrastructure dependency happens when another resource requires another one to exist before creating or modifying it.

> **DEFINITION** **An infrastructure dependency expresses a relationship where an infrastructure resource depends on the existence and attributes of another resource.**

Usually, you identify the server's dependency on the network by hard-coding the network identifier. However, hard-coding more tightly binds the dependency between server and network. Any time you change the network, you must update the hard-coded dependency.

In Chapter 2, you learned how to avoid hard-coding values with variables to promote reproducibility and evolvability. Passing the network identifier as a variable better decouples the server and network. However, a variable only works between resources in the same module. How can you express dependencies *between* modules?

The last chapter grouped resources into modules to enhance composability. This chapter covers patterns for managing infrastructure dependencies to enhance evolvability (change). You can more easily replace one module with another when they have a loose dependency.

In reality, infrastructure systems can be pretty, complex and you can’t swap modules without some disruption. Loosely coupled dependencies offer mitigation for change failure but don’t guarantee 100% availability!

4.1 Unidirectional relationships

Different dependency relationships affect infrastructure change. Imagine you add a firewall rule each time you create a new application. The firewall rule has a ***unidirectional dependency*** on the application IP address to allow traffic. Any change to the application gets reflected in the firewall rule.

> **DEFINITION** A unidirectional dependency expresses a one-way relationship where only one resource refers to another.

You can express unidirectional dependencies between any set of resources or modules. Figure 4.1 describes the unidirectional relationship between the firewall rule and the application. The rule *depends on* the application, which makes it higher up the infrastructure stack than the lower-level application.

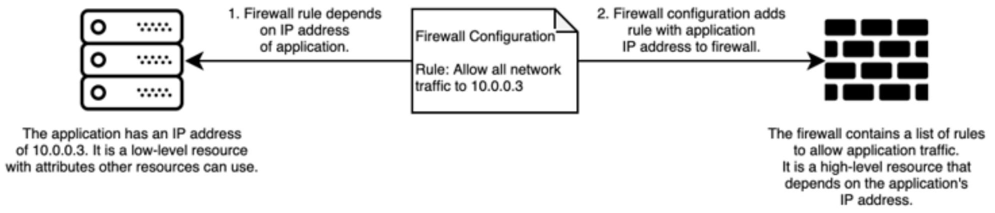

1. Firewall rule depends on IP address of application.
Firewall Configuration
Rule: Allow all network traffic to 10.0.0.3
2. Firewall configuration adds rule with application IP address to firewall.
The application has an IP address of 10.0.0.3. It is a low-level resource with attributes other resources can use.
The firewall contains a list of rules to allow application traffic. It is a high-level resource that depends on the application's IP address.

Figure 4.1. The firewall rule unidirectionally depends on the application’s IP address.

When you express a dependency, you have a ***high-level*** resource like the firewall that depends on the existence of a ***low-level*** resource like the application.

> **DEFINITION** A high-level resource depends on another resource or module. A low-level resource has high-level resources depending on it.

Let’s say a reporting application needs a list of rules for the firewall. It sends the rules to an audit application. However, the firewall needs to know the IP address of the reporting application. Should you update the IP address of the reporting application or the firewall rule first? Figure 4.2 shows the conundrum of deciding which application you should update first.

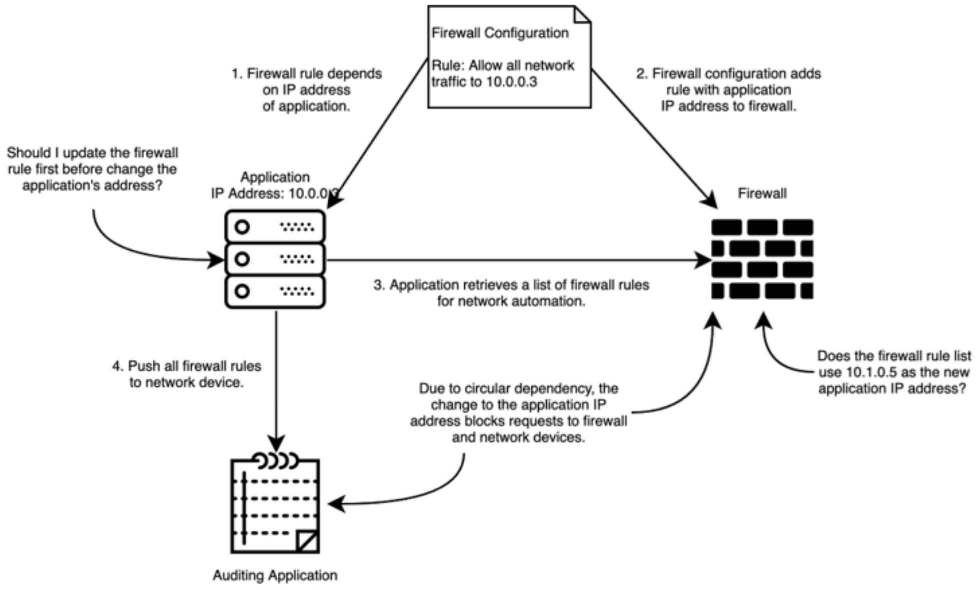

Firewall Configuration
Rule: Allow all network traffic to 10.0.0.3
1. Firewall rule depends on IP address of application.
2. Firewall configuration adds rule with application IP address to firewall.
Should I update the firewall rule first before change the application's address?
Application IP Address: 10.0.0.3
Firewall
3. Application retrieves a list of firewall rules for network automation.
4. Push all firewall rules to network device.
Due to circular dependency, the change to the application IP address blocks requests to firewall and network devices.
Does the firewall rule list use 10.1.0.5 as the new application IP address?
Auditing Application

Figure 4.2. The reporting application and the firewall have a circular dependency on each other. Changes block connectivity to the application.

Your example encountered a circular dependency, which introduces a chicken-or-egg problem. You cannot change one resource without affecting the other. If you first change the address of the reporting application, the firewall rule must change. However, the reporting application fails because it can't connect. You might have blocked its request!

Circular dependencies cause unexpected behaviors during changes, which ultimately affect composability and evolvability. You don't know which resource to update first. By contrast, you can identify how a low-level module's change might affect the high-level one. Unidirectional dependency relationships make changes more predictable. After all, a successful infrastructure change depends on two factors: predictability and isolation.

4.2 Dependency injection

Unidirectional dependencies help you engineer ways to minimize the impact of low-level module changes to high-level modules. For example, network changes should not disrupt high-level resources like queues, applications, or databases. This section applies the software development concept of dependency injection to infrastructure and further decouples unidirectional dependencies. Dependency injection involves two principles: inversion of control and dependency inversion.

4.2.1 Inversion of Control

When you enforce unidirectional relationships in your infrastructure dependencies, your high-level resource gets information about the low-level resource. Then it can run its changes. For example, a server gets information about the network's ID and IP address range before it claims an IP address (figure 4.3).

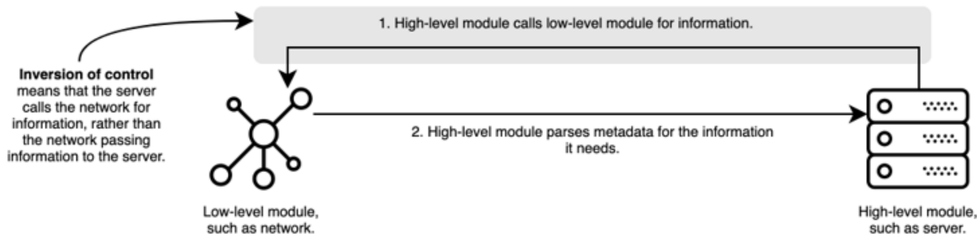

1. High-level module calls low-level module for information.
Inversion of control means that the server calls the network for information, rather than the network passing information to the server.
2. High-level module parses metadata for the information it needs.
Low-level module, such as network.
High-level module, such as server.

Figure 4.3. With inversion of control, the high-level resource or module calls the low-level module for information and parses its metadata for any dependencies.

The server calls the network, naturally applying a software development principle called ***inversion of control***. The high-level resource calls for information about the low-level resource before updating.

> **DEFINITION** Inversion of control is the principle in which the high-level resource calls the low-level one for attributes or references.

As a non-technical example, you use inversion of control when you call to schedule a doctor's appointment instead of the doctor's office automatically scheduling your appointment.

Let's apply inversion of control to implement the server's dependency on the network. You create the network using a network module. The network module outputs a network name and saves it in a file called "terraform.tfstate." High-level resources, like the server, can parse the network name from this JSON file.

Listing 4.1 Network module outputs in JSON file

```
{
 "outputs": {     #A
   "name": {     #B
     "value": "hello-world-subnet",    #B
     "type": "string"    #B
   }
 }    #C
}
```

#A Creating the network with Terraform generates a JSON file with a list of outputs. Terraform uses this file to track the resources it creates.

#B The network module outputs the subnet name as a string.
#C The remainder of the JSON file has been omitted for clarity.

Using inversion of control, the server *calls* the network's "terraform.tfstate" file and reads the subnet name. Since the module expresses outputs in the JSON file, your server module needs to parse for the the value of the subnet name ("hello-world-subnet").

Listing 4.2 Applying inversion of control to create a server on a network

```
import json

class NetworkModuleOutput:    #A
   def __init__(self):
       with open('network/terraform.tfstate', 'r') as network_state:
           network_attributes = json.load(network_state)
       self.name = network_attributes['outputs']['name']['value']    #B

class ServerFactoryModule:    #C
   def __init__(self, name, zone='us-central1-a'):    #D
       self._name = name
       self._network = NetworkModuleOutput()    #E
       self._zone = zone
       self.resources = self._build()    #G

   def _build(self):    #G
       return {
           'resource': [{
               'google_compute_instance': [{    #D
                   self._name: [{
                       'allow_stopping_for_update': True,
                       'boot_disk': [{
                           'initialize_params': [{
                               'image': 'ubuntu-1804-lts'
                           }]
                       }],
                       'machine_type': 'f1-micro',
                       'name': self._name,    #D
                       'zone': self._zone,    #D
                       'network_interface': [{
                           'subnetwork': self._network.name    #F
                       }]
                   }]
               }]
           }]
       }

if __name__ == "__main__":    #H
   server = ServerFactoryModule(name='hello-world')    #H
   with open('main.tf.json', 'w') as outfile:    #H
       json.dump(server.resources, outfile, sort_keys=True, indent=4)    #G
```

#A Create an object that captures the schema of the network module's output. This makes it easier for the server to retrieve the subnet name.
#B The object for the network output parses the value of the subnet name from the JSON object.
#C Create a module for the server, which uses the factory pattern
#D Create the Google compute instance using a Terraform resource with a name and zone.

#E The server module calls the network output object, which contains the subnet name parsed from the network module's JSON file.
#F The server module references the network output's name and passes it to the "subnetwork" field.
#G Use the module to create the JSON configuration for the server using the subnetwork name.
#H Write it out to a JSON file to be executed by Terraform later.

Implementing inversion of control eliminates a direct reference to the subnet in my server module. I can also control and limit the information the network returns for high-level resources to use. More importantly, I improve my composability because I can create other servers and high-level resources on the subnet name offered by the network module.

What if other high-level resources need other low-level attributes? For example, I might create a queue that needs the subnet IP address range. To solve this problem, I evolve the network module to output the subnet IP address range. The queue can reference the outputs for the address it needs.

Inversion of control improves my evolvability as high-level resources require different attributes. I can evolve low-level resources without rewriting the infrastructure as code for high-level resources. However, you need a way to protect the high-level resources from any attribute updates or renaming on the low-level resources.

4.2.2 Dependency inversion

While inversion of control enables the evolution of high-level modules, it does not protect them from changes to low-level modules. Let's imagine I change the network name to its ID. The next time I deploy changes to my server module, it breaks! The server module does not recognize the network ID.

To protect my server module from changes to the network outputs, I add a layer of abstraction between the network output and server. In figure 4.4, the server accesses the network's attributes through an API or some stored configuration instead of the network output. All of these interfaces serve as abstractions to retrieve network metadata.

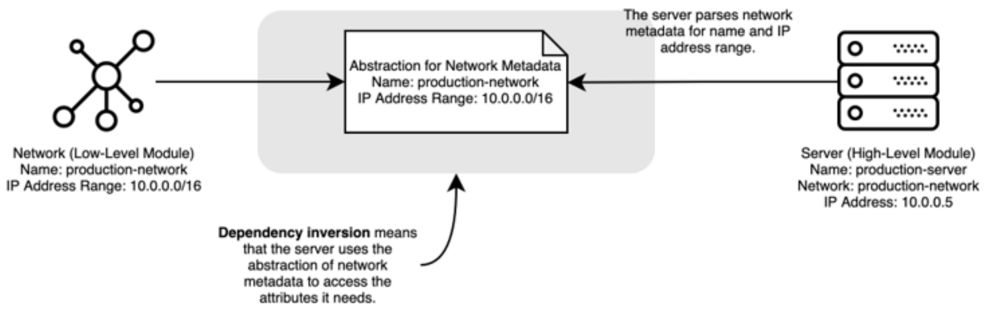

The server parses network metadata for name and IP address range.
Abstraction for Network Metadata
Name: production-network
IP Address Range: 10.0.0.0/16
Network (Low-Level Module)
Name: production-network
IP Address Range: 10.0.0.0/16
Server (High-Level Module)
Name: production-server
Network: production-network
IP Address: 10.0.0.5
Dependency inversion means that the server uses the abstraction of network metadata to access the attributes it needs.

Figure 4.4. Dependency inversion returns an abstraction of the low-level resource metadata to the resource that depends on it.

You can use the principle of dependency inversion to isolate changes to low-level modules and mitigate disruption to their dependencies. ***Dependency inversion*** dictates that high-level and low-level resources should have dependencies expressed through abstractions.

> **DEFINITION** Dependency inversion is the principle of expressing dependencies between high-level and low-level modules or resources through abstractions.

The abstraction layer behaves as a translator that communicates the required attributes. It serves as a buffer for changes to the low-level module away from the high-level one. In general, you can choose from three types of abstraction:

1. Interpolation of resource attributes (within modules)
2. Module outputs (between modules)
3. Infrastructure state (between modules)

Some abstractions like attribute interpolation or module outputs depend on your tool. Abstraction by infrastructure state will depend on your tool or infrastructure API. Figure 4.5 shows the abstractions by attribute interpolation, module output, or infrastructure state to pass network metadata to the server.

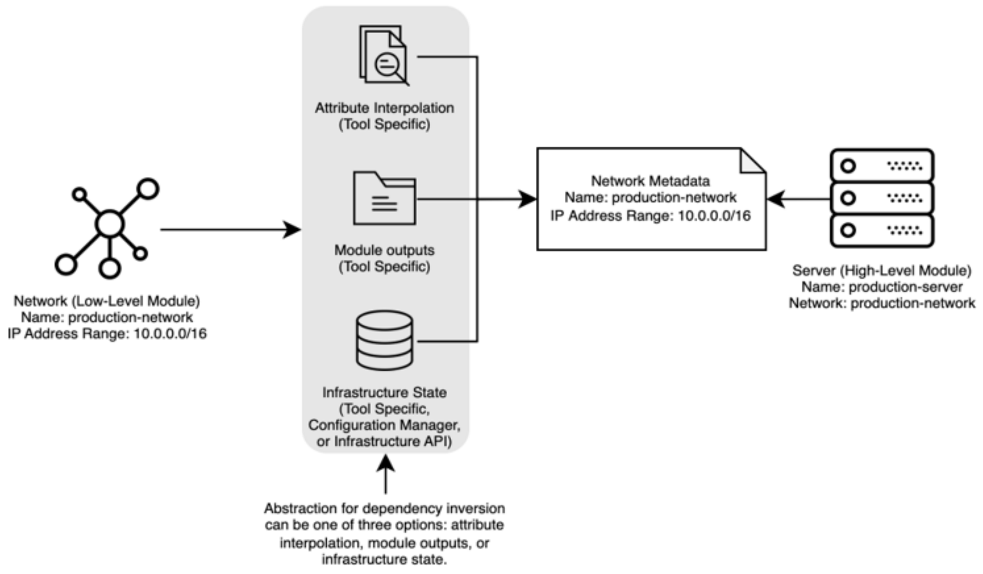

Attribute Interpolation (Tool Specific)
Module outputs (Tool Specific)
Infrastructure State (Tool Specific, Configuration Manager, or Infrastructure API)
Network (Low-Level Module) Name: production-network IP Address Range: 10.0.0.0/16
Network Metadata Name: production-network IP Address Range: 10.0.0.0/16
Server (High-Level Module) Name: production-server Network: production-network
Abstraction for dependency inversion can be one of three options: attribute interpolation, module outputs, or infrastructure state.

Figure 4.5. Depending on the tool and dependencies, abstractions for dependency inversion can use attribute interpolation, module outputs, or infrastructure state.

Let's examine how to implement the three types of abstraction by building modules for the network and server. I'll start with attribute interpolation. Attribute interpolation handles attribute passing between resources or tasks within a module or configuration. Using Python, a subnet interpolates the name of the network by accessing the “name” attribute assigned to the network object.

Listing 4.3 Use attribute interpolation to get the network name

```
import json

class Network:    #A
   def __init__(self, name="hello-network"):    #A
       self.name = name
       self.resource = self._build()    #B

   def _build(self):    #B
       return {
           'google_compute_network': [    #A
               {
                   f'{self.name}': [
                       {
                           'name': self.name    #A
                       }
                   ]
               }
           ]
       }

class Subnet:    #C
   def __init__(self, network, region='us-central1'):    #D
       self.network = network    #D
       self.name = region    #C
       self.subnet_cidr = '10.0.0.0/28'
       self.region = region
       self.resource = self._build()

   def _build(self):
       return {
           'google_compute_subnetwork': [    #C
               {
                   f'{self.name}': [
                       {
                           'name': self.name,    #C
                           'ip_cidr_range': self.subnet_cidr,
                           'region': self.region,
                           'network': self.network.name   #E
                       }
                   ]
               }
           ]
       }

if __name__ == "__main__":
   network = Network()   #B
   subnet = Subnet(network)    #F

   resources = {    #G
       "resource": [    #G
           network.resource,   #B
           subnet.resource    #F
       ]   #G
   }    #G

   with open(f'main.tf.json', 'w') as outfile:    #H
       json.dump(resources, outfile, sort_keys=True, indent=4)    #H
```

#A Create the Google network using a Terraform resource named "hello-network".
#B Use the module to create the JSON configuration for the network.
#C Create the Google subnetwork using a Terraform resource named after the region, "us-central1.".
#D Pass the entire network object to the subnet. The subnet calls the network object for the attributes it needs.
#E Interpolate the network name by retrieving it from the object.
#F Use the module to create the JSON configuration for the subnet and pass the network object to the subnet.
#G Merge the network and subnet JSON objects into a Terraform-compatible JSON structure.
#H Write it out to a JSON file to be executed by Terraform later.

Domain-specific languages

Infrastructure as code tools that use domain-specific languages (DSLs) offer their own variable interpolation format. The example in HashiCorp Terraform would use `google_compute_network.hello-world-network.name` **to dynamically pass the name of the network to the subnet. AWS CloudFormation allows you to reference parameters with** `Ref`**. You can reference** `properties` **of a resource in Azure Bicep.**

Attribute interpolation works between modules or resources in a configuration. However, interpolation only works for specific tools and not necessarily across tools. When you have more resources and modules in composition, you cannot use interpolation.

One alternative to attribute interpolation uses explicit module outputs to pass resource attributes between modules. You can customize outputs to any schema or parameters you need. For example, we can group the subnet and network into one module and export its attributes for the server to use.

Let's refactor the subnet and network and add the server.

Listing 4.4 Setting the subnet name as the output for a module .

```
import json

    #A
class NetworkModule:    #B
   def __init__(self, region='us-central1'):
       self._region = region
       self._network = Network()    #B
       self._subnet = Subnet(self._network)    #B
       self.resource = self._build()    #C

   def _build(self):    #C
       return [    #C
           self._network.resource,    #C
           self._subnet.resource    #C
       ]    #C

   class Output:    #D
       def __init__(self, subnet_name):    #D
           self.subnet_name = subnet_name    #D

   def output(self):    #E
       return self.Output(self._subnet.name)    #E

class ServerModule:    #F
   def __init__(self, name, network,    #G
                zone='us-central1-a'):
       self._name = name
       self._subnet_name = network.subnet_name    #H
       self._zone = zone
       self.resource = self._build()    #I

   def _build(self):    #I
       return [{
           'google_compute_instance': [{
               self._name: [{
                   'allow_stopping_for_update': True,
                   'boot_disk': [{
                       'initialize_params': [{
                           'image': 'ubuntu-1804-lts'
                       }]
                   }],
                   'machine_type': 'e2-micro',
                   'name': self._name,
                   'zone': self._zone,
                   'network_interface': [{
                       'subnetwork': self._subnet_name
                   }]
               }]
           }]
       }]

if __name__ == "__main__":
   network = NetworkModule()    #B
   server = ServerModule("hello-world",    #F
                         network.output())    #G
   resources = {    #J
       "resource": network.resource + server.resource    #J
   }    #J

   with open(f'main.tf.json', 'w') as outfile:    #K
       json.dump(resources, outfile, sort_keys=True, indent=4)    #K
```

#A Network and subnet objects omitted for clarity.
#B Refactor network and subnet creation into a module. This follows the composite pattern. The module creates the Google network and subnet using Terraform resources.
#C Use the module to create the JSON configuration for the network and subnet.
#D Create a nested class for the network module output. The nested class exports the name of the subnet for high-level attributes to use.
#E Create an output function for the network module to retrieve and export all network outputs.
#F This module creates the Google compute instance (server) using a Terraform resource.
#G Pass the network outputs as an input variable for the server module. The server will choose the attributes it needs.
#H Using the network output object, get the subnet name and set it to the server's subnet name attribute.
#I Use the module to create the JSON configuration for the server.
#J Merge the network and server JSON objects into a Terraform-compatible JSON structure.
#K Write it out to a JSON file to be executed by Terraform later.

Domain-specific languages

For a provisioning tool like AWS CloudFormation, Azure Bicep, or HashiCorp Terraform, you generate outputs for modules or stacks that higher-level ones can consume. A configuration management tool such as Ansible passes variables by standard output between automation tasks.

Module outputs help expose specific parameters for high-level resources. The approach copies and repeats the values. However, module outputs can get very complicated! You'll often forget which outputs you exposed and their names. Contract testing in chapter 6 might help you enforce required module outputs.

Rather than use outputs, you can use infrastructure state as a state file or infrastructure provider's API metadata. Many tools keep a copy of the infrastructure state, what I call ***tool state***, to detect drift between actual resource state and configuration and track which resources it manages.

DEFINITION Tool state is a representation of infrastructure state stored by an infrastructure as code tool. It tracks the configuration of resources managed by the tool.

Tools often store their state in a file. You already encountered an example for using tool state in the section on inversion of control. I parsed the name of the network from a file called "terraform.tfstate", which is the tool state for Terraform. However, not all tools offer a state file. As a result, you may have some difficulty parsing low-level resource attributes across tools.

If you have multiple tools and providers in your system, you have two main options. First, consider using a configuration manager as a standard interface to pass metadata. A *configuration manager*, like a key-value store, manages a set of fields and their values.

The configuration manager helps you create your own abstraction layer for tool state. For example, some network automation scripts might read IP address values stored in a key-value store. However, you have to maintain the configuration manager and make sure your infrastructure as code can access it.

As a second option, consider using an infrastructure provider's API. Infrastructure APIs do not often change, provide detailed information, and account for out-of-band changes that a state file may not include. You can use client libraries to access information from infrastructure APIs.

Domain-specific languages

Many provisioning tools offer a capability to make API calls to an infrastructure API. For example, AWS-specific parameter types and `Fn::ImportValue` in AWS CloudFormation retrieve values from the AWS API or other stacks. Azure Bicep offers a keyword called `existing` to import resource properties outside of the current file.

HashiCorp Terraform offers data sources to read metadata on an infrastructure resource from the API. Similarly, a module can reference Ansible facts, which gather metadata about a resource or your environment.

You will encounter a few downsides to using the infrastructure API. Unfortunately, your infrastructure as code needs network access. You won't know the *value* of the attribute until you run the infrastructure as code because the code must make a request to the API. If the infrastructure API experiences an outage, your infrastructure as code may not resolve attributes for low-level resources.

When you add an abstraction with dependency inversion, you protect high-level resources from changing attributes on lower-level resources. While you can't prevent all failures or disruptions, you minimize the blast radius of potential failures due to updated low-level resources. Think of it like a contract - if both high and low-level resources agree on the attributes they need, they can evolve independently of each other.

4.2.3 Applying dependency injection

What happens when you combine inversion of control and dependency inversion? Figure 4.6 shows how you can combine both principles to decouple the server and network example. The server calls the network for attributes and parses the metadata using the infrastructure API or state. If you make changes to the network name, it updates the metadata. The server retrieves the updated metadata and adjusts its configuration separately.

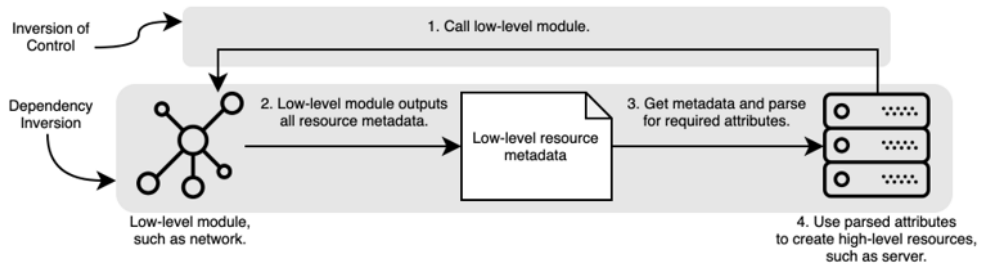

Inversion of Control
Dependency Inversion
1. Call low-level module.
2. Low-level module outputs all resource metadata.
Low-level resource metadata
3. Get metadata and parse for required attributes.
Low-level module, such as network.
4. Use parsed attributes to create high-level resources, such as server.

Figure 4.6. Dependency injection combines the principle of inversion of control and dependency inversion to loosen infrastructure dependencies and isolate low-level and high-level resources.

Harnessing the power of both principles helps promote evolution and composability because the abstraction layer behaves as a buffer between each building block of your system. You use ***dependency injection*** to combine inversion of control and dependency inversion. Inversion of control isolates changes to the high-level modules or resources, while dependency inversion isolates changes to the low-level resources.

> **DEFINITION** **Dependency injection combines the principles of inversion of control and dependency inversion. High-level modules or resources call for attributes from low-level ones through an abstraction.**

Let's implement dependency injection for the server and network example with Apache Libcloud. Apache Libcloud is a library for Google Cloud Platform (GCP) API. I use it to search for the network. The server calls the GCP API for the subnet name, parses the GCP API metadata, and assigns itself the fifth IP address in the network's range.

Listing 4.5 Using dependency injection to create a server on a network .

```
import credentials
import ipaddress
import json
from libcloud.compute.types import Provider    #A
from libcloud.compute.providers import get_driver    #A

def get_network(name):  #B
   ComputeEngine = get_driver(Provider.GCE)  #C
   driver = ComputeEngine(   #D
       credentials.GOOGLE_SERVICE_ACCOUNT,   #D
       credentials.GOOGLE_SERVICE_ACCOUNT_FILE,   #D
       project=credentials.GOOGLE_PROJECT,   #D
       datacenter=credentials.GOOGLE_REGION)   #D
   return driver.ex_get_subnetwork(  #E
       name, credentials.GOOGLE_REGION)  #E

class ServerFactoryModule:    #F
   def __init__(self, name, network, zone='us-central1-a'):
       self._name = name
       gcp_network_object = get_network(network)  #B
       self._network = gcp_network_object.name  #G
       self._network_ip = self._allocate_fifth_ip_address_in_range(   #H
           gcp_network_object.cidr)  #H
       self._zone = zone
       self.resources = self._build()    #I

   def _allocate_fifth_ip_address_in_range(self, ip_range):    #H
       ip = ipaddress.IPv4Network(ip_range)    #H
       return format(ip[-2])    #H

   def _build(self):    #I
       return {
           'resource': [{
               'google_compute_instance': [{    #F
                   self._name: [{
                       'allow_stopping_for_update': True,
                       'boot_disk': [{
                           'initialize_params': [{
                               'image': 'ubuntu-1804-lts'
                           }]
                       }],
                       'machine_type': 'f1-micro',
                       'name': self._name,
                       'zone': self._zone,
                       'network_interface': [{
                           'subnetwork': self._network,    #G
                           'network_ip': self._network_ip    #H
                       }]
                   }]
               }]
           }]
       }

if __name__ == "__main__":
   server = ServerFactoryModule(name='hello-world', network='default')    #F
   with open('main.tf.json', 'w') as outfile:    #K
       json.dump(server.resources, outfile, sort_keys=True, indent=4)    #K
```

#A Import the Apache Libcloud library, which allows you to access the Google Cloud Platform (GCP) API. You must import the provider object and Google driver.
#B This function retrieves the network information using the Apache Libcloud library. The network and subnet were created separately. Their code has been omitted for clarity.
#C Import the Google Compute Engine (GCE) driver for Apache Libcloud.
#D Pass the GCP service account credentials you want Apache Libcloud to use for accessing the GCP API.
#E Use Apache Libcloud driver to get the subnet information by its name.
#F This module creates the Google compute instance (server) using a Terraform resource.
#G Parse the subnet name from the GCP network object returned by Apache Libcloud and use it to create the server.
#H Parse the CIDR block from the GCP network object returned by Apache Libcloud and use it to calculate the fifth IP address on the network. The server uses the result as its network IP address.
#I Use the module to create the JSON configuration for the server.
#K Write it out to a JSON file to be executed by Terraform later.

AWS and Azure equivalent

You can use the AWS Python SDK to get AWS VPC information or Azure libraries for Python to get Azure virtual network information. The SDK interacts with the AWS or Azure API and returns similar information to the GCP example.

Using the infrastructure API as an abstraction layer, I account for the evolution of the network independent of the server. For example, what happens when I change the IP address range for the network? I deploy the update to the network before I run the infrastructure as code for the server. The server calls the infrastructure API for network attributes and recognizes a new IP address range. Then, it recalculates the fifth IP address.

Figure 4.7 shows the responsiveness of the server to the change because of dependency injection. When I change the IP address range for the network, my server gets the updated address range and reallocates the IP address if needed.

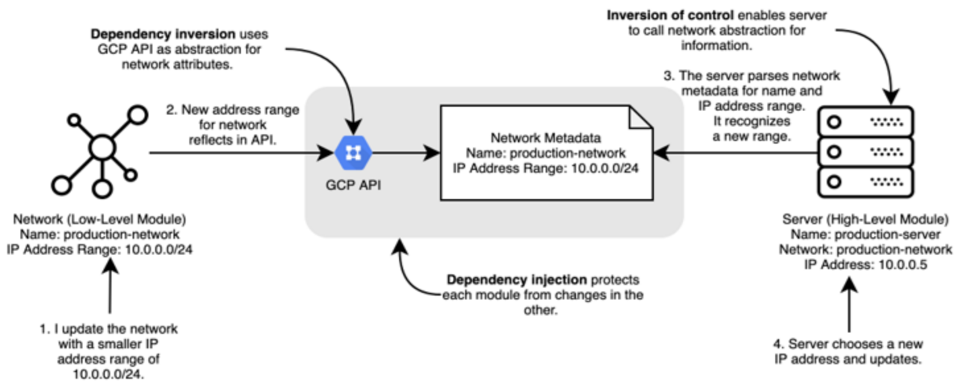

Dependency inversion uses GCP API as abstraction for network attributes.
Inversion of control enables server to call network abstraction for information.
2. New address range for network reflects in API.
3. The server parses network metadata for name and IP address range. It recognizes a new range.
GCP API
Network Metadata
Name: production-network
IP Address Range: 10.0.0.0/24
Network (Low-Level Module)
Name: production-network
IP Address Range: 10.0.0.0/24
Server (High-Level Module)
Name: production-server
Network: production-network
IP Address: 10.0.0.5
Dependency injection protects each module from changes in the other.
1. I update the network with a smaller IP address range of 10.0.0.0/24.
4. Server chooses a new IP address and updates.

Figure 4.7. Dependency injection allows me to change the low-level module (the network) and automatically propagate the change to the high-level module (the server).

Thanks to dependency inversion, you can evolve low-level resources separately from dependencies. Inversion of control helps high-level resources respond to changes in low-level resources. Combining the two as dependency injection ensures the composability of the system, as you can add more high-level resources on the low-level ones. Decoupling due to dependency injection helps you minimize the blast radius of failed changes across modules in your system.

In general, you should apply dependency as an essential principle for infrastructure dependency management. If you apply dependency injection when you write your infrastructure configuration, you sufficiently decouple dependencies so that you can change them independently without affecting other infrastructure. As your module grows, you can continue to refactor to more specific patterns and further decouple infrastructure based on the type of resources and modules.

4.3 Facade

Applying the principle of dependency injection generates similar patterns for expressing dependencies. The patterns align with structural design patterns in software development. In the pursuit of decoupling dependencies, I often find myself repeating the same three patterns in my infrastructure as code.

Imagine you want to create a storage bucket to store some static files. You can control who accesses the files with an access control API in GCP. Figure 4.8 creates the bucket and sets the outputs to include the bucket's name. The access control rules for the bucket can use the outputs to get the bucket's name.

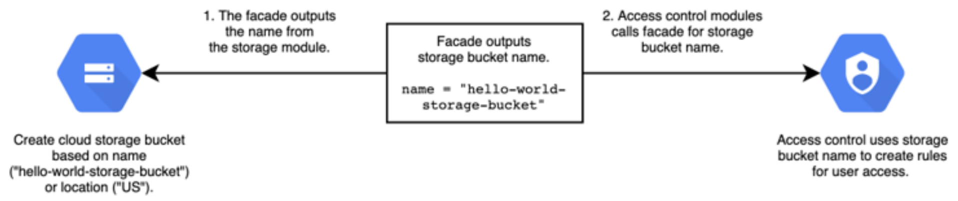

1. The facade outputs the name from the storage module.
Facade outputs storage bucket name.
name = "hello-world-storage-bucket"
2. Access control modules calls facade for storage bucket name.
Create cloud storage bucket based on name ("hello-world-storage-bucket") or location ("US").
Access control uses storage bucket name to create rules for user access.

Figure 4.8. The facade simplifies attributes to the name of the storage bucket for use by the access control module.

The pattern of using outputs and an abstraction layer seems *very* familiar. In fact, you encountered it in the chapter's first half. You've been unknowingly using the facade pattern to pass multiple attributes between modules!

The ***facade pattern*** uses module outputs as the abstraction for dependency injection. It behaves like a mirror, reflecting the attributes to other modules and resources.

DEFINITION The facade pattern outputs attributes from resources in a module for dependency injection.

A facade reflects the attributes and nothing more. The pattern does decouple dependencies between high and low-level resources and conforms to the principle of dependency injection. The high-level resource still calls the low-level resource for information and outputs serve as the abstraction.

Implement the facade pattern in code by building an output method. My bucket module returns the bucket object and name in its output method. My access module uses the output method to retrieve the bucket object and access its name.

Listing 4.6 Output the bucket name as a facade for access control rules

```
import json
import re


class StorageBucketFacade:    #E
   def __init__(self, name):    #E
       self.name = name    #E


class StorageBucketModule:    #A
   def __init__(self, name, location='US'):    #B
       self.name = f'{name}-storage-bucket'
       self.location = location
       self.resources = self._build()

   def _build(self):
       return {
           'resource': [
               {
                   'google_storage_bucket': [{    #B
                       self.name: [{
                           'name': self.name,
                           'location': self.location,
                           'force_destroy': True    #C
                       }]
                   }]
               }
           ]
       }

   def outputs(self):    #D
       return StorageBucketFacade(self.name)    #E


class StorageBucketAccessModule:    #F
   def __init__(self, bucket, user, role):    #G
       if not self._validate_user(user):    #I
           print("Please enter valid user or group ID")
           exit()
       if not self._validate_role(role):    #J
           print("Please enter valid role")
           exit()
       self.bucket = bucket    #G
       self.user = user
       self.role = role
       self.resources = self._build()
```

```
    def _validate_role(self, role):    #I
        valid_roles = ['READER', 'OWNER', 'WRITER']
        if role in valid_roles:
            return True
        return False

    def _validate_user(self, user):    #J
        valid_users_group = ['allUsers', 'allAuthenticatedUsers']
        if user in valid_users_group:
            return True
        regex = r'^[a-z0-9]+[\._]?[a-z0-9]+[@]\w+[.]\w{2,3}$'
        if(re.search(regex, user)):
            return True
        return False

    def _change_case(self):
        return re.sub('[^0-9a-zA-Z]+', '_', self.user)

    def _build(self):
        return {
            'resource': [{
                'google_storage_bucket_access_control': [{
                    self._change_case(): [{
                        'bucket': self.bucket.name,    #H
                        'role': self.role,
                        'entity': self.user
                    }]
                }]
            }]
        }


if __name__ == "__main__":
    bucket = StorageBucketModule('hello-world')
    with open('bucket.tf.json', 'w') as outfile:
        json.dump(bucket.resources, outfile, sort_keys=True, indent=4)

    server = StorageBucketAccessModule(
        bucket.outputs(), 'allAuthenticatedUsers', 'READER')
    with open('bucket_access.tf.json', 'w') as outfile:
        json.dump(server.resources, outfile, sort_keys=True, indent=4)
```

#A Create a low-level module for the GCP storage bucket, which uses the factory pattern to generate a bucket.
#B Create the Google storage bucket using a Terraform resource based on the name and location.
#C Set an attribute on the Google storage bucket to destroy it when you delete Terraform resources.
#D Create an output method for the module that returns a list of attributes for the storage bucket.
#E Using the facade pattern, output the bucket name as part of the storage output object. This implements dependency inversion to abstract away unnecessary bucket attributes.
#F Create a high-level module to add access control rules to the storage bucket.
#G Pass the bucket's output facade to the high-level module.
#H Create Google storage bucket access control rules using a Terraform resource.
#I Validate that the roles passed to the module match valid roles in GCP.
#J Validate that the users passed to the module match valid user group types for all users or all authenticated users.
#A Network facade contains the network name and the IP CIDR range.
#B The network factory module outputs information defined by the network facade.
#C The server accepts the network facade as an input representing the network dependency.

#D The server factory module accesses the network information it needs from the facade instead of the network output.

AWS and Azure equivalent

A GCP storage bucket is similar to an AWS S3 bucket or Azure Blob Storage.

Why output the entire bucket object and not just the name? Remember that you want to build an abstraction layer to conform to the principle of dependency inversion. If you create a new module that depends on the bucket location, you can update the bucket object's facade to output the name and location. The update does not affect the access module.

You can implement a facade with low effort and still get the benefits from decoupling of dependencies. One such benefit includes the flexibility to make isolated, self-contained updates in one module without affecting others. Adding new high-level dependencies does not require much effort.

The facade pattern also makes it easier to debug problems. It mirrors the outputs without adding logic for parsing, making it simple to trace back problems to the source and fix the system. You'll learn more about reverting failed changes in chapter 11.

Domain-specific languages

Using a domain-specific language, you can mimic a facade using an output variable with a customized name. The high-level resource references the customized output names.

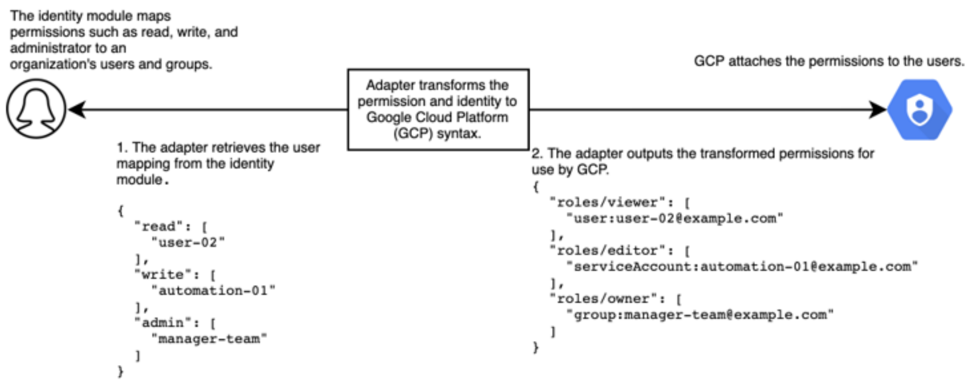
As a general practice, you'll start a facade with one or two fields. Always keep this to the minimum number of fields you'll need for high-level resources. Review and prune the fields when you don't need them every few weeks.

The facade pattern works for simpler dependencies, such as a few high-level modules to one low-level one. However, when you add many high-level modules and the depth of your dependencies increases, you will have difficulty maintaining the facade pattern for the low-level modules. When I need to change a field name in the output, I must change every module that references it. Changing every module reference does not scale when you have hundreds of resources that depend on one low-level module.

4.4 Adapter

The facade mirrors the values as outputs for one infrastructure module to high-level modules in the previous section. It works well for simple dependency relationships but falls apart with more complex modules. More complex modules usually involve one to many dependencies or span multiple infrastructure providers.

Let’s say you have an identity module that passes a list of users and roles for configuring infrastructure. The identity module needs to work across multiple platforms. In figure 4.9, you set up the module to output a JSON-formatted object that maps permissions such as “read,” “write,” or “admin” with the corresponding usernames. Teams must map these usernames and their generic permissions to GCP-specific terms. GCP’s access management uses “viewer,” “editor,” and “owner,” which transform to “read,” “write,” and “admin.”


The identity module maps permissions such as read, write, and administrator to an organization's users and groups.
Adapter transforms the permission and identity to Google Cloud Platform (GCP) syntax.
GCP attaches the permissions to the users.
1. The adapter retrieves the user mapping from the identity module.
2. The adapter outputs the transformed permissions for use by GCP.

```
{
  "read": [
    "user-02"
  ],
  "write": [
    "automation-01"
  ],
  "admin": [
    "manager-team"
  ]
}
```

```
{
  "roles/viewer": [
    "user:user-02@example.com"
  ],
  "roles/editor": [
    "serviceAccount:automation-01@example.com"
  ],
  "roles/owner": [
    "group:manager-team@example.com"
  ]
}
```

Figure 4.9. The adapter pattern transforms attributes to a different interface that high-level modules can consume.

How do you map a generic set of roles to the specific infrastructure provider roles? The mapping needs to ensure that you can reproduce and evolve the module over multiple infrastructure providers. You want to extend the module in the future to add users to equivalent roles across platforms.

As a solution, the ***adapter pattern*** transforms metadata from the low-level resource so that any high-level resource can use it. An adapter behaves like a travel plug. You can change the plug depending on the country’s outlet and still use your electronic devices.

> **DEFINITION** The adapter pattern transforms and outputs metadata from the low-level resource or module so any high-level resource or module can use it.

To start, you create a dictionary that maps generic role names to users. For example, you want to assign a read-only role to the audit team and two users. These generic roles and usernames do not match any of the GCP permissions and roles.

Listing 4.7 Create a static object that maps generic roles to usernames

```
class Infrastructure:
   def __init__(self):
       self.resources = {
           'read': [    #A
               'audit-team',    #A
               'user-01',    #A
               'user-02'    #A
           ],
           'write': [    #B
               'infrastructure-team',    #B
               'user-03',    #B
               'automation-01'    #B
           ],
           'admin': [   #C
               'manager-team'    #C
           ]   #C
       }
```

#A Assign the audit-team, user-01, and user-02 to a read-only role. The mapping describes that the user can only read information on any infrastructure provider.
#B Assign the infrastructure-team, user-02, and automation-01 to a write role. The mapping describes that the user can update information on any infrastructure provider.
#C Assign the manager team to the administrator role. The mapping describes that the user can manage any infrastructure provider.

However, you cannot do anything with the static object with role mappings. GCP does not understand the usernames or roles! Implement the adapter pattern to map generic permissions to the infrastructure-specific permissions. I built an identity adapter specific to GCP, which maps generic permissions like "read" to GCP-specific terms like "roles/viewer." GCP can use the map to add users, service accounts, and groups to the correct roles.

Listing 4.8 Use the adapter pattern to transform generic permissions

```
import json
import access

class GCPIdentityAdapter:    #A
   EMAIL_DOMAIN = 'example.com'    #B

   def __init__(self, metadata):
       gcp_roles = {    #C
           'read': 'roles/viewer',  #C
           'write': 'roles/editor',  #C
           'admin': 'roles/owner'  #C
       }  #C
       self.gcp_users = []
       for permission, users in metadata.items():    #D
           for user in users:    #D
               self.gcp_users.append(    #D
                   (user, self._get_gcp_identity(user),    #E
                       gcp_roles.get(permission)))    #D

   def _get_gcp_identity(self, user):  #E
       if 'team' in user:  #F
           return f'group:{user}@{self.EMAIL_DOMAIN}'  #F
       elif 'automation' in user:  #G
           return f'serviceAccount:{user}@{self.EMAIL_DOMAIN}'  #G
       else:  #H
           return f'user:{user}@{self.EMAIL_DOMAIN}'  #H

   def outputs(self):    #I
       return self.gcp_users    #I

class GCPProjectUsers:    #J
   def __init__(self, project, users):
       self._project = project
       self._users = users
       self.resources = self._build()    #K

   def _build(self):    #K
       resources = []
       for (user, member, role) in self._users:    #C
           resources.append({
               'google_project_iam_member': [{    #L
                   user: [{    #L
                       'role': role,    #L
                       'member': member,    #L
                       'project': self._project    #L
                   }]    #L
               }]    #L
           })
       return {
           'resource': resources
       }

if __name__ == "__main__":
   users = GCPIdentityAdapter(access.Infrastructure().resources).outputs()    #A

   with open('main.tf.json', 'w') as outfile:    #M
       json.dump(    #M
           GCPProjectUsers(    #M
               'infrastructure-as-code-book',    #M
               users).resources, outfile, sort_keys=True, indent=4)    #M
```

#A Create an adapter to map generic role types to Google role types.
#B Set the email domain as a constant, which you'll append to each user.
#C Create a dictionary to map generic roles to GCP-specific permissions and roles.
#D For each permission and user, build a tuple with the user, GCP identity, and role.
#E Transform the usernames to GCP-specific member terminology, which uses user type and email address.
#F If the username has "team", the GCP identity needs to be prefixed with "group" and suffixed with the email domain.
#G If the username has "automation", the GCP identity needs to be prefixed with "serviceAccount" and suffixed with the email domain.
#H For all other users, the GCP identity needs to be prefixed with "user" and suffixed with the email domain.
#I Output the list of tuples containing the users, GCP identities, and roles.
#J Create a module for the GCP project users, which uses the factory pattern to attach users to GCP roles for a given project.
#K Use the module to create the JSON configuration for the project's users and roles.
#L Create a list of Google project IAM members using a Terraform resource. The list retrieves the GCP identity, role, and project to attach a username to read, write, or administrator permissions in GCP.
#M Write it out to a JSON file to be executed by Terraform later.

AWS and Azure equivalent

For those more familiar with AWS, the equivalent policies for each permission set would be "AdministratorAccess" for "admin," "PowerUserAccess" for "write," and "ViewOnlyAccess" for "read." Azure role-based access control uses "Owner" for "admin," "Contributor" for "write," and "Reader" for "read."

You could extend your identity adapter to map the generic dictionary of access requirements to another infrastructure provider, like AWS or Azure. In general, an adapter translates the provider-specific or prototype module-specific language into generic terms. This pattern works best for modules with different infrastructure providers or dependencies. I also use the adapter pattern to create a consistent interface for infrastructure providers with poorly defined resource parameters.

For a more complex example, imagine configuring a virtual private network (VPN) connection between two clouds. Instead of passing network information from each provider through a facade, you use an adapter in figure 4.10. Your network modules for each provider to output a network object with more general fields, such as "name" and "CIDR block". This use case benefits from an adapter because it reconciles two different language semantics (e.g., GCP Cloud VPN Gateway and AWS Customer Gateway).

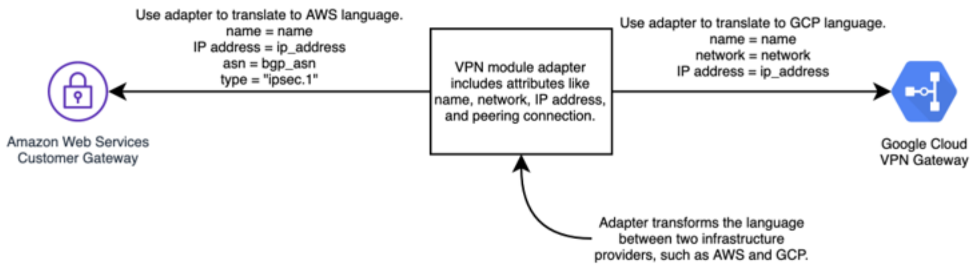

Use adapter to translate to AWS language.
name = name
IP address = ip_address
asn = bgp_asn
type = "ipsec.1"
Amazon Web Services Customer Gateway
VPN module adapter includes attributes like name, network, IP address, and peering connection.
Use adapter to translate to GCP language.
name = name
network = network
IP address = ip_address
Google Cloud VPN Gateway
Adapter transforms the language between two infrastructure providers, such as AWS and GCP.

Figure 4.10. An adapter translates language and attributes between two cloud providers.

Azure equivalent

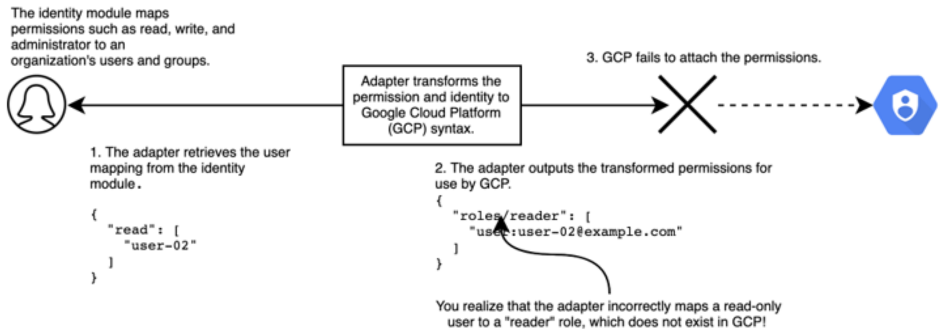
Azure VPN Gateway achieves similar functionality to the AWS Customer Gateway and GCP Cloud VPN Gateway.

Why use the adapter to promote composability and evolvability? The pattern heavily relies on dependency inversion to abstract any transformation of attributes between resources. An adapter behaves as a contract between one to other modules. As long as both modules agree on the contract outlined by the adapter, I can continue to change high-level and low-level modules somewhat independently of each other.

Domain-specific languages

A domain-specific language (DSL) translates the provider or resource-specific language or resource. DSLs implement an adapter within their framework to represent infrastructure state. Infrastructure state often includes the same resource metadata as the infrastructure API. Some tools will allow you to interface with the state file and treat the schema as an adapter for high-level modules.

However, the adapter pattern only works if you *maintain the contract* between modules. Recall that you built an adapter to transform permissions and usernames to GCP. What happens if your teammate accidentally updates the mapping for read-only roles to "roles/reader", which doesn't exist? Figure 4.11 demonstrates that if you don't use the right role specific to GCP, your infrastructure as code fails.


The identity module maps permissions such as read, write, and administrator to an organization's users and groups.
Adapter transforms the permission and identity to Google Cloud Platform (GCP) syntax.
3. GCP fails to attach the permissions.
1. The adapter retrieves the user mapping from the identity module.
2. The adapter outputs the transformed permissions for use by GCP.
You realize that the adapter incorrectly maps a read-only user to a "reader" role, which does not exist in GCP!

```
{
  "read": [
    "user-02"
  ]
}
```

```
{
  "roles/reader": [
    "user:user-02@example.com"
  ]
}
```

Figure 4.11. You need to troubleshoot and test the adapter to map the fields correctly.

In the example, you broke the contract between the generic and GCP roles! The broken contract causes your infrastructure as code to fail. Make sure you maintain and update the correct mappings in your adapter to minimize failure.

Furthermore, troubleshooting becomes more difficult with an adapter. The pattern obfuscates the resources depending on a specific adapter attribute. You need to investigate if an error results from the wrong field output from the source module, an incorrect attribute in the adapter, or the wrong field consumed by the dependent module. Module versioning and testing in chapters 5 and 6 can alleviate the challenges and troubleshooting of an adapter.

4.5 Mediator

The adapter and facade patterns isolate changes and make it easy to manage one dependency. However, infrastructure as code often includes some complex resource dependencies. To detangle the web of dependencies, you can build some very opinionated automation that structures when and how infrastructure as code should create resources.

Imagine you want to add a firewall rule to allow SSH to the server's IP address in our canonical server and network example. However, you can only create the firewall rule if the server exists. Similarly, you can only create the server if the network exists. You need automation to capture the complexity of the relationship between firewall, server, and network.

Let's try to capture the logic of creating the network, the server, and the firewall. Automation can help *mediate* which resources to create first. Figure 4.12 diagrams the workflow for the automation. If the resource is a server, infrastructure as code creates the network and then the server. If the resource is a firewall rule, it creates the network first, the server second, and the firewall rule third.

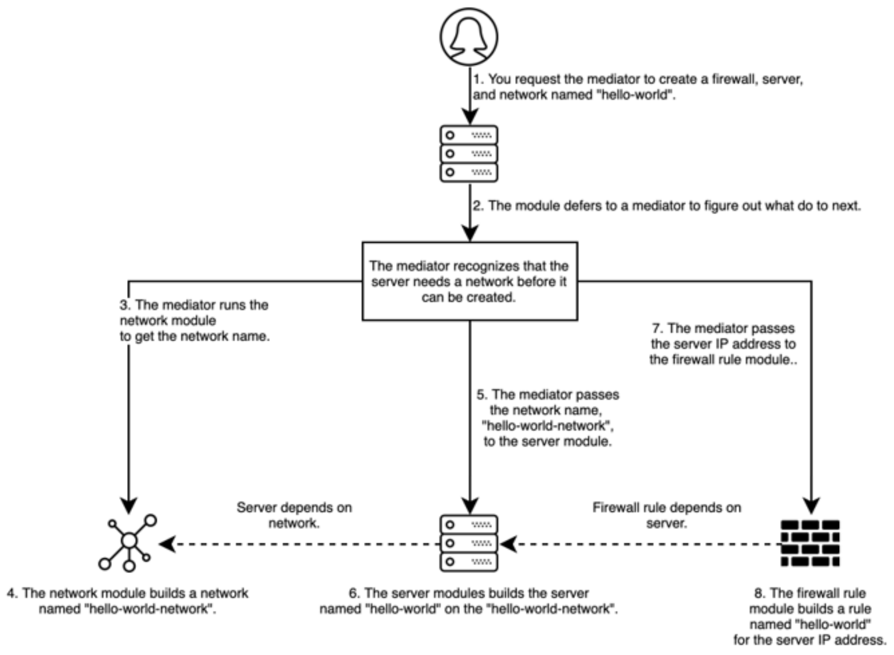

1. You request the mediator to create a firewall, server, and network named "hello-world".
2. The module defers to a mediator to figure out what do to next.
The mediator recognizes that the server needs a network before it can be created.
3. The mediator runs the network module to get the network name.
7. The mediator passes the server IP address to the firewall rule module..
5. The mediator passes the network name, "hello-world-network", to the server module.
Server depends on network.
Firewall rule depends on server.
4. The network module builds a network named "hello-world-network".
6. The server modules builds the server named "hello-world" on the "hello-world-network".
8. The firewall rule module builds a rule named "hello-world" for the server IP address.

Figure 4.12. The mediator becomes the authority on which resource to configure first.

The infrastructure as code implements dependency injection to abstract and control network, server, and firewall dependencies. It relies on the principle of idempotency to run continuously and achieve the same end state (network, server, and firewall), no matter the existing resources. Composability also helps to establish the building blocks of infrastructure resources and dependencies.

This pattern, called the ***mediator pattern***, works like air traffic control at an airport. It controls and manages inbound and outbound flights. A mediator's sole purpose is to organize the dependencies between these resources and create or delete objects as needed.

DEFINITION The mediator pattern organizes dependencies between infrastructure resources and includes logic to create or delete objects based on their dependencies.

Let's implement the mediator pattern for the network, server, and firewall. Implementing a mediator in Python requires a few "if-else" statements to check each resource type and build its low-level dependencies. For example, the firewall depends on creating the server and the network first.

Listing 4.9 Use the mediator pattern to organize server and dependencies.

```
import json
from server import ServerFactoryModule    #A
from firewall import FirewallFactoryModule    #A
from network import NetworkFactoryModule    #A


class Mediator:    #B
   def __init__(self, resource, **attributes):
       self.resources = self._create(resource, **attributes)

   def _create(self, resource, **attributes):     #C
       if isinstance(resource, FirewallFactoryModule):    #D
           server = ServerFactoryModule(resource._name)   #D
           resources = self._create(server)   #D
           firewall = FirewallFactoryModule(     #E
               resource._name, depends_on=resources[1].outputs())     #E
           resources.append(firewall)     #E
       elif isinstance(resource, ServerFactoryModule):    #F
           network = NetworkFactoryModule(resource._name)    #F
           resources = self._create(network)    #F
           server = ServerFactoryModule(    #G
               resource._name, depends_on=network.outputs())    #G
           resources.append(server)    #G
       else:    #H
           resources = [resource]    #H
       return resources

   def build(self):    #J
       metadata = []    #J
       for resource in self.resources:    #J
           metadata += resource.build()    #J
       return {'resource': metadata}    #J


if __name__ == "__main__":
   name = 'hello-world'
   resource = FirewallFactoryModule(name)    #I
   mediator = Mediator(resource)    #I

   with open('main.tf.json', 'w') as outfile:    #K
       json.dump(mediator.build(), outfile, sort_keys=True, indent=4)    #K
```

#A Import the factory modules for the network, server, and firewall.
#B Create a mediator to decide how and what order to automate changes to resources.
#C When you call the mediator to create a resource like a network, server, or firewall, you allow the mediator to decide all the resources to configure.
#D If you want to create a firewall rule as a resource, the mediator will recursively call itself to create the server first.
#E After the mediator creates the server configuration, it builds the firewall rule configuration.
#F If you want to create a server as a resource, the mediator will recursively call itself to create the network first.
#G After the mediator creates the network configuration, it builds the server configuration.
#H If you pass any other resource to the mediator, like the network, it will build its default configuration.
#I Pass the mediator a firewall resource. The mediator will create the network, server, and then the firewall configuration.
#J Use the module to create a list of resources from the mediator and render the JSON configuration.
#K Write it out to a JSON file to be executed by Terraform later.

AWS and Azure equivalent

Firewall rules for GCP are similar in behavior to AWS security group or Azure network security group rules. They control ingress and egress traffic to and from IP address ranges to tagged targets.

If you have a new resource, like a load balancer, you can expand the mediator to build it after the server or firewall. The mediator pattern works best with modules that have many levels of dependencies and multiple system components.

However, you might find the mediator challenging to implement. The mediator pattern must follow idempotency. You need to run multiple times and achieve the same target state. You have to write and test all of the logic in a mediator. If you do not test your mediator, you may accidentally break a resource. Writing your own mediator takes lots of code!

Fortunately, you do not have to implement your own mediator often. Most infrastructure as code tools behave as mediators to resolve complex dependencies and decide how to create resources. The majority of provisioning tools have built-in mediators to identify dependencies and order-of-operations. For example, the container orchestration of Kubernetes uses a mediator to orchestrate changes to the resources in the cluster. Ansible uses a mediator to determine which automation steps to compose and run from various configuration modules.

Infrastructure dependencies and graph theory

Some infrastructure as code tools implement the mediator pattern by mapping dependencies between resources using graph theory. The resources serve as nodes. Links pass attributes to dependent resources. If you want to create resources without a tool, you can manually diagram dependencies in your system. Diagrams can help organize your automation and code. It also identifies which modules you can decouple. The exercise of graphing dependencies might help you implement a mediator.

I only implement the mediator pattern when I cannot find it in a tool or need something between tools. For example, I sometimes write a mediator to control creating a Kubernetes cluster in one tool before another tool deploys services on the Kubernetes cluster. A mediator reconciles automation between these two tools, such as checking cluster health before deploying services with the second tool.

4.6 Choosing a pattern

The facade, adapter, and mediator all use the principle of dependency injection to decouple changes between high-level and low-level modules. You can apply any of the patterns, and they will express dependencies between modules and isolate changes within them. As your system grows, you may need to change these patterns depending on the structure of your module.

Your choice of pattern depends on how many dependencies you have on a low-level module or resource. The facade pattern works for one low-level module to a few high-level ones.

Consider an adapter if you have a low-level module with many high-level module dependencies. When you have many dependencies between modules, you may need a mediator to control resource automation. Figure 4.13 outlines the decision tree for identifying which dependency pattern to use.

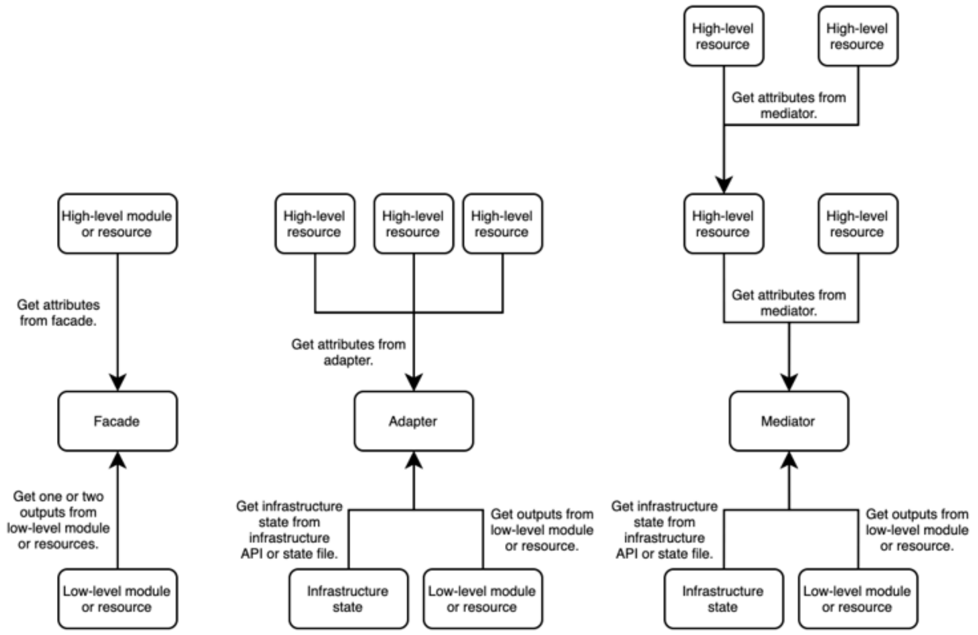

High-level resource
High-level resource
Get attributes from mediator.
High-level module or resource
High-level resource
High-level resource
High-level resource
High-level resource
High-level resource
Get attributes from facade.
Get attributes from adapter.
Get attributes from mediator.
Facade
Adapter
Mediator
Get one or two outputs from low-level module or resources.
Get infrastructure state from infrastructure API or state file.
Get outputs from low-level module or resource.
Get infrastructure state from infrastructure API or state file.
Get outputs from low-level module or resource.
Low-level module or resource
Infrastructure state
Low-level module or resource
Infrastructure state
Low-level module or resource

Figure 4.13. Choosing your abstraction depends on the relationship of the dependency, whether it be intra-module, one to one, or one to many.

All of the patterns promote idempotency, composability, and evolvability through dependency injection. However, why would you start with a facade and then consider the adapter or mediator? As your system grows, you will need to optimize your dependency management pattern to reduce the operational burden of changes.

Figure 4.14 shows the relationship between troubleshooting and implementation effort and scalability and isolation for facade, mediator, and adapter patterns. For example, a facade has the benefit of minimal effort for implementation and troubleshooting but does not scale or isolate changes with more resources. Adapters and mediators offer improved scalability and isolation at the cost of troubleshooting and implementation effort.

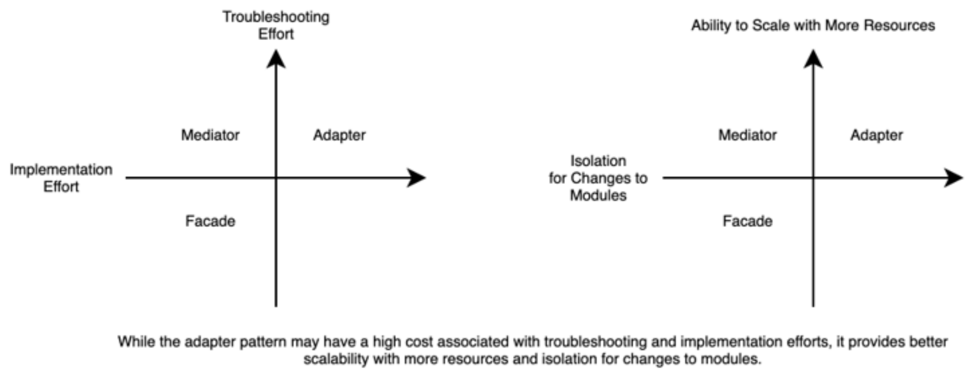

Troubleshooting Effort
Implementation Effort
Mediator
Adapter
Facade
Ability to Scale with More Resources
Isolation for Changes to Modules
Mediator
Adapter
Facade
While the adapter pattern may have a high cost associated with troubleshooting and implementation efforts, it provides better scalability with more resources and isolation for changes to modules.

Figure 4.14. Some patterns may have a low cost of troubleshooting and implementation but cannot isolate changes to modules and scale.

Lower your initial effort by choosing a tool with a mediator implementation. Then, use the tool's built-in facade implementation to manage dependencies between modules or resources. When you find it difficult to manage a facade because you have multiple systems depending on each other, you can start examining an adapter or mediator.

An adapter takes more effort to implement but provides the best foundation for expanding and growing your infrastructure system. You can always add new infrastructure providers and systems without worrying about changing low-level modules. However, you cannot expect to use the adapter for every module because it takes time to implement and troubleshoot.

A tool with a mediator chooses which components get updated and when. An existing tool lowers your overall implementation effort but introduces some concerns during troubleshooting. You need to know your tool's behavior to troubleshoot failed changes for dependencies. Depending on how you use the tool, a tool with a mediator allows you to scale but may not fully isolate changes to modules.

Exercise 4.1

Given:

```
class Database:
  def __init__(self, name):
    spec = {
      'name': name,
      'settings': {
        'ip_configuration': {
          'private_network': 'default'
        }
      }
    }
```

How can we better decouple the database's dependency on the network?

A. **The approach adequately decouples the database from the network.**
B. **Pass the network ID as a variable instead of hard-coding it as `default`.**
C. **Implement and pass a `NetworkOutput` object to the database module for all network attributes.**
D. **Add a function to the network module to push its network ID to the database module.**
E. **Add a function to the database module to call the infrastructure API for the `default` network ID.**

Find answers and explanations at the end of the chapter.

4.7 Exercises and Solutions

Exercise 4.1

Given:

```
class Database:
  def __init__(self, name):
    spec = {
      'name': name,
      'settings': {
        'ip_configuration': {
          'private_network': 'default'
        }
      }
    }
```

How can we better decouple the database's dependency on the network?

A. The approach adequately decouples the database from the network.
B. Pass the network ID as a variable instead of hard-coding it as default.
C. Implement and pass a NetworkOutput object to the database module for all network attributes.
D. Add a function to the network module to push its network ID to the database module.
E. Add a function to the database module to call the infrastructure API for the default network ID.

Answer:

You can implement the adapter pattern to output the network attributes (C). The database chooses which network attributes it uses, such as network ID or CIDR block. This approach best follows the principle of dependency injection. While D does implement dependency inversion, it does not implement inversion of control. Answer E does implement dependency injection but continues to hard-code the network ID.

4.8 Summary

- Apply infrastructure dependency patterns such as facade, adapter, mediator to decouple modules and resources, and you can make changes to modules in isolation.
- Inversion of control states that the high-level resource calls the low-level one for attributes.
- The dependency inversion principle states that the high-level resource should use an abstraction of low-level resource metadata.
- Dependency injection combines the principles of inversion of control and dependency inversion.
- If you do not recognize an applicable pattern, you can use dependency injection for a high-level resource to call a low-level resource and parse its object structure for the values it needs.
- Use the facade pattern to reference a simplified interface for attributes.
- Use the adapter pattern to transform metadata from one resource for another to use. It works best with resources across different infrastructure providers or prototype modules.
- The mediator pattern organizes the dependencies between these resources and creates or deletes objects as needed. Most tools serve as a mediator between resources.

5 Structuring and sharing modules

This chapter covers

- **Constructing module versions and tags for infrastructure changes**
- **Choosing a single repository versus multiple repositories**
- **Organizing shared infrastructure modules across teams**
- **Releasing infrastructure modules without affecting critical dependencies**

Up to this point in the book, you learned practices and patterns for writing infrastructure as code and breaking them down into groups of infrastructure components. However, you can write the most optimal configurations but still have trouble maintaining and mitigating the risk of failure to your systems. The difficulties happen because your team does not standardize collaboration practices when updating infrastructure modules.

Imagine a company, Datacenter for Veggies, starts by automating their growing operations for herbs. Applications in Google Cloud Platform monitor and adjust for optimal herb growth. Each team uses the singleton pattern and creates a unique infrastructure configuration. Over time, Datacenter for Veggies becomes more popular and wants to expand to all vegetables. It hires a new application development team specializing in growing software for different vegetables, from herbs to leafy greens to root vegetables. Each team creates an infrastructure configuration independent of the others.

Datacenter for Veggies hires you to develop an application to grow fruit. You realize you cannot reuse any infrastructure configurations because they are unique to each vegetable team. The company needs a consistent, reusable way to build, secure, and manage infrastructure.

You realize that Datacenter for Veggies could use some module patterns from chapter 3 to organize infrastructure configuration into modules for composability. In figure 5.1, you sketch out a diagram to organize and coordinate groups of infrastructure for different teams.

The herb, root vegetable, leafy green vegetable, and fruit teams can all use standardized configuration for networks, databases, and servers.

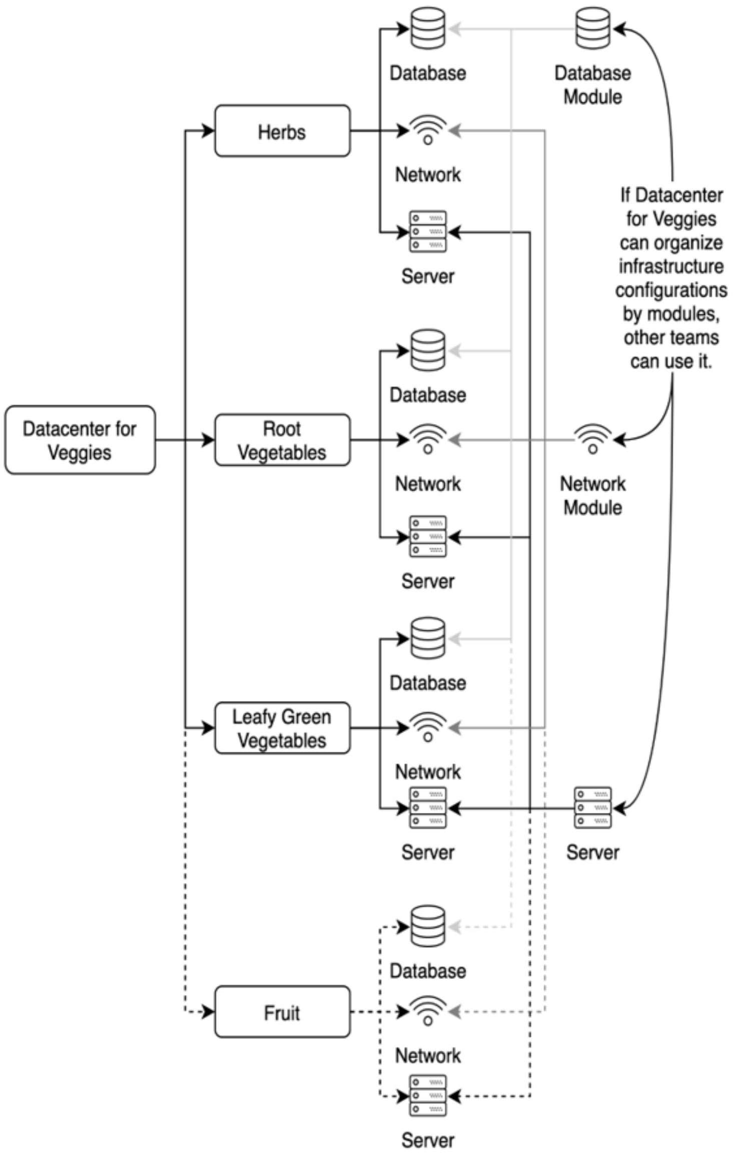


Figure 5.1. Datacenter for Veggies can use modules to organize and standardize infrastructure configuration across application teams.

Sharing modules across different teams promotes reproducibility, composability and evolvability. The teams do not have to spend as much time building infrastructure as code because they reproduce established configurations. Team members can choose how they compose their systems and override the configurations for their specific needs.

To fully realize the benefit of standardized modules, you need to treat them with a development lifecycle outside of regular infrastructure changes. This chapter will cover practices for sharing and managing infrastructure modules. You'll learn techniques and practices to release stable modules without introducing critical failures to higher-level dependencies.

5.1 Repository structure

Imagine that each team in Datacenter for Veggies uses a singleton pattern for their infrastructure. Herbs and Leafy Greens teams realize they use similarly configured servers, networks, and databases. Can they merge their infrastructure configuration into one module?

Rather than copy and paste each other's configuration, Herbs and Leafy Greens teams want to update it in one place and reference it in their configuration. Should Datacenter for Veggies put all infrastructure in one repository? Or should it divide its modules across multiple repositories?

5.1.1 Single repository

At first, each Datacenter for Veggies team stored their infrastructure configuration into a single code repository. Every team organized their configuration into a dedicated directory so they do not mix up configurations. If they want to reference a module, they import the module using a local file path.

Figure 5.2 shows how Datacenter for Veggies structures their single code repository. The repository contains two folders at the top level separating modules and environments. The company subdivides the environments directory for each team, such as the Leafy Green team. The Leafy Green team separates configurations by development and production environments.

When the Leafy Green team wants to create a database, they can use a module in the "modules" folder. In their infrastructure as code, they import the module by setting a local path. After importing, they can use the database factory and build the resource in production.

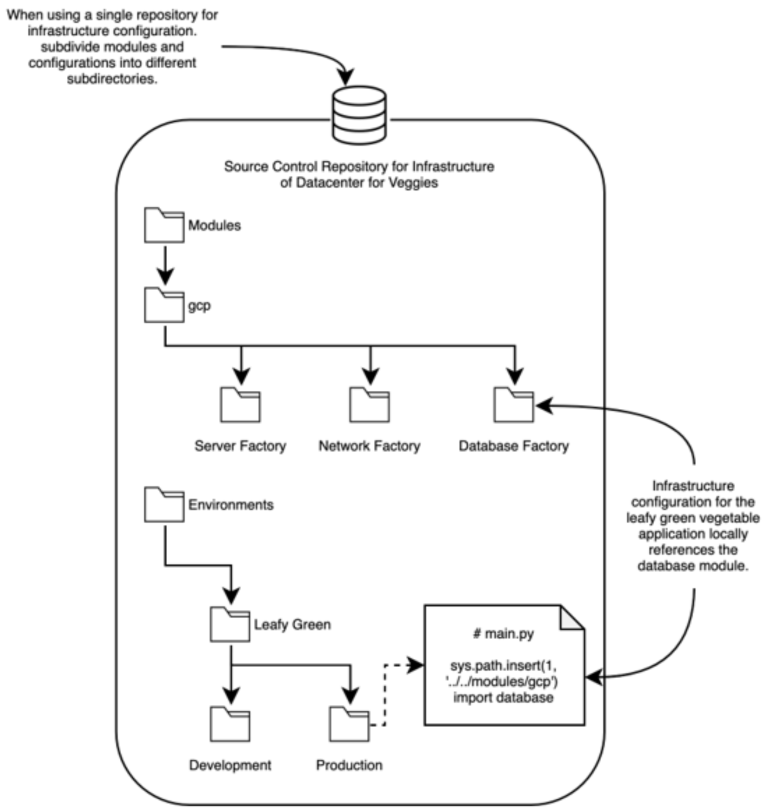

When using a single repository for infrastructure configuration. subdivide modules and configurations into different subdirectories.
Source Control Repository for Infrastructure of Datacenter for Veggies
Modules
gcp
Server Factory
Network Factory
Database Factory
Environments
Leafy Green
Development
Production
# main.py
sys.path.insert(1, '../../modules/gcp')
import database
Infrastructure configuration for the leafy green vegetable application locally references the database module.

Figure 5.2. The leafy green team's production and development environments use the directories containing server, network, and database factory modules in a single repository structure.

Datacenter for Veggies started defining infrastructure with a ***single repository***, also known as a mono repository or monorepo, to contain all configuration and modules for each team.

> **DEFINITION** A single repository structure (also known as mono repository or monorepo) contains all infrastructure as code (configuration and modules) for a team or function.

In general, the company likes the single repository structure. All teams can reproduce their configuration with copy-and-paste and compose new resources by adding a new folder for a module. For example, the Leafy Green team builds a new database module. Using Python, they insert a local file path to modules using the `sys.path` method. They use the database by importing its module into the codebase.

Listing 5.1 Reference infrastructure modules in a different directory

```
import sys
sys.path.insert(1, '../../modules/gcp')     #A

from database import DatabaseFactoryModule     #B
from server import ServerFactoryModule     #B
from network import NetworkFactoryModule     #B

import json


if __name__ == "__main__":
   environment = 'production'
   name = f'{environment}-hello-world'
   network = NetworkFactoryModule(name)    #B
   server = ServerFactoryModule(name, environment, network)    #B
   database = DatabaseFactoryModule(name, server, network, environment)    #B
   resources = {    #C
       'resource': network.build() + server.build() + database.build()    #C
   }    #C

   with open('main.tf.json', 'w') as outfile:    #D
       json.dump(resources, outfile, sort_keys=True, indent=4)    #D
```

#A Import the directory with the modules because it exists in the same repository.
#B Import the server, database, and network factory modules for the production environment.
#C Use the modules to create the JSON configuration for the network, server, and database.
#D Write it out to a JSON file to be executed by Terraform later.

Using local folders to store modules helps the teams reference the infrastructure they want. Everyone can look in the same repository for modules or examine other teams' configuration. If someone on the Herbs team wants to learn about the Fruits team's infrastructure as code, they can use the `tree` command to examine the directory structure.

```
$ tree .
.
├── environments
│   ├── fruits
│   │   ├── development
│   │   └── production
│   ├── herbs
│   │   ├── development
│   │   └── production
│   ├── leafy-greens
│   │   ├── development
│   │   └── production
│   └── roots
│       ├── development
│       └── production
└── modules
    └── gcp
        ├── database.py
        ├── network.py
        ├── server.py
        └── tags.py
```

To better organize configurations, each team separates development and production environment configurations into their folders. Separate directories isolate environment configurations and changes to each environment. Ideally, all environments should be the same. Realistically, you will have differences between environments to address cost or resource constraints.

Other tools

A single repository structure applies to many other infrastructure as code tools. You can apply the single repository structure for reusing roles and playbooks for configuration management tools like Ansible. You can reference and build playbooks or configuration management modules based on each local directory in the single repository.

However, AWS CloudFormation works a bit differently. You can host all of your stack definition files in a single repository. However, you must release the "child" template (what I consider a module) into an S3 bucket and reference it with the `TemplateURL` parameter in the `AWS::CloudFormation::Stack` resource. Later in this chapter, you'll learn how to deliver and release changes to modules.

Datacenter for Veggies uses one infrastructure provider, Google Cloud Platform (GCP). In the future, teams can add new directories for different infrastructure tools or providers. These tools can update servers or networks (`ansible` directory), build virtual machine images (`packer` directory), or deploy the database to Amazon Web Services (`aws` directory).

```
$ tree .
.
├── environments
│   ├── development
│   └── production
└── modules
    └── ansible
    └── aws
    └── gcp
    └── packer
```

You may encounter the principle of *"don't repeat yourself" (DRY)* in other infrastructure as code materials. DRY promotes reuse and composability. Infrastructure modules reduce duplication and repetition in configuration, which conforms to DRY. If you can have identical development and production environments, you could omit the development and production directories instead of separate environment files and reference one module.

You cannot fully comply with DRY in infrastructure. Depending on the infrastructure or tool's language and syntax, you will always have repetitive configuration. As a result, you can have occasional repetition for clearer configuration or within the limitations of tools or platforms.

5.1.2 Multiple repositories

As Datacenter for Veggies grows, its infrastructure repository has hundreds of folders. Each folder contains many more nested ones. Every week, you spend time rebasing the configuration with all the updates from every repository. You also wait 20 minutes each time you push to production because your continuous integration framework must recursively search for changes. The security team also expresses concern because contractors working with the leafy green team have access to all of the fruit team's infrastructure!

You divide network, tag, server, and database modules into individual repositories. Each repository has its workflow for building and delivering the module, which takes less time for the continuous integration framework. You can control access to each repository, only allowing contractors on the leafy green team to access the leafy green configuration.

Different teams in Datacenter for Veggies can use the module's repository or packaged version. Each team stores its configuration and modules in a separate repository. Anyone in the company can download and use the modules in their configurations.

Figure 5.2 shows the different code repositories Datacenter for Veggies uses to create infrastructure as code. Each team and module gets their own code repository. When the Leafy Green team wants to create a database, they download and import the database module from a GitHub repository URL instead of the local folder. If teams have multiple environments, they subdivide their code repository into folders.

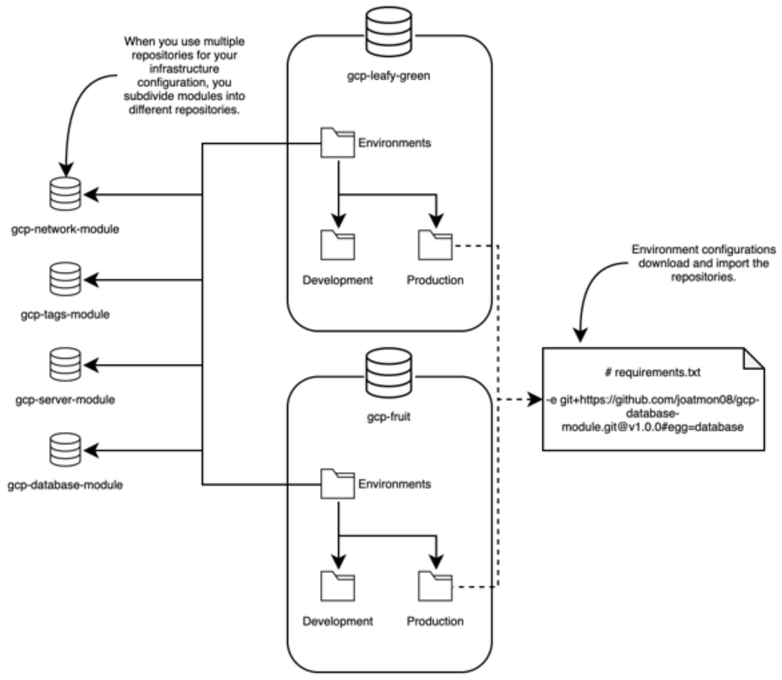

When you use multiple repositories for your infrastructure configuration, you subdivide modules into different repositories.
gcp-leafy-green
Environments
Development
Production
gcp-network-module
gcp-tags-module
gcp-server-module
gcp-database-module
Environment configurations download and import the repositories.
# requirements.txt
-e git+https://github.com/joatmon08/gcp-database-module.git@v1.0.0#egg=database
gcp-fruit
Environments
Development
Production

Figure 5.3. In a multiple repository structure, you store each module in its own code repository. The configuration references the repository URL to use the module.

Datacenter for Veggies migrated from a single repository structure to a ***multiple repository***, or multi repo, structure. They separated modules into different repositories based on the teams.

> **DEFINITION** A multiple repository (also known as multi repo) structure separates infrastructure as code (configuration or modules) into different repositories based on team or function.

Recall that a single repository pattern promotes reproducibility and composability. A multiple repository pattern helps improve the principle of evolvability. Separating the modules into their own repositories helps structure each module's lifecycle and management.

To implement a multiple repository structure, you split the modules into their own version control repository. Then, you configure Python's package manager to download each module by adding them as library requirements in `requirements.txt`. Each library requirement must include a URL to the version control repository and a specific tag to download.

Listing 5.2 Python requirements.txt references module repositories

```
-e git+https://github.com/joatmon08/gcp-tags-module.git@1.0.0#egg=tags    #A
-e git+https://github.com/joatmon08/gcp-network-module.git@1.0.0#egg=network    #B
-e git+https://github.com/joatmon08/gcp-server-module.git@0.0.1#egg=server    #B
-e git+https://github.com/joatmon08/gcp-database-module.git@1.0.0#egg=database    #B
```

#A Download the prototype module for tags from a GitHub repository. Pick the module version based on tag.
#B Download the factory module for the network, server, and database from a GitHub repository. Pick the module version based on tag.

First, you create a repository for the production configuration of the fruit application's infrastructure. After you create the repository, you add the `requirements.txt` to it. Then, you run Python's package installation manager to download each module for the infrastructure configuration.

```
$ pip install -r requirements.txt
Obtaining tags from git+https://github.com/joatmon08/gcp-tags-module.git@1.0.0#egg=tags
...
Successfully installed database network server tags
```

Rather than set a local path and import the modules, you need to run Python's package installation manager to download from the remote repository first. After downloading the modules, teams can import them in environment configurations using Python.

Listing 5.3 Import the modules for use in infrastructure configuration

```
from tags import StandardTags    #A
from server import ServerFactoryModule    #A
from network import NetworkFactoryModule    #A
from database import DatabaseFactoryModule    #A

import json

if __name__ == "__main__":
   environment = 'production'
   name = f'{environment}-hello-world'

   tags = StandardTags(environment)
   network = NetworkFactoryModule(name)
   server = ServerFactoryModule(name, environment, network, tags.tags)
   database = DatabaseFactoryModule(
       name, server, network, environment, tags.tags)
   resources = {
       'resource': network.build() + server.build() + database.build()    #B
   }

   with open('main.tf.json', 'w') as outfile:    #C
       json.dump(resources, outfile, sort_keys=True, indent=4)    #C
```

#A Import the modules downloaded by the package manager.
#B Use the modules to create the JSON configuration for the network, server, and database.
#C Write it out to a JSON file to be executed by Terraform later.

Recall that Datacenter for Veggies separately configures the development and production environments. The teams would implement code to reference the same factory and prototype modules hosted in version control. Consistent modules between development and production environments prevent drift between environments and help you test module changes before production. You'll learn more about testing and environments in chapter 6.

The infrastructure as code implementation for a multiple repository does not differ too much from a single repository. Both structures support reproducibility and composability. However, they differ in that you independently evolve a module in an external repository.

Updating a configuration in a multiple repository structure involves re-downloading new modules with your package manager. Running the package manager to use a new module can introduce friction in your infrastructure as code workflow. Someone may update a module, and you won't know unless you review its repository. Later in this chapter, you'll learn about solving this problem with versioning.

Domain-specific languages

If a tool can reference modules or libraries with version control or artifact URLs, it can support a multiple repository structure.

When you adopt a multiple repository structure, you must establish a few standard practices to share and maintain modules. First, standardize a module file structure and format. It helps the different teams across your organization identify and filter modules in version control. Consistent file structures and naming for module repositories also help with auditing and future automation.

For example, infrastructure modules in Datacenter for Veggies follow the same pattern and file structure. Their names include *infrastructure provider, resource, and tool or purpose*. In figure 5.4, the "gcp-server-module" describes GCP as the infrastructure provider, server as the resource type, and module as the purpose.

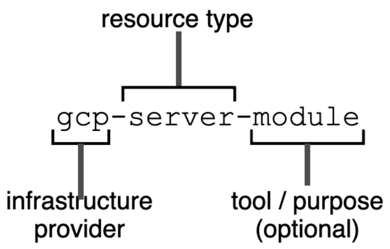


Figure 5.4. The repository name should include the infrastructure provider, resource type, and purpose.

If your modules use a specific tool or have a unique purpose, you can append it to the end of the repository name. It helps to add the tool to the name to identify the module type. Similar to the practices outlined in chapter 2, you want your module name descriptive enough for a teammate to identify.

You can apply the repository naming approach to naming folders in a single repository as well. However, subdirectories in a single repository make it easier to nest and identify infrastructure provider and resource type. Depending on your organization and team's preferences, you can always add more fields to a repository name.

5.1.3 Choosing a repository structure

The scalability of your system and continuous integration framework decides the use of a single repository and multiple repositories. Datacenter for Veggies started with a single repository, which worked well because it had tens of modules and a few environments. Each has two environments for development and production. Each environment needs a few servers, one database, a network, and a monitoring system.

A single repository provides a few benefits. Figure 5.5 outlines some of the advantages and limitations. First, anyone in your team can access modules and configurations in one repository. Second, you only need to go to one place to compare and identify differences between environments. For example, you can compare two files in the repository to check if development uses three servers and production uses five servers.

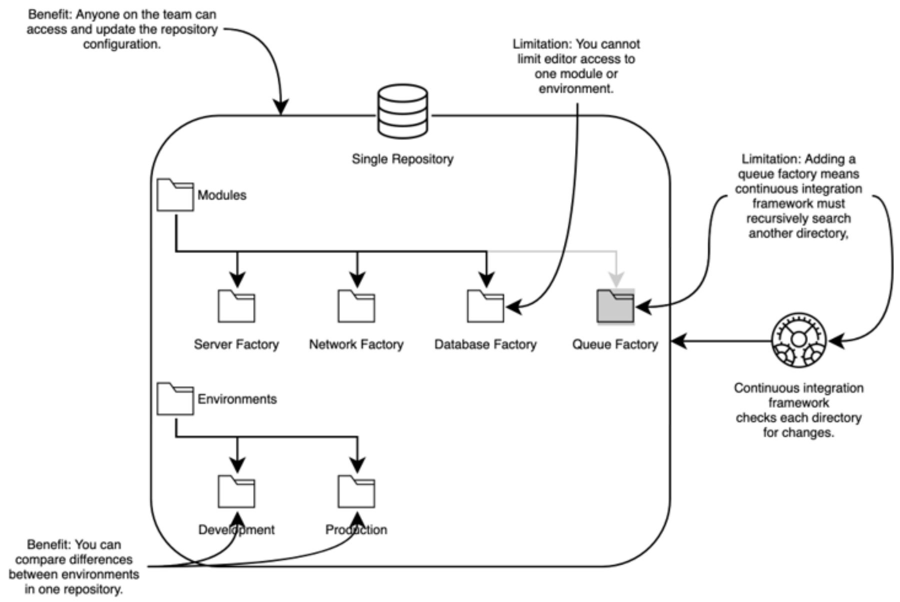

Benefit: Anyone on the team can access and update the repository configuration.
Limitation: You cannot limit editor access to one module or environment.
Single Repository
Modules
Limitation: Adding a queue factory means continuous integration framework must recursively search another directory.
Server Factory
Network Factory
Database Factory
Queue Factory
Continuous integration framework checks each directory for changes.
Environments
Development
Production
Benefit: You can compare differences between environments in one repository.

Figure 5.5. A single repository offers one view for all modules and configurations but limits continuous integration frameworks or granular access control.

Drawing on the infrastructure as code principles, a single repository structure still offers composability, evolvability, and reproducibility. Anyone can go into a folder and evolve a module. You can still build modules on each other because you have a singular view of all infrastructure and configuration.

On the other hand, a single repository structure has some limitations. If anyone can go and change a module, it could break the infrastructure as code that depends on it! Furthermore, your continuous integration system might break down as it recursively checks each directory for changes.

As a result, you need to adopt some practices and tools to handle single repositories. These include opinionated versioning and specialized build systems. If your organization cannot build or adopt a tool that helps alleviate single repository management, you may choose a multiple repository structure.

Single repository build systems

You will find a few tools that help with building and managing single repositories. They have additional code to handle nested subdirectories and individual build workflows. Some of them include Bazel, Pants, and yarn.

The migration from single to multiple repository structure happens more than you think. I had to do it twice! One organization started with three environments and four modules. Over a few years, the infrastructure as code grew to hundreds of modules and environments.

Unfortunately, the continuous integration framework (Jenkins) took nearly three hours to run a standard change scaling up servers. The framework spent most of its time searching each directory and nested directory for changes! We eventually refactored the configurations and modules into multiple repositories.

Refactoring into multiple repositories alleviated some of the problems with the continuous integration framework. A multiple repository structure also provided more granular access control to specific modules. The security team could grant module edit access to specific teams. You'll learn more about refactoring in Chapter 10.

Figure 5.6 shows the benefits and limitations of multiple repositories, including granular access control and scalable continuous integration workflows. However, a multiple repository structure reduces your singular view of modules and configurations for your organization.

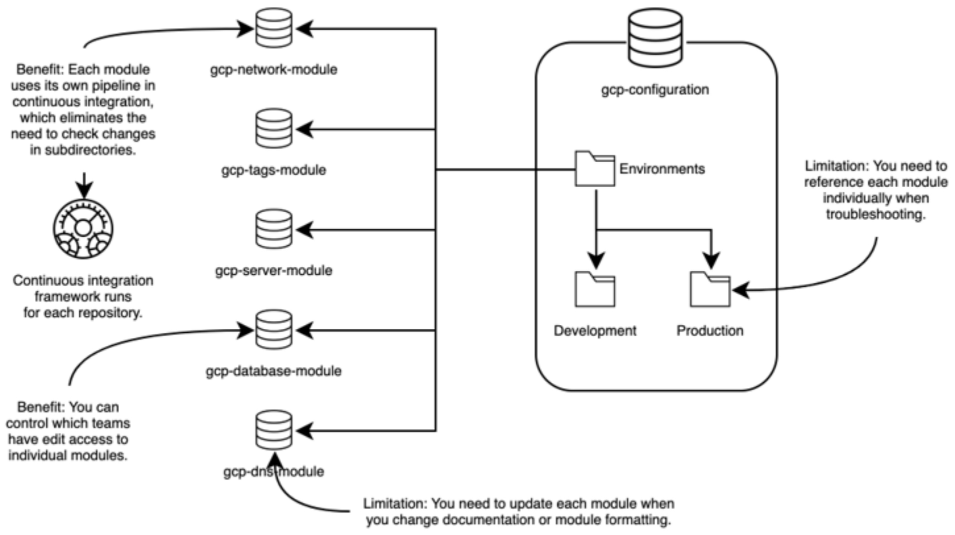

Benefit: Each module uses its own pipeline in continuous integration, which eliminates the need to check changes in subdirectories.
Continuous integration framework runs for each repository.
Benefit: You can control which teams have edit access to individual modules.
gcp-network-module
gcp-tags-module
gcp-server-module
gcp-database-module
gcp-dns-module
Limitation: You need to update each module when you change documentation or module formatting.
gcp-configuration
Environments
Development
Production
Limitation: You need to reference each module individually when troubleshooting.

Figure 5.6. Multiple repositories help reduce the burden of running tests and configurations with continuous integration frameworks but require constant verification of conformance to formatting and troubleshooting.

By refactoring the configuration into a multiple repository structure, you can isolate access to and evolve the infrastructure configuration for specific teams. You have greater control over the evolution and life cycle of modules. Most continuous integration frameworks support multiple repositories and will run workflows in parallel when it detects changes to a given repository.

However, multiple repositories do have some downsides. Imagine Datacenter for Veggies has ten or more modules in different repositories. How do you know if they all conform to the same file standards and naming?

Figure 5.7 shows one solution to the problem of file and standard conformance. You can capture all of the tests for formatting and linting checks into a prototype module. Then, the continuous integration (CI) framework downloads the tests and checks for README and Python files in the server, network, database, and DNS modules.

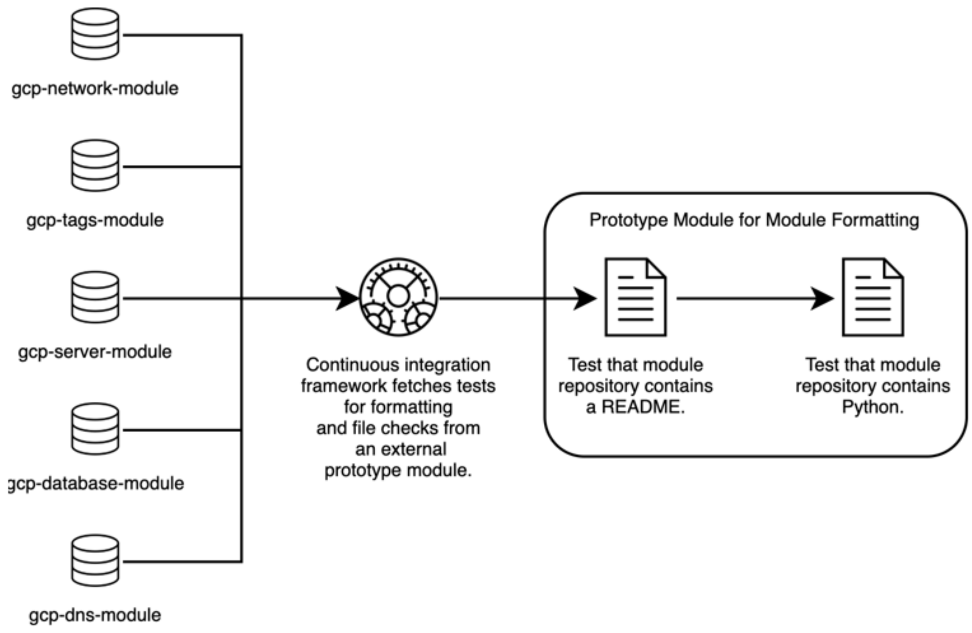

gcp-network-module
gcp-tags-module
gcp-server-module
gcp-database-module
gcp-dns-module
Continuous integration framework fetches tests for formatting and file checks from an external prototype module.
Prototype Module for Module Formatting
Test that module repository contains a README.
Test that module repository contains Python.

Figure 5.7. Creating a prototype module that contains all of the checks for module repository format will help fix older repositories that do not conform to new standards.

The prototype module with tests help enforce formatting for older modules that you don't use as often, like DNS. If you want to add a new standard, you update the prototype module with a new test. The next time someone updates a module or configuration, they need to update their module format to conform.

A standardized set of checks helps alleviate the operational burden of finding and replacing files in hundreds of repositories. It distributes the responsibility of updating the module repository to the module's maintainers. For more on module conformance testing and integrating them in your workflow, you can apply the practices in chapters 6, 7, and 8.

A second disadvantage to a multiple repository structure involves the challenge of troubleshooting. When you reference a module in your configuration, you need to search for the module repository to identify which inputs and outputs it needs. The search adds extra effort and time when you debug failures in configuration.

If you have a build system that can handle single repository building requirements, you can use a single repository for everything. However, most build systems do not scale with recursive directory searching. To solve this problem, you can use a combination of single and multiple repositories.

Let’s apply this solution to Datacenter for Veggies. They separate each configuration for different types of fruits and vegetables. Leafy Green uses one repository, while Fruits use another. Both of them reference shared modules for network, tags, database, and DNS.

Figure 5.8 shows that the Fruits team needs a queue but the Leafy Green team does not. As a result, the Fruits repository includes a local module for creating queues. The Fruits team uses a single repository for its unique configurations but references multiple repositories for common modules.

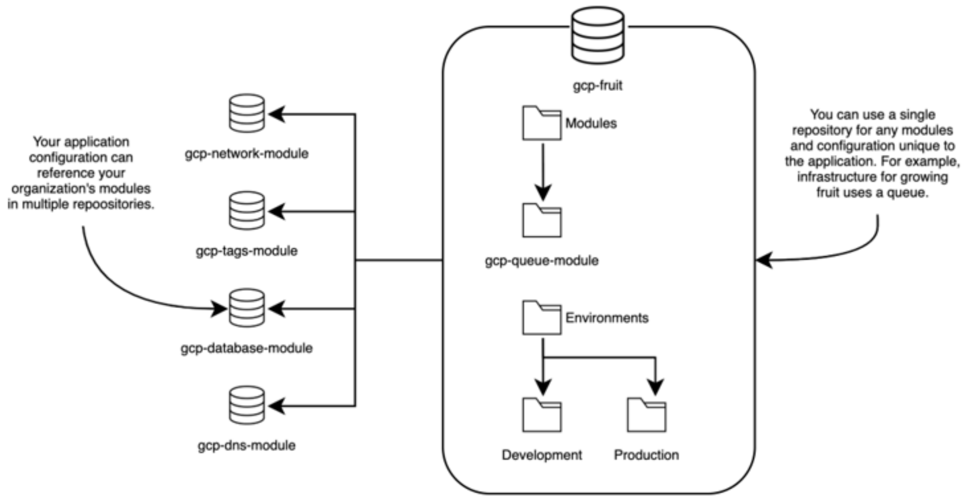

Your application configuration can reference your organization's modules in multiple repoositories.
gcp-network-module
gcp-tags-module
gcp-database-module
gcp-dns-module
gcp-fruit
Modules
gcp-queue-module
Environments
Development
Production
You can use a single repository for any modules and configuration unique to the application. For example, infrastructure for growing fruit uses a queue.

Figure 5.8. Your organization can combine multiple repositories with a single repository for application or system-specific configuration.

When you use this mix-and-match approach, recognize the kind of access control you want for individual repositories or shared configurations. If you want to improve composability and reproducibility for other teams, you might put a module in its own repository. However, if you want to maintain evolvability for a specialized configuration, you might manage the module locally with your configuration.

As you choose your repository structure, recognize the trade-off between approaches and refactor as the number of modules and configurations grows. When you add more configuration and resources into a single repository, you need to make sure the tools and processes scale with it!

5.2 Versioning

Throughout this chapter, you use the practice of keeping infrastructure configuration or code in version control. For example, Datacenter for Veggies teams can always reference infrastructure based on the commit hash. One day, the security team for Datacenter for Veggies expressed concern about the age of usernames and passwords for soil monitoring databases.

They recommend using a secret manager to store and rotate the password every 30 days. Problematically, *all* teams use the soil monitoring database module. Figure 5.9 shows that the application currently references the output of the database module. The module outputs the password for the database, which applications use to write and read data. The security team wants you to use a secrets manager instead.

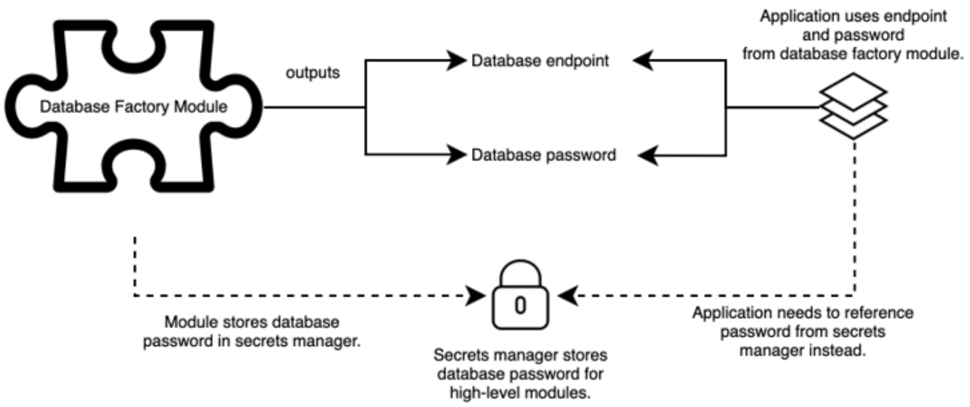

Database Factory Module
outputs
Database endpoint
Database password
Application uses endpoint and password from database factory module.
Module stores database password in secrets manager.
Secrets manager stores database password for high-level modules.
Application needs to reference password from secrets manager instead.

Figure 5.9. The applications reference the database endpoint and password from the soil monitoring module but should use the password from the secrets manager.

The database module output affects the secret's evolvability and security. How can we update the database to use the secrets manager without disrupting soil data collection? The infrastructure team at Datacenter for Veggies decides to add ***versioning*** to the database module.

DEFINITION Versioning is the process of assigning unique versions to iterations of code.

Let’s examine how the Datacenter for Veggies team implements module versions. The team uses version control to tag the current version of the database module as “v1.0.0”. Version “v1.0.0” will output the database password for applications.

```
$ git tag v1.0.0
```

They push the tag for “v1.0.0” to version control.

```
$ git push origin v1.0.0
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
 * [new tag]         v1.0.0 -> v1.0.0
```

You must refactor configurations for Fruit, Leafy Green, Grain, and Herb growth to use the “v1.0.0” version of the database module in a process called *version pinning*. Version pinning preserves the principle of idempotency. When you run the infrastructure as code, the configurations continue to use the database module outputs. You should not detect any drift between a pinned module and the existing infrastructure.

After all of the teams pin the versions to “v1.0.0”, you can now rewrite the module to use a secrets manager. The database module stores the password in the secrets manager. The team tags the new database module as “v2.0.0”, which outputs the database endpoint and location of the password in the secrets manager.

```
$ git tag v2.0.0
```

They push the tag for “v2.0.0” to version control.

```
$ git push origin v2.0.0
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
 * [new tag]         v2.0.0 -> v2.0.0
You can examine the difference between the two versions of the module based on the commit
       history.
$ git log --oneline
7157d3e (HEAD -> main, tag: v2.0.0, origin/main) Change database module to store password
       in secrets manager
5c5fd65 (tag: v1.0.0) Add database factory module
```

Now that you created a new version of the database factory module, you ask some of the teams to try it. The Fruit team bravely volunteers. The Fruit team currently uses version 1.0.0. That module version outputs the database endpoint and password.

When the Fruit team updates to module version 2.0.0 in figure 5.10, they need to account for changes in the module’s workflow. They cannot use the database password in the module’s output. The module outputs an API path to the database password stored in the secrets manager. As a result, the Fruit team refactors their infrastructure as code to get the database password from the secrets manager before creating the database.

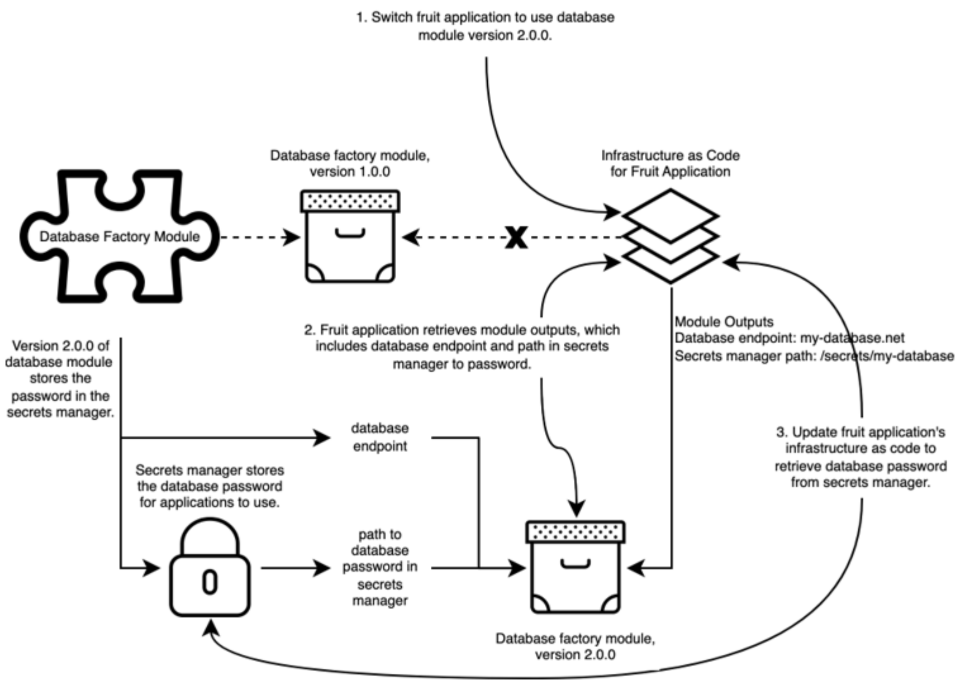

1. Switch fruit application to use database module version 2.0.0.
Database factory module, version 1.0.0
Infrastructure as Code for Fruit Application
Database Factory Module
X
2. Fruit application retrieves module outputs, which includes database endpoint and path in secrets manager to password.
Module Outputs
Database endpoint: my-database.net
Secrets manager path: /secrets/my-database
Version 2.0.0 of database module stores the password in the secrets manager.
database endpoint
3. Update fruit application's infrastructure as code to retrieve database password from secrets manager.
Secrets manager stores the database password for applications to use.
path to database password in secrets manager
0
Database factory module, version 2.0.0

Figure 5.10. You can refactor the Fruit application to reference version 2.0.0 of the database module and retrieve the database password from the secrets manager.

You'll apply a few essential practices to module versioning. First, make sure you run your infrastructure as code and eliminate any drift before you update. Second, establish a versioning approach that *does not* reference the latest version of the module.

Datacenter for Veggies follows *semantic versioning*, assigning version numbers that convey essential information about the configuration. You can specify module versions in a few ways, including tagging the commit with a number in version control or packaging and labeling the module in an artifact repository.

Semantic versioning

I often update the major version for significant updates that remove inputs, outputs, and resources. I usually update the minor version if I update configuration values, inputs, or outputs to modules that do not affect dependencies using previous versions. Finally, I will change the patch version for minor configuration value changes scoped to the module and its resources. For additional details on semantic versioning and its approaches, you can reference https://semver.org/.

Using a consistent versioning approach, you can more effectively evolve downstream infrastructure resources without breaking upstream ones because you control their dependencies. Versioning also helps with the auditing of active versions. To save resources, reduce confusion, and promote the latest changes, versioning allows you to identify and deprecate older, inactive versions of the module.

However, you must continuously remember and enforce certain versioning practices. The longer you wait to update the application to use "v2.0.0" of the database, the higher chance it will fail. You might consider putting a timeline on how you can use the module version "v1.0.0". You do not need to immediately delete "v1.0.0" of the database module. Generally, I upgrade dependent modules within a few minor version changes. Trying to upgrade with a broader "jump" between versions increases the change's risk and possible failure rate.

Versions for patches, support changes, or hotfixes

If you use feature-based development or Git Flow, you can accommodate patches or hotfixes in the same workflow as software development. You can make a branch based on the version tag, update the changes, increase the patch version, and add a new tag for the hotfix branch. You will need to keep the branch for the commit history.

This versioning process works well for a multiple repository structure. What about a single repository? You can still apply the version control tagging approach. You may want to add a prefix to the tag with the module name ("module-name-v2.0.0"). Then, you can package and release your module to an artifact repository. Your build system packages the contents of the module subdirectory and tags the version in the artifact store. Your configuration references the remote modules in the artifact repository instead of a local file.

5.3 Releasing

I explained the practice of module versioning to help with module evolution and minimize disruption to your system. However, you don't want every team to update their infrastructure as code to the newest module immediately. Instead, you want to make sure the module works and doesn't break your infrastructure *before* you use it in production.

Figure 5.11 shows how you evaluated your database module update before allowing all Datacenter for Veggies teams to use it. After you update the database module to store a password in the secrets manager, you push the changes to version control. You asked the Fruit team to *test* the module in a separate environment and confirm the module works. They confirm it works correctly. You tag the release with a new version, 2.0.0, and update the documentation on the secrets manager.

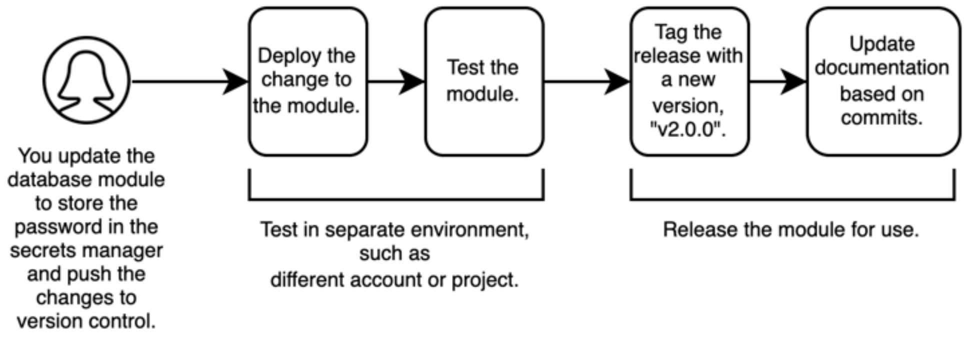

You update the database module to store the password in the secrets manager and push the changes to version control.
Deploy the change to the module.
Test the module.
Tag the release with a new version, "v2.0.0".
Update documentation based on commits.
Test in separate environment, such as different account or project.
Release the module for use.

Figure 5.11. When you make module updates, ensure you include a testing stage before releasing the module and updating its documentation.

In the previous section, the Datacenter for Veggies infrastructure team updated the module and tested it with the Fruit team's development environment first. Now that the module passed the test, other teams can use the new database module with a secret manager. The team followed a ***release*** process to certify that other teams can use the new module.

> **DEFINITION** Releasing is the process of distributing software to a consumer.

A release process identifies and isolates any problems from module updates. You do not package a new module unless the tests certify that it works.

I recommend running module tests in a dedicated testing environment away from development and production workloads. A separate account or project for module testing helps you track the cost of running the tests and isolates failures away from active environments. You'll learn more about testing and testing environments in chapter 6

Code listing for releasing modules

For a detailed code listing of a continuous delivery pipeline for releasing modules, check out github.com/joatmon08/gcp-server-module/blob/v0.0.1/.github/workflows/module.yml. The GitHub Actions pipeline automatically builds a GitHub release when the tests succeed based on a commit message.

After testing the modules, you tag the release with a new version for your team to use. Datacenter for Veggies releases the database module as version "v2.0.0" and uses the Python package manager to reference the tag. Alternatively, you can package the module and push it to an artifact repository or storage bucket.

For example, imagine Datacenter for Veggies has some teams that use Amazon Web Services (AWS) CloudFormation. These teams prefer to reference modules (or

CloudFormation stacks) stored in an AWS S3 bucket. In figure 5.12, the teams add a step to their delivery pipeline to compress their modules and upload them to an AWS S3 storage bucket. As a last step, they update documentation outlining the changes they made.

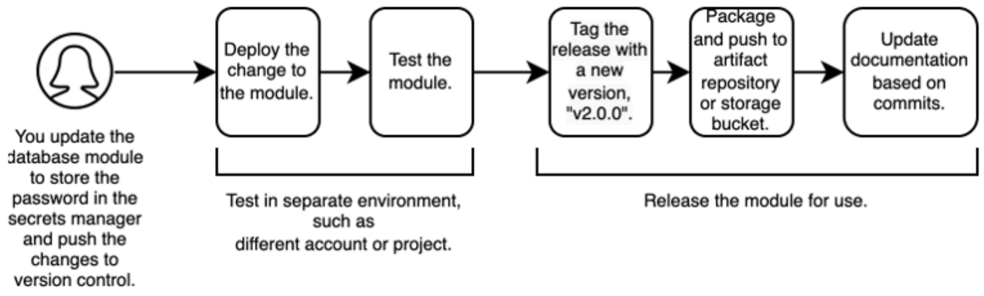

Deploy the change to the module.
Test the module.
Tag the release with a new version, "v2.0.0".
Package and push to artifact repository or storage bucket.
Update documentation based on commits.
You update the database module to store the password in the secrets manager and push the changes to version control.
Test in separate environment, such as different account or project.
Release the module for use.

Figure 5.12. After testing, you optionally choose to package and push the module to an artifact repository or storage bucket.

Some organizations prefer packaging the artifact and storing it in a separate repository for additional security control. If you have a secure network that cannot access an external version control endpoint, you can reference the artifact repository instead. Just make sure to keep the tag in version control so someone can correlate the artifact to the correct code version.

After packaging and pushing the artifact, you should update documentation outlining your changes. Thai documentation, called ***release notes***, outlines breaking changes to outputs and inputs. Release notes communicate a summary of changes to other teams.

> **DEFINITION** Release notes list changes to code for a given release. You should store them in a document in the repository, often called a changelog.

You can manually update the release notes, but I prefer an automated semantic release tool (such as semantic-release) to examine the commit history and build release notes for me. Make sure that you use the correct commit message format for the tool to match and parse changes. Chapter 2 emphasized the importance of writing descriptive commit messages. You'll also find them helpful for an automated release tool.

For example, the database module stores the password in a secret manager. Datacenter for Veggies considers this major feature, so you prefix the commit message with "feat".

```
$ git log -n 1 --oneline
1b65555 (HEAD -> main, tag: v0.0.1, origin/main, origin/HEAD) feat(security): store
       password in secrets manager
```

A commit analyzer in an automated release tool automatically updates the major version of the tag to "v2.0.0" based on this commit.

Image building

You might encounter the practice of building immutable server or container images using image-building tools. By baking the packages you want into a server or container image, you can create new servers with updates without the problems of in-place updates. When you release immutable images, use a workflow to create a test server based on the image, check if it runs correctly, and update the version of the image tag. We'll cover some of these workflows in Chapter 7.

Besides updating release notes, make sure you update commonly used files and documentation. Common files help your teammates use the module. For instance, Datacenter for Veggies agrees that teams must always include a ***README***. A README documents the purpose, inputs, and outputs of each module. You can write a test or use a linting rule to check for the existence of a README file.

DEFINITION A README is a document in a repository that explains usage and contribution instructions for code. For infrastructure as code, use it to document a module's purpose, inputs, and outputs.

In the Python examples, the modules include common files like `__init__.py` for identifying the package and `setup.py` for module configuration. I often refer to files with configuration or metadata that help specific tools or languages as *helper files*. They change depending on the tool and platform you use. You will want to standardize them across your organization so you can change or search them in parallel using automation.

5.4 Sharing modules

As Datacenter for Veggies grows more produce, they add *new teams* that automate the growth of Grains, Tea, Coffee, and Beans. They also create a new team for researching wild strains of produce. Each team needs to be able to expand the existing modules but also create new ones.

For example, the Beans team needs to change a database module to use PostgreSQL version 12. Should they be able to edit the module with the version update? Or should they file a ticket with you, the infrastructure team, to update it?

You need to empower different teams to create and update modules with infrastructure as code. However, you want to make sure that teams do not change an attribute and compromise security or functionality. You'll find a few practices that can help you share modules across your organization.

Imagine all teams in Datacenter for Veggies need a database. You create a new, opinionated database module that establishes a default set of parameters to provide security and functionality. The database module uses embedded defaults for module inputs to cover many Datacenter for Veggies use cases. Even if the Coffee team doesn't know how to create a database, they can use the module to build a secure, working database.

As a general practice, set *opinionated defaults* in your module. You want to err on the prescriptive side. If a team needs more flexibility, they can update the module or override the default attributes. Preset defaults help teach secure and standard practices for deploying specific infrastructure resources.

In the scenario, the Beans team expresses that they need more flexibility. The module does not use a new version of the database, PostgreSQL version 12. No other team uses that version of PostgreSQL. The Beans team decides to update the database version and push the changes into the repository.

However, the changes do not get released immediately. The build system sends a notification to module approvers in the infrastructure team. In Figure 5.13, the infrastructure team pauses the build system and reviews the changes. If the changes pass their approval, the build system releases the module. The Beans teams can use the new version of the database module with PostgreSQL version 12.

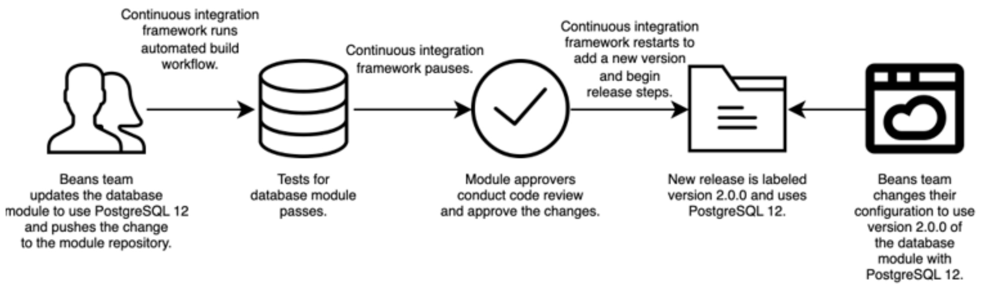

Continuous integration framework runs automated build workflow.
Continuous integration framework pauses.
Continuous integration framework restarts to add a new version and begin release steps.
Beans team updates the database module to use PostgreSQL 12 and pushes the change to the module repository.
Tests for database module passes.
Module approvers conduct code review and approve the changes.
New release is labeled version 2.0.0 and uses PostgreSQL 12.
Beans team changes their configuration to use version 2.0.0 of the database module with PostgreSQL 12.

Figure 5.13. An application team can update the database module themselves. However, they must wait for approval from subject matter experts before they can use the new release.

Why should you allow the Beans team to change the infrastructure module? *Self-service* of module changes empowers all teams to update their systems and reduce the burden on infrastructure and platform teams. You want to balance their development progress with security and infrastructure availability. Adding an approval before module release identifies potential failures or non-standard changes to infrastructure.

The practice of allowing any team to use modules and edit them with approvers works best with established module development standards and processes. If you don't establish module standards, this approach falls apart and adds friction to delivering the infrastructure change to production.

Let's return to the example. The infrastructure team does not have much confidence in the change, so they ask a database administrator for additional review. The database administrator points out that if the Beans team upgrades their module version, the resulting behavior deletes the previous database and creates an empty one with the new version! This would significantly disrupt the application supporting bean growth!

In figure 5.14, the Beans team submits a request for help from the database team. An administrator recommends some practices that will help update the database without deleting data. The Beans team implements these practices and asks module approvers for a second review. Once the module gets released, they can use the module without worrying about disrupting their applications.

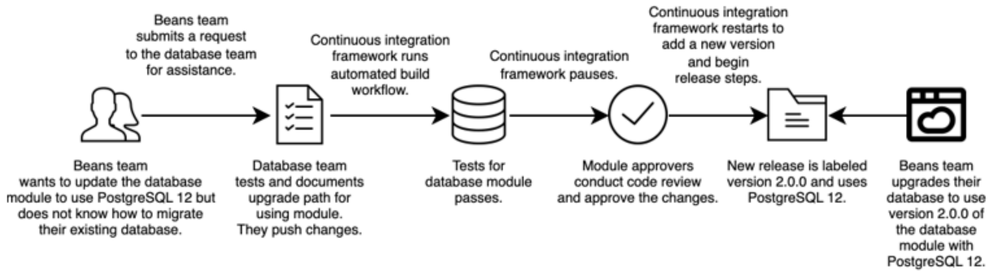

Beans team submits a request to the database team for assistance.
Continuous integration framework runs automated build workflow.
Continuous integration framework pauses.
Continuous integration framework restarts to add a new version and begin release steps.
Beans team wants to update the database module to use PostgreSQL 12 but does not know how to migrate their existing database.
Database team tests and documents upgrade path for using module. They push changes.
Tests for database module passes.
Module approvers conduct code review and approve the changes.
New release is labeled version 2.0.0 and uses PostgreSQL 12.
Beans team upgrades their database to use version 2.0.0 of the database module with PostgreSQL 12.

Figure 5.14. For disruptive module updates, the application team submits a ticket to the database team to verify database migration steps before releasing a new module version.

If you have concerns that a change might be particularly disruptive to a system's architecture, security, or availability, *ask for review from a subject matter expert* before releasing a new version. A subject matter expert can help identify any problems that will affect other teams using the module and advise on the best way to update the module. The process of review helps you evolve your infrastructure as code and identify potential failures from infrastructure changes.

In general, you need a process that empowers your team to make infrastructure changes *and* provide them the knowledge and support to complete them successfully without disrupting critical systems. Manual review may seem tedious but helps educate your team and prevent problems in production. Your team must optimize between manual review and governance versus change deployment, something I'll expand on in chapter 7.

By working collaboratively on modules, you share infrastructure as code knowledge across teams and collectively identify potential disruptions to critical infrastructure. You can treat modules as *artifacts* for use across an organization, similar to shared application libraries, container images, or virtual machine images. Anyone in the company can use and update modules (with additional help, if needed!) to evolve infrastructure architecture, security, or availability.

5.5 Summary

- Structure and share modules and configurations in a single repository or multiple repositories.
- A single repository structure organizes all configuration and modules in one place, making it easier to troubleshoot and identify usable resources.
- A multiple repository structure organizes all configuration and modules into their own code repositories, divided by business domain, function, team, or environment.
- A multiple repository structure allows better access control for individual infrastructure configuration or modules and streamlines pipeline execution for each repository.
- A single repository may not scale as more people collaborate on infrastructure as code and require additional resources for a build system to process changes quickly.
- Refactor a single repository into multiple repositories, one for each module.
- Choose a consistent versioning methodology for modules and update them using git tags.
- Package and release a module to an artifact repository, which will allow anyone in the organization to retrieve a specific module version.
- When sharing modules across teams, establish opinionated default parameters in modules to maintain security and functionality.
- Allow anyone in the organization to suggest updates to modules but add governance to identify potentially disruptive changes to modules that affect architecture, security, or infrastructure availability.

6 Testing

This chapter covers

- **Identifying which type of tests to write for infrastructure systems**
- **Writing tests to verify infrastructure configuration or modules**
- **Understanding the cost of different types of tests**

Recall from Chapter 1 that infrastructure as code involves an entire process to push a change to a system. You update scripts or configurations with infrastructure changes, push them to a version control system, and apply the changes in an automated way. However, you can use every module and dependency pattern from chapters 3 and 4 and still have failed changes! How do you catch a failed change before you apply it to production?

You can solve this problem by implementing tests for infrastructure as code. ***Testing*** is a process that evaluates whether or not a system works as expected. This chapter reviews some considerations and concepts related to testing infrastructure as code to reduce the rate of change failure and build confidence in infrastructure changes.

> **DEFINITION** Testing infrastructure as code is a process that evaluates whether or not infrastructure works as expected.

Imagine you configure a network switch with a new network segment. You can manually test existing networks by pinging each server on each network and verifying their connectivity. To test if you set up the new network correctly, you create a new server and check if it responds when you connect to it. This manual test takes a few hours for two or three networks.

As you create more networks, you can take days to verify your network connectivity. For every network segment update, you must manually verify the network connectivity and the servers, queues, databases, and other resources running on the network. You *cannot* test everything, so you only check a few resources. Unfortunately, this approach can leave hidden bugs or issues that only appear weeks, even months, later!

To reduce the burden of manual testing, you can instead *automate* your tests by scripting each command. Your script creates a server on the new network, checks its connectivity, and tests connections to existing networks. You invest some time and effort into writing the tests but save hours of manual verification by running an automated script for any subsequent changes to the network.

Figure 6.1 shows the amount of effort in hours compared to the number of infrastructure resources when you do manual and automated testing. When you run the network tests manually, you have to spend a lot of time on testing. The effort increases the more resources you add to your system. By comparison, writing automated tests takes an initial effort. However, the effort to maintain the test generally decreases as your system grows. You can even run automated tests in parallel to reduce the overall testing effort.

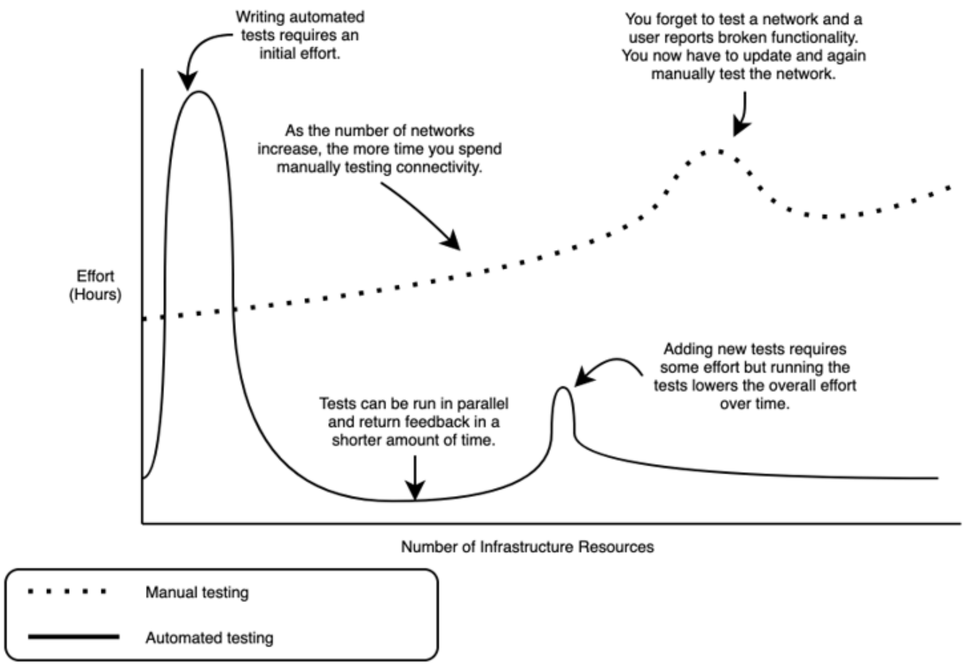


Figure 6.1. Manual testing may have lower effort initially, but as the number of infrastructure resources in your system increases, it increases in effort. Automated testing takes a high initial effort but decreases as you grow your system.

Of course, testing doesn't catch every problem or eliminate all failures from your system. However, automated testing serves as documentation for what you should test in your system every time you make a change. If a hidden bug chooses to appear, you spend some time writing a new test to verify the bug doesn't happen again! Tests lower the overall operational effort over time.

You can use testing frameworks for your infrastructure provider or tool or native testing libraries in programming languages. The code listings use a Python testing framework called `pytest` and Apache Libcloud, a Python library to connect to Google Cloud Platform. I tried to write the tests to focus on *what the test verifies* and not the syntax. You can apply the general approach to any tool or framework.

For more on pytest and Apache Libcloud

To run the tests, refer to the code repository at https://github.com/joatmon08/manning-book for instructions, examples, and dependencies. It includes some links and references for getting started with pytest and Apache Libcloud.

Do not write tests for every single bit of infrastructure as code in your system. Tests can become difficult to maintain and, on occasion, redundant. Instead, I'll explain how to assess when to write a test and what type of test applies to the resource you're changing. Infrastructure testing is a heuristic - you're never going to be able to predict or simulate a change to production fully. A helpful test provides insight and practice into configuring infrastructure or how a change will impact a system. I'll also separate which tests apply to *modules* such as factories, prototypes, or builders versus general composite or singleton *configuration* for a live environment.

6.1 The infrastructure testing cycle

Testing helps you gain confidence and assess the impact of changes to infrastructure systems. However, how can you test a system without creating it first? Furthermore, how do you know that your system works after applying changes?

You can use the infrastructure testing cycle in figure 6.2 to structure your testing workflow. After you define an infrastructure configuration, you run some initial tests to check your configuration. If they pass, you can apply the changes to active infrastructure and test the system.

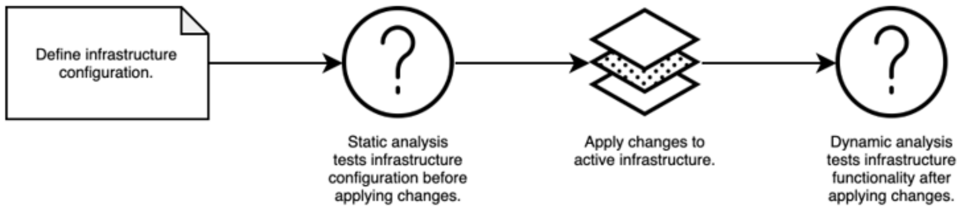

Define infrastructure configuration.
Static analysis tests infrastructure configuration before applying changes.
Apply changes to active infrastructure.
Dynamic analysis tests infrastructure functionality after applying changes.

Figure 6.2. Infrastructure testing verifies whether or not you can apply changes to a system. After applying changes, you can use additional tests to confirm the changes succeeded.

In this workflow, you run two types of tests. One kind of test statically analyzes the configuration before you deploy the infrastructure changes, and the other dynamically analyzes the infrastructure resource to make sure it still works. Most of your tests follow this pattern by running before and after change deployment.

6.1.1 Static analysis

How would you apply the infrastructure testing cycle to our network example? Imagine you parse your network script to verify that the new network segment has the correct IP address range. You don't need to deploy the changes to the network. Instead, you analyze the script, a static file.

In figure 6.3, you define the network script and run static analysis. If you find the wrong IP address, the tests fail. You can revert or fix your network changes and re-run the tests. If they pass, you can apply the correct network IP address to the active network.

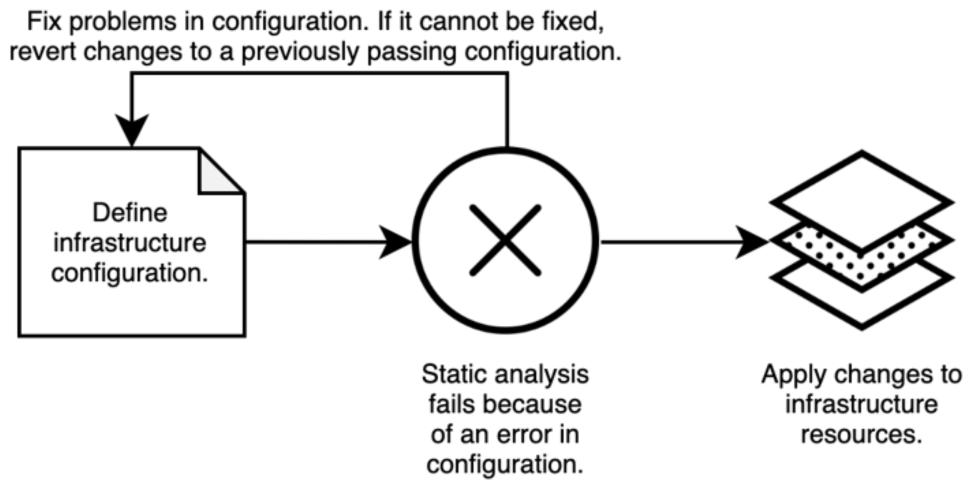

Fix problems in configuration. If it cannot be fixed, revert changes to a previously passing configuration.
Define infrastructure configuration.
Static analysis fails because of an error in configuration.
Apply changes to infrastructure resources.

Figure 6.3. You can either fix the configuration to pass the tests or revert to a previously successful configuration when static analysis fails.

Tests that evaluate infrastructure configuration before deploying changes to infrastructure resources perform ***static analysis***.

> **DEFINITION** Static analysis for infrastructure as code verifies plaintext infrastructure configuration before deploying changes to live infrastructure resources.

Tests for static analysis do not require infrastructure resources since it usually parses the configuration. They do not run the risk of impacting any active systems. If static analysis tests pass, we have more confidence that we can apply the change.

I often use static analysis tests to check for infrastructure naming standards and dependencies. They run before applying changes, and in a matter of seconds, they identify any inconsistent naming or configuration concerns. I can correct the changes, rerun the tests to pass, and apply the changes to infrastructure resources.

Tests for static analysis do not apply changes to active infrastructure, making rollback more straightforward. If tests for static analysis fail, you can return to the infrastructure configuration, correct the problems, and commit the changes again. If you cannot fix the configuration to pass static analysis, you can revert your commit to a previous one that succeeds! You'll learn more about reverting changes in chapter 11.

6.1.2 Dynamic analysis

If the static analysis passes, you can deploy changes to the network. However, you don't know if the network segment actually works. After all, a server needs to connect to the network. To test connectivity, you create a server on the network and run a test script to check inbound and outbound connectivity.

Figure 6.4 shows the cycle of testing network functionality. Once you apply changes to the live infrastructure environment, you run some tests to check the functionality of the system. If the test script fails and shows the server cannot connect, you return to the configuration and fix it for the system.

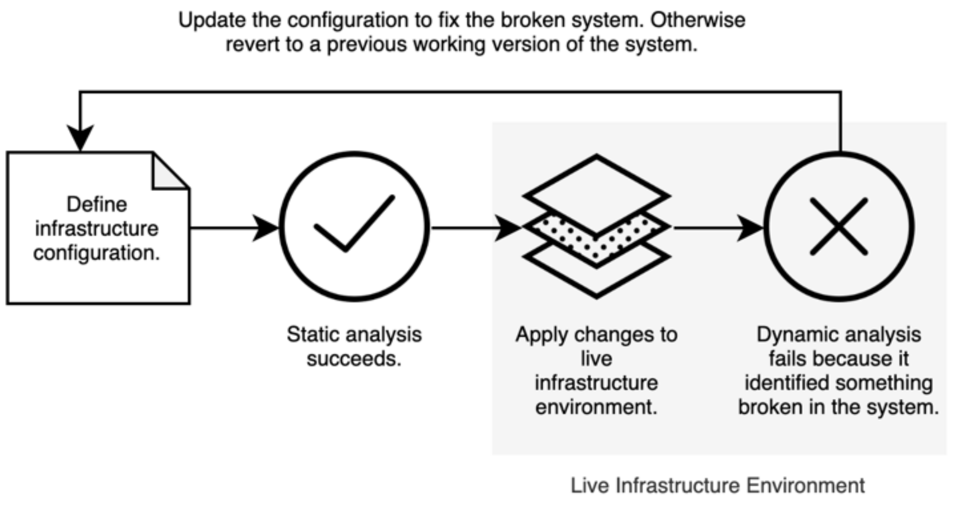

Update the configuration to fix the broken system. Otherwise revert to a previous working version of the system.
Define infrastructure configuration.
Static analysis succeeds.
Apply changes to live infrastructure environment.
Dynamic analysis fails because it identified something broken in the system.
Live Infrastructure Environment

Figure 6.4. When dynamic analysis fails, you can fix the testing environment by updating the configuration or reverting to a previously working configuration.

Note that your testing script needs a live network to create a server and test its connectivity. The tests that verify infrastructure functionality after applying changes to live infrastructure resources perform ***dynamic analysis***.

> **DEFINITION** **Dynamic analysis for infrastructure as code verifies system functionality after applying changes to live infrastructure resources.**

When these tests pass, we have more confidence that the update succeeded. However, if they fail, they identify a problem in the system. If the tests fail, you know that you need to debug, fix the configuration or scripts, and rerun the tests. They provide an early warning system for changes that might break infrastructure resources and system functionality.

You can only dynamically analyze a live environment. What if you don’t know if the update will work? Can you isolate these tests from a production environment? Rather than apply all changes to a production environment and test it, you can use an intermediate testing environment to separate your updates and test them.

6.1.3 Infrastructure testing environments

Some organizations duplicate entire networks in a separate environment so they can test larger network changes. Applying changes to a testing environment makes it easier to identify and fix the broken system, update configuration, and commit the new changes without affecting business-critical systems.

When you run your tests in a separate environment before promoting to the active one, you add to the infrastructure testing cycle. In Figure 6.5, you keep the static analysis step. However, you apply your network change in a testing environment and run dynamic analysis. If it passes the testing environment, you can apply the changes to production and run dynamic analysis in production.

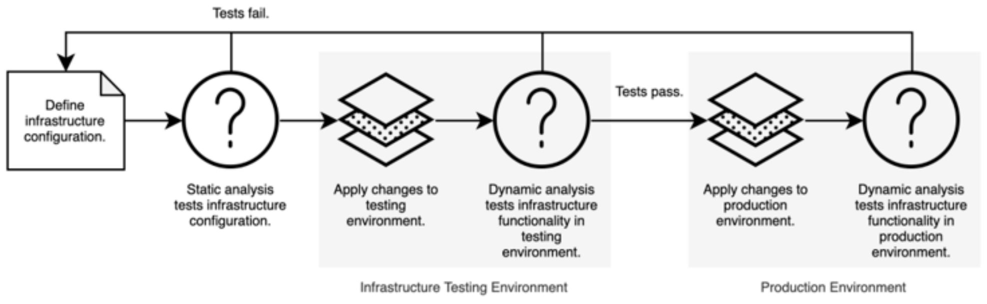

Tests fail.
Define infrastructure configuration.
Static analysis tests infrastructure configuration.
Apply changes to testing environment.
Dynamic analysis tests infrastructure functionality in testing environment.
Tests pass.
Apply changes to production environment.
Dynamic analysis tests infrastructure functionality in production environment.
Infrastructure Testing Environment
Production Environment

Figure 6.5. You can run a static and dynamic analysis of infrastructure in development before applying the changes to production.

A ***testing environment*** isolates changes and tests from the production environment.

> **DEFINITION** A testing environment is an environment that is separate from production and used for testing infrastructure changes.

A testing environment before production helps you *practice* and *check* changes before deploying production. You better understand how they affect existing systems. If you cannot fix the updates, you can revert the testing environment to a working configuration version.

You can use testing environments for the following:

- Examine the effect of an infrastructure change before applying it to a production system
- Isolate testing for infrastructure modules (refer to Chapter 5 for module sharing practices).

However, keep in mind that you have to maintain testing environments like production environments. When possible, an infrastructure testing environment should adhere to the following requirements:

- Its configuration must be as similar to production as possible.
- It must be a different environment from the application’s development environment.
- It must be persistent (i.e., do not create and destroy it each time you test).

In previous chapters, I mentioned the importance of reducing drift across environments. If your infrastructure testing environment duplicates production, you will have more accurate testing behavior. You also want to test infrastructure changes in isolation, away from a development environment dedicated to applications. Once you’ve confirmed that your infrastructure changes have not broken anything, you can push them to the application’s development environment.

It helps to have a persistent infrastructure testing environment. This way, you can test whether or not updates to running infrastructure will potentially affect business-critical systems. Unfortunately, maintaining an infrastructure testing environment may not be practical from a cost or resources standpoint. I’ll outline some techniques for cost management of testing environments in Chapter 12.

In the remainder of the chapter, I’ll discuss the different types of tests that perform static and dynamic analysis and how they fit into your testing environment. Some tests will allow you to reduce your dependency on a testing environment. Others will be critical to assessing the functionality of a production system after changes. Later in this book, I will cover rollback techniques specific to production and incorporate testing into continuous infrastructure delivery.

6.2 Unit tests

I mentioned the importance of running static analysis on infrastructure as code. Static analysis evaluates the files for specific configurations. What kinds of tests can you write for static analysis?

Imagine you have a factory module to create a network named “hello-world-network” and three subnets with IP address ranges in 10.0.0.0/16. You want to verify their network names and IP ranges. You *expect* the subnets to divide the 10.0.0.0/16 range amongst themselves.

As a solution, you can write some tests to check the network name and subnet IP address ranges in your infrastructure as code without creating the network and subnet. This static analysis verifies the configuration parameters for expected values in a matter of seconds.

Figure 6.6 shows that your static analysis consists of several tests run simultaneously. You check the network name, number of subnets, and IP ranges for subnets.

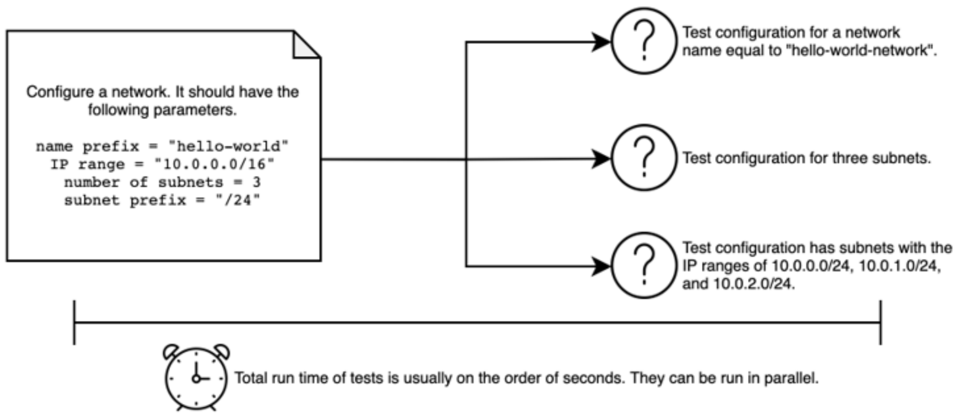

Configure a network. It should have the following parameters.
name prefix = "hello-world"
IP range = "10.0.0.0/16"
number of subnets = 3
subnet prefix = "/24"
Test configuration for a network name equal to "hello-world-network".
Test configuration for three subnets.
Test configuration has subnets with the IP ranges of 10.0.0.0/24, 10.0.1.0/24, and 10.0.2.0/24.
Total run time of tests is usually on the order of seconds. They can be run in parallel.

Figure 6.6. Unit tests verify that a configuration parameter, such as network name, equals an expected value.

We just ran unit tests on the network infrastructure as code. A ***unit test*** runs in isolation and statically analyzes infrastructure configuration or state. These tests do not rely on active infrastructure resources or dependencies and check for the smallest subset of configuration.

DEFINITION Unit tests statically analyze plaintext infrastructure configuration or state. They do not rely on live infrastructure resources or dependencies.

Note that unit tests can analyze metadata in infrastructure configuration *or* state files. Some tools offer information directly in configuration, while others expose values through state. The next few sections provide examples to test both types of files. Depending on your infrastructure as code tool, testing framework, and preference, you may test one, the other, or both.

6.2.1 Testing infrastructure configuration

We'll start by writing unit tests for modules that use templates to generate infrastructure configuration. Our network factory module uses a function to create an object with the network configuration. You need to know if the function "_network_configuration()" generates the correct configuration.

For the network factory module, you can write unit tests in `pytest` to check the functions that generate the JSON configuration for networks and subnets. The testing file includes three tests, one for the network name, the number of subnets, and IP ranges.

Pytest will identify tests by looking for files and tests prefixed by "test_". We named the testing file "test_network.py" so `pytest` can find it. The tests in the file each have the prefix "test_" and some descriptive information on what the test checks.

Listing 6.1 Use pytest to run unit tests in test_network.py

```
import pytest    #A
from main import NetworkFactoryModule    #B

NETWORK_PREFIX = 'hello-world'    #C
NETWORK_IP_RANGE = '10.0.0.0/16'    #C


@pytest.fixture(scope="module")    #D
def network():    #D
   return NetworkFactoryModule(    #D
       name=NETWORK_PREFIX,    #D
       ip_range=NETWORK_IP_RANGE,    #D
       number_of_subnets=3)    #D


@pytest.fixture    #E
def network_configuration(network):    #E
   return network._network_configuration()['google_compute_network'][0]    #E


@pytest.fixture    #F
def subnet_configuration(network):    #F
   return network._subnet_configuration()[    #F
       'google_compute_subnetwork']    #F


def test_configuration_for_network_name(network, network_configuration):    #G
   assert network_configuration[network._network_name][
       0]['name'] == f"{NETWORK_PREFIX}-network"


def test_configuration_for_three_subnets(subnet_configuration):     #H
   assert len(subnet_configuration) == 3


def test_configuration_for_subnet_ip_ranges(subnet_configuration):     #I
   for i, subnet in enumerate(subnet_configuration):
       assert subnet[next(iter(subnet))
                    ][0]['ip_cidr_range'] == f"10.0.{i}.0/24"
```

#A Import pytest, a Python testing library. You need to name the file and tests prefixed with "test_" for pytest to run them.
#B Import the network factory module from main.py. You need to run the method for network configuration.
#C Set expected values as constants, such as network prefix and IP range.
#D Create the network from the module as a test fixture based on expected values. This fixture offers a consistent network object for all tests to reference.
#E Create a separate fixture for the network configuration since you need to parse the "google_compute_network." One test uses this fixture to test the network name.
#F Create a separate fixture for the subnet configuration since you need to parse for the "google_compute_subnetwork." Two tests use this fixture for checking the number of subnets and their IP address ranges.
#G Pytest will run this test to check the configuration for the network name to match "hello-world-network." It references the "network_configuration" fixture.
#H Pytest will run this test to check the configuration for the number of subnets to equal three. It references the "subnet_configuration" fixture.

#I Pytest will check the correct subnet IP range in the network example configuration. It references the “subnet_configuration” fixture.

The testing file includes a static network object passed between tests. This ***test fixture*** creates a consistent network object that each test can reference. It reduces repetitive code used to build a test resource.

> **DEFINITION** A test fixture is a known configuration used to run a test. It often reflects known or expected values for a given infrastructure resource.

Some of the fixtures separately parse the network and subnet information. Any time we add new tests, we don’t have to copy-and-paste the parsing. Instead, we reference the fixture for the configuration.

You can run `pytest` in your command line and pass an argument with a test file. Pytest runs a set of three tests and outputs their success.

```
$ pytest test_network.py
==================== test session starts ====================
collected 3 items

test_network.py ...                                   [100%]

===================== 3 passed in 0.06s =====================
```

In this example, we imported the network factory module, created a network object with configuration, and tested it. You don’t need to write any configuration to a file. Instead, you reference the function and test the object.

This example uses the same approach I take to unit testing application code. It often results in smaller, more modular functions that you can test more efficiently. The function that generates the network configuration needs to output the configuration for the test. Otherwise, the tests cannot parse and compare the values.

6.2.2 Testing domain-specific languages

How do you test your network and subnet configuration if you use a domain-specific language (DSL)? You don’t have functions that you can call in your test. Instead, your unit tests must parse values out of the configuration or dry run file. Both types of files store some kind of plaintext metadata about infrastructure resources.

Imagine you used a DSL instead of Python to create your network. The example creates a JSON file with Terraform-compatible configuration. The JSON file contains all three subnetworks, their IP address ranges, and names. In figure 6.7, you decide to run the unit tests against the network’s JSON configuration file. The tests run quickly because you do not deploy the networks.

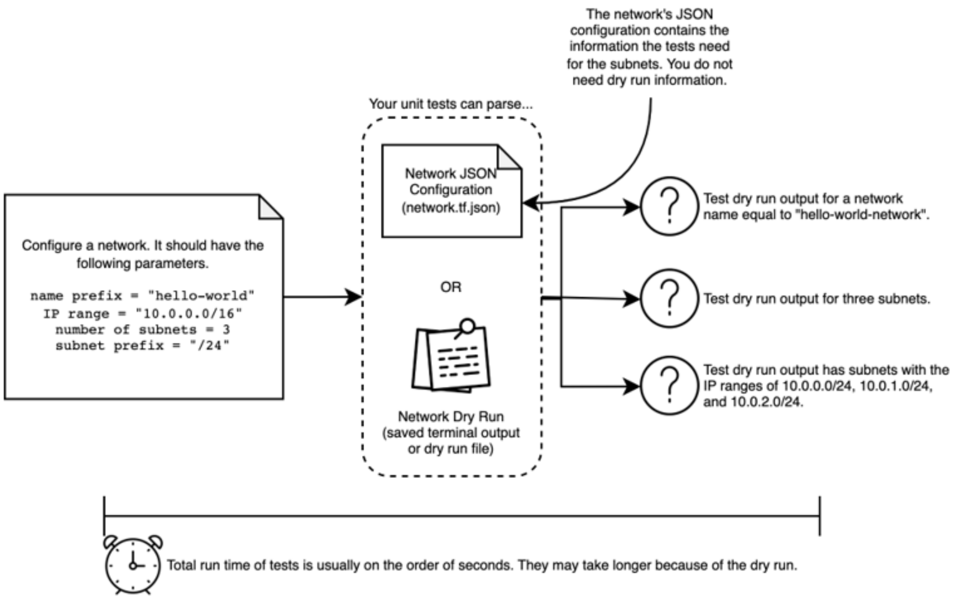

The network's JSON configuration contains the information the tests need for the subnets. You do not need dry run information.
Your unit tests can parse...
Network JSON Configuration (network.tf.json)
OR
Network Dry Run (saved terminal output or dry run file)
Configure a network. It should have the following parameters.
name prefix = "hello-world"
IP range = "10.0.0.0/16"
number of subnets = 3
subnet prefix = "/24"
Test dry run output for a network name equal to "hello-world-network".
Test dry run output for three subnets.
Test dry run output has subnets with the IP ranges of 10.0.0.0/24, 10.0.1.0/24, and 10.0.2.0/24.
Total run time of tests is usually on the order of seconds. They may take longer because of the dry run.

Figure 6.7. Unit tests against dry runs require generating a preview of changes to infrastructure resources and checking it for valid parameters.

In general, you can always unit test the files you used to define infrastructure as code. If a tool uses a configuration file, like AWS CloudFormation, HashiCorp Terraform, Ansible, Puppet, Chef, and more, you can unit test any lines in the configuration.

For example, you can test the network name, number of subnets, and subnet IP address ranges for your network module *without* generating a dry run. I run similar tests with `pytest` to check the same parameters.

Listing 6.2 Use pytest to run unit tests in test_network_configuration.py

```
import json    #A
import pytest

NETWORK_CONFIGURATION_FILE = 'network.tf.json'    #B

expected_network_name = 'hello-world-network'    #C


@pytest.fixture(scope="module")    #D
def configuration():    #D
   with open(NETWORK_CONFIGURATION_FILE, 'r') as f:    #D
       return json.load(f)    #D


@pytest.fixture
def resource():    #E
   def _get_resource(configuration, resource_type):    #E
       for resource in configuration['resource']:    #E
           if resource_type in resource.keys():    #E
               return resource[resource_type]    #E
   return _get_resource    #E


@pytest.fixture    #F
def network(configuration, resource):    #F
   return resource(configuration, 'google_compute_network')[0]    #F


@pytest.fixture    #G
def subnets(configuration, resource):    #G
   return resource(configuration, 'google_compute_subnetwork')    #G


def test_configuration_for_network_name(network):    #H
   assert network[expected_network_name][0]['name'] \    #H
       == expected_network_name    #H


def test_configuration_for_three_subnets(subnets):    #I
   assert len(subnets) == 3    #I


def test_configuration_for_subnet_ip_ranges(subnets):    #J
   for i, subnet in enumerate(subnets):    #J
       assert subnet[next(iter(subnet))    #J
                    ][0]['ip_cidr_range'] == f"10.0.{i}.0/24"    #J
```

#A Import Python's JSON library because you will need to load a JSON file.
#B Set a constant with the expected file name for the network configuration. The tests read the network configuration from "network.tf.json".
#C Set the expected network name to "hello-world-network".
#D Open the JSON file with the network configuration and load it as a test fixture.
#E Create a new test fixture that references the loaded JSON configuration and parses for any resource type. It parses the JSON based on Terraform's JSON resource structure.
#F Get the "google_compute_network" Terraform resource out of the JSON file.
#G Get the "google_compute_subnetwork" Terraform resource out of the JSON file.

#H Pytest will run this test to check the configuration for the network name to match "hello-world-network." It references the "network" fixture.
#I Pytest will run this test to check the configuration for the number of subnets to equal three. It references the "subnet" fixture.
#J Pytest will check for the correct subnet IP range configuration. It references the "subnet" fixture.

You might notice the unit tests for DSLs look similar to those of programming languages. They check the network name, number of subnets, and IP addresses. Some tools have specialized testing frameworks. They usually use the same workflow of generating a dry run or state file and parsing it for values.

However, your configuration file may not contain everything. For example, you won't have certain configurations in HashiCorp Terraform or Ansible until *after* you do a dry run. A ***dry run*** previews infrastructure as code changes without deploying them and internally identifies and resolves potential problems.

> **DEFINITION** A dry run previews infrastructure as code changes without deploying them. It internally identifies and resolves potential problems.

Dry runs come in different formats and standards. Most dry runs output to a terminal, which you can save the output to a file. Some tools will automatically generate the dry run to a file.

Generating dry runs for unit tests

Some tools will save their dry runs in a file, while others output the changes in a terminal. If you use HashiCorp Terraform, you write the Terraform plan to a JSON file using the following command:

```
$ terraform plan -out=dry_run && terraform show -json dry_run > dry_run.json
```

AWS CloudFormation offers change sets, which you can parse the change set description after it completes. Similarly, you can get Kubernetes dry run information with the `kubectl run`'s `--dry-run=client` option.

As a general practice, I prioritize tests that check configuration files. I write tests to parse dry runs when I cannot get the value from configuration files. A dry run typically needs network access to the infrastructure provider API and takes a bit of time to run. On occasion, the output or file contains some sensitive information or identifiers that I do not want a test to explicitly parse.

While dry run configuration may not adhere to the more traditional software development definition of unit tests, the parsing of dry runs does not require any changes to active infrastructure. It remains a form of static analysis. The dry run itself serves as a unit test to validate and output the expected change behavior before applying the change.

6.2.3 When should you write unit tests?

Unit tests help you verify that your logic generates the correct names, produces the correct number of infrastructure resources, and calculates the correct IP ranges or other attributes. Some unit tests may overlap with formatting and linting, concepts I mentioned in Chapter 2. I classify linting and formatting as part of unit testing because they help you understand how to name and organize your configuration.

Figure 6.8 summarizes some use cases for unit tests. You should write additional unit tests to verify any logic you used to generate infrastructure configuration, especially with loops or conditional (if-else) statements. Unit tests can also capture wrong or problematic configurations, such as the wrong operating system.

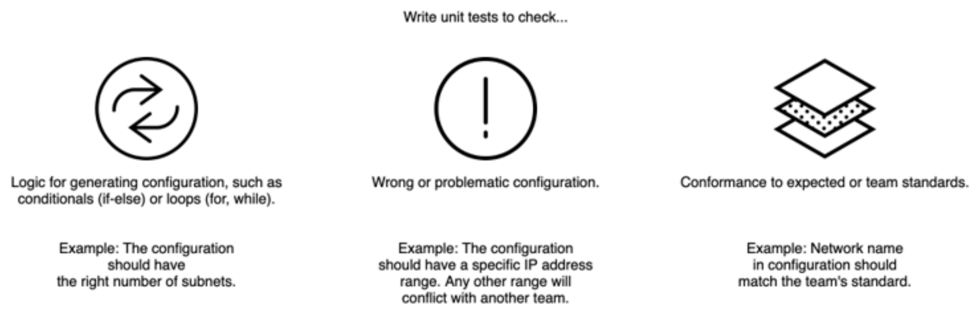

Write unit tests to check...
Logic for generating configuration, such as conditionals (if-else) or loops (for, while).
Wrong or problematic configuration.
Conformance to expected or team standards.
Example: The configuration should have the right number of subnets.
Example: The configuration should have a specific IP address range. Any other range will conflict with another team.
Example: Network name in configuration should match the team's standard.

Figure 6.8. Write unit tests to verify the resource logic, highlight potential problems, or identify team standards.

Since unit tests check the configuration in isolation, they do not precisely reflect how a change will affect a system. As a result, you can't expect a unit test to prevent a major failure during production changes. However, you should still write unit tests! While they won't identify problems while running a change, unit tests can *prevent* problematic configurations before production.

For example, someone might accidentally type a configuration for 1000 servers instead of 10 servers. A test to verify the maximum number of servers in a configuration can prevent someone from overwhelming the infrastructure and manage the cost. Unit tests can also prevent any insecure or non-compliant infrastructure configuration from a production environment. I will cover how to apply unit tests to secure and audit infrastructure configuration in Chapter 8.

In addition to early identification of wrong configuration values, unit tests help automate checking complex systems. When you have many infrastructure resources managed by different teams, you can no longer manually search through one resource list and check each configuration. Unit tests communicate the most critical or standard configurations to other teams. When you write unit tests for infrastructure modules, you verify that the internal logic of the module produces the expected resources.

Unit testing your automation

Good unit tests require an entire book to describe! I scoped my explanation in this section to testing infrastructure configuration. However, you might write a custom automation tool directly accessing an infrastructure API. Automation uses a more sequential approach to configure a resource step-by-step (also known as the imperative style). You should use unit tests to check the individual steps and their idempotency. Unit tests should run the individual steps with different prerequisites and check that they have the same result. If you need to access an infrastructure API, you can mock the API responses in your unit tests.

Use cases for unit tests include checking that you've created the expected number of infrastructure resources, pinned specific versions of infrastructure, or used the correct naming standard. Unit tests run quickly and offer rapid feedback at virtually zero cost (after you've written them!). They run on the order of seconds because they do not post updates to infrastructure or require the creation of active infrastructure resources. If you write unit tests to check the output of a dry run, you add a bit of time because of the initial time spent generating the dry run.

6.3 Contract tests

Unit tests verify configuration or modules in isolation, but what about dependencies between modules? In Chapter 4, I mentioned the idea of a contract between dependencies. The output from a module must agree with the expected input to another. You can uses tests to enforce that agreement.

For example, let's create a server on a network. The server accesses the network name and IP address using a facade, which mirrors the name and IP address range of the network. How do you know that the network module outputs the network name and IP CIDR range and not another identifier or configuration?

You use a contract test in figure 6.9 to test that the network module outputs the facade correctly. The facade must contain the network name and IP address range. If the test fails, it shows that the server cannot create itself on the network.

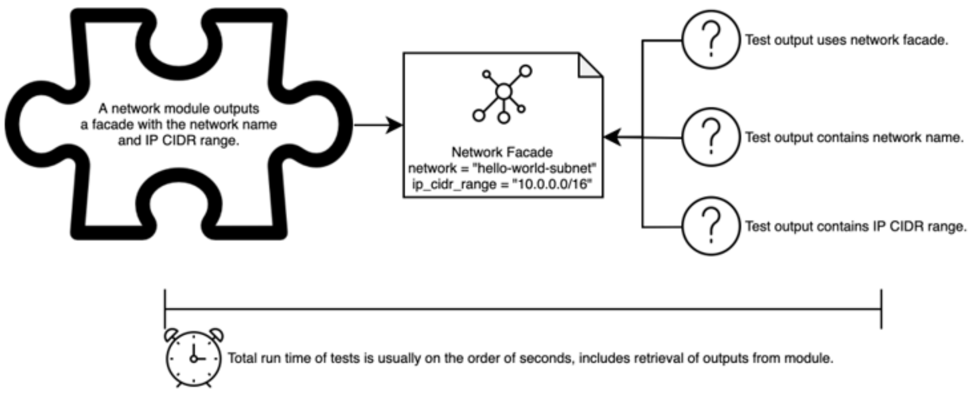

A network module outputs a facade with the network name and IP CIDR range.
Network Facade
network = "hello-world-subnet"
ip_cidr_range = "10.0.0.0/16"
Test output uses network facade.
Test output contains network name.
Test output contains IP CIDR range.
Total run time of tests is usually on the order of seconds, includes retrieval of outputs from module.

Figure 6.9. Contract tests can quickly verify a configuration parameter equals an expected value, such as a network facade with proper outputs.

A ***contract test*** uses static analysis to check that module inputs and outputs match an expected value or format.

> **DEFINITION** Contract tests statically analyze and compare module or resource inputs and outputs to match an expected value or format.

Contract tests help enable evolvability of individual modules while preserving the integration between the two. When you have many infrastructure dependencies, you cannot manually check all of their shared attributes. Instead, a contract test automates the verification of the type and value of attributes between modules.

You'll find contract tests most useful for checking inputs and outputs of heavily parameterized modules (such as factory, prototype, or builder patterns). Writing and running contract tests helps detect wrong inputs and outputs and documents the module's minimum resources. When you do not have contract tests for your modules, you won't find out if you broke something in the system until the next time you apply the configuration to a live environment.

Let's implement a contract test for the server and the network. Using `pytest`, you set up the test by creating a network with a factory module. Then, you verify the network's output includes a facade object with the network name and IP address range. You add these tests to the server's unit tests.

Listing 6.3 Contract test to compare the module outputs with inputs

```
from network import NetworkFactoryModule, NetworkFacade
import pytest

network_name = 'hello-world'    #E
network_cidr_range = '10.0.0.0/16'    #F


@pytest.fixture
def network_outputs():    #A
   network = NetworkFactoryModule(    #B
       name=network_name,    #B
       ip_range=network_cidr_range)    #B
   return network.outputs()    #C


def test_network_output_is_facade(network_outputs):    #D
   assert isinstance(network_outputs, NetworkFacade)    #D


def test_network_output_has_network_name(network_outputs):     #E
   assert network_outputs._network == f"{network_name}-subnet"     #E


def test_network_output_has_ip_cidr_range(network_outputs):     #F
   assert network_outputs._ip_cidr_range == network_cidr_range     #F
```

#A Set up the test with a fixture that uses a network factory module and returns its outputs.
#B Create a network using the factory module with the name and IP address range.
#C Test fixture should return a network facade with different output attributes.
#D Pytest will run this test to check if the module outputs the network facade object.
#E Pytest will run this test to check if the network name matches the expected value, “hello-world”.
#F Pytest will run this test to check the network output’s IP CIDR range matches 10.0.0.0/16.

Imagine you update the network module to output the network ID instead of the name. That breaks the functionality of the upstream server module because the server expects the network name! Contract testing ensures that you do not break the *contract* (or interface) between two modules when you update either one. Use a contract test to verify your facades and adapters when expressing dependencies between resources.

Why should you add the example contract test to the server, a higher level resource? Your server *expects* specific outputs from the network. If the network module changes, you want to detect it from the high-level module first.

In general, a high-level module should defer to changes in the low-level module to preserve composability and evolvability. You want to avoid making significant changes to the interface of a low-level module because it may affect other modules that depend on it.

Domain-specific languages

The example uses Python to verify the module outputs. If you use a tool with a DSL, you might be able to use built-in functionality that allows you to validate that inputs adhere to certain types or regular expressions (such as checking for a valid ID or name formatting). If a tool does not have a validation function, you may need to use a separate testing framework to parse the output types from one module's configuration and compare them to the high-level module inputs.

Infrastructure contract tests require some way to extract the expected inputs and outputs, which may involve API calls to infrastructure providers and verifying the responses against expected values for modules. Sometimes, this involves creating test resources to examine the parameters and understand how fields like ID should be structured. When you need to make API calls or create temporary resources, your contract tests can run longer than a unit test.

6.4 Integration tests

How do you know that you can apply your configuration or module changes to an infrastructure system? You need to apply the changes to a testing environment and *dynamically analyze* the running infrastructure. An ***integration test*** runs against test environments to verify successful changes to a module or configuration.

DEFINITION Integration tests run against testing environments and dynamically analyze infrastructure resources to verify if they are affected by module or configuration changes.

Integration tests require an isolated testing environment to verify the integration of modules and resources. In the next sections, you'll learn about the different integration tests you can write for infrastructure modules and configurations.

6.4.1 Testing modules

Imagine a module that creates a Google Cloud Platform (GCP) server. You want to make sure you can create and update the server successfully. In figure 6.10, you write an integration test. First, configure the server and apply the changes to a testing environment. Then, you run integration tests to check that your configuration update succeeds, create a server, and name it "hello-world-test." The total runtime of the test takes a few minutes because you need to wait for a server to provision.

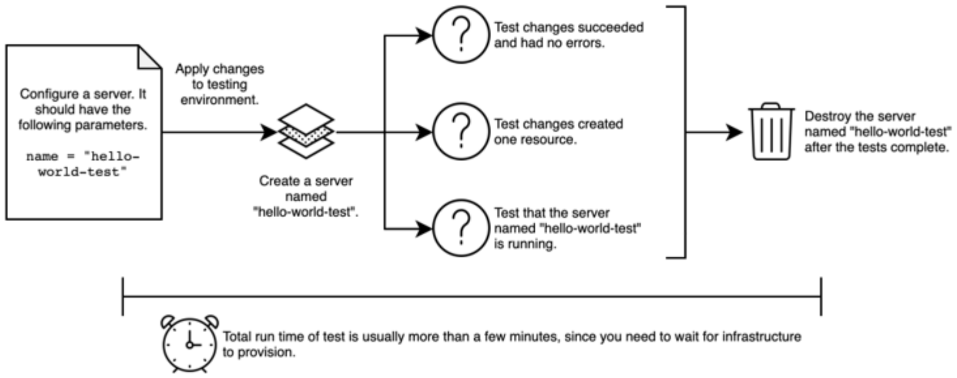

Configure a server. It should have the following parameters.
name = "hello-world-test"
Apply changes to testing environment.
Create a server named "hello-world-test".
Test changes succeeded and had no errors.
Test changes created one resource.
Test that the server named "hello-world-test" is running.
Destroy the server named "hello-world-test" after the tests complete.
Total run time of test is usually more than a few minutes, since you need to wait for infrastructure to provision.

Figure 6.10. Integration tests usually create and update infrastructure resources in a testing environment, test their configuration and status for correctness or availability, and remove them after the tests.

When you implement an integration test, you need to compare the active resource to your infrastructure as code. The active resource tells you whether or not your module deployed successfully. If someone cannot deploy the module, they potentially break their infrastructure.

An integration test must retrieve information about the active resource with the infrastructure provider's API. For example, you can import a Python library to access the Google Cloud Platform (GCP) API in your server module's integration test. The integration test imports Apache Libcloud, a Python library, as a client SDK for the GCP API.

The test builds the server's configuration using the module, waits for the server to deploy, and checks the server's state in the GCP API. If the server returns a "running" status, then the test passes. Otherwise, the test fails and identifies a problem with the module. Finally, the test tears down the test server it created.

Listing 6.4 Integration tests for server creation in test_integration.py

```
from libcloud.compute.types import NodeState    #G
from main import generate_json, SERVER_CONFIGURATION_FILE
import os
import pytest
import subprocess
import test_utils

TEST_SERVER_NAME = 'hello-world-test'


@pytest.fixture(scope='session')    #A
def apply_changes():    #A
   generate_json(TEST_SERVER_NAME)    #B
   assert os.path.exists(SERVER_CONFIGURATION_FILE)    #B
   assert test_utils.initialize() == 0    #C
   yield test_utils.apply()    #C
   assert test_utils.destroy() == 0    #H
   os.remove(SERVER_CONFIGURATION_FILE)    #H


def test_changes_have_successful_return_code(apply_changes):    #D
   return_code = apply_changes[0]
   assert return_code == 0


def test_changes_should_have_no_errors(apply_changes):     #E
   errors = apply_changes[2]
   assert errors == b''


def test_changes_should_add_1_resource(apply_changes):     #F
   output = apply_changes[1].decode(encoding='utf-8').split('\n')
   assert 'Apply complete! Resources: 1 added, 0 changed, 0 destroyed' in output[-2]


def test_server_is_in_running_state(apply_changes):      #G
   gcp_server = test_utils.get_server(TEST_SERVER_NAME)      #G
   assert gcp_server.state == NodeState.RUNNING      #G
```

#A During the test session, use a pytest test fixture to apply the configuration and create a test server on Google Cloud Platform.
#B Generate a Terraform JSON file that uses the server module.
#C Using Terraform, initialize and deploy the server using the Terraform JSON file.
#D Pytest will run this test to verify that the output status of the changes has succeeded.
#E Pytest will run this test to verify that the changes do not return with an error.
#F Pytest will run this test and check that the configuration adds one resource, the server.
#G Pytest uses Apache Libcloud to call the GCP API and get the server's current state. It checks that the server is running.
#H Delete the test server with Terraform and remove the JSON configuration file at the end of the test session.

When you run the tests in this file in your command line, you'll notice that it takes a few minutes because the test session creates the server and deletes it.

```
$ pytest test_integration.py
========================= test session starts =========================
collected 4 items

test_integration.py ....                                          [100%]

==================== 4 passed in 171.31s (0:02:51) =====================
```

The integration tests for the server apply two main practices. First, I wrote tests that follow the sequence of:

- Render configuration, if applicable
- Deploy changes to infrastructure resources
- Run tests, accessing the infrastructure provider's API for comparison
- Delete infrastructure resources, if applicable.

The example implements the sequence using a fixture. You can use it to apply any arbitrary infrastructure configuration and remove it after testing.

Integration tests for configuration management

Integration tests work very similarly for configuration management tooling. For example, you can install packages and run processes on your server. After running the tests, you can expand the server integration tests by checking the server's packages and processes and destroying the server. Rather than writing the tests using a programming language, I recommend evaluating specialized server testing tooling that logs into the server and runs tests against the system.

Second, I ran module integration tests in a separate ***module testing environment*** (such as a test account or project) away from testing or production environments supporting applications. To prevent conflicts with other module tests in the environment, I label and name the resources based on the specific module type, version, or commit hash.

> **DEFINITION** A module testing environment is an environment that is separate from production and used for testing module changes.

Testing modules in a different environment than a testing or production environment helps isolate failed modules from an active environment with applications. You can also measure and control your infrastructure cost from testing modules. I'll cover the cost of cloud computing in greater detail in Chapter 12.

6.4.2 Testing configuration for environments

Integration tests for infrastructure modules can create and delete resources in a testing environment, but integration tests for environment configurations cannot. Imagine you need to add an A record to your current domain name configured by a composite or singleton configuration. How do you write some integration tests to check if you added the record correctly?

You encounter two problems. First, you cannot simply create and then destroy DNS records as part of your integration tests because it may affect applications. Second, the A record depends on a server IP address to exist before you can configure the domain.

Instead of creating and destroying the server and A record in a testing environment, you run the integration tests against a *persistent* testing environment that matches production. In figure 6.11, you update the DNS record in infrastructure as code for the testing environment. Your integration tests that the DNS in the testing environment matches the expected correct DNS record. After the test passes, you can update the DNS record for production.

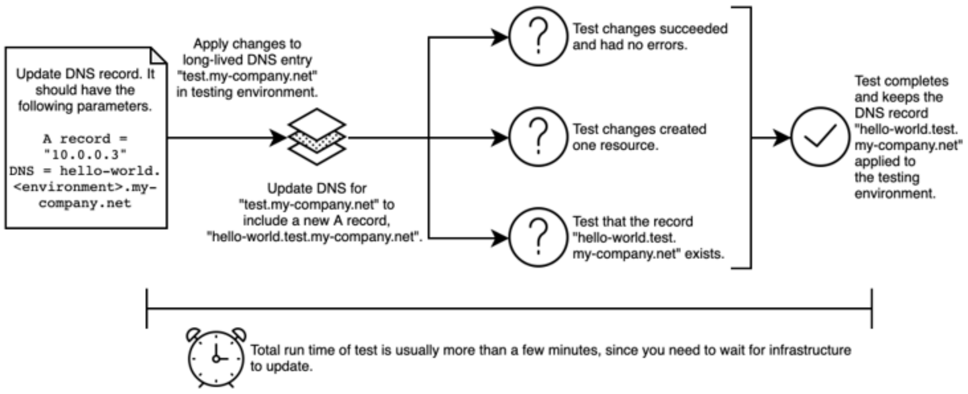

Update DNS record. It should have the following parameters.
A record = "10.0.0.3"
DNS = hello-world.<environment>.my-company.net
Apply changes to long-lived DNS entry "test.my-company.net" in testing environment.
Update DNS for "test.my-company.net" to include a new A record, "hello-world.test.my-company.net".
Test changes succeeded and had no errors.
Test changes created one resource.
Test that the record "hello-world.test.my-company.net" exists.
Test completes and keeps the DNS record "hello-world.test.my-company.net" applied to the testing environment.
Total run time of test is usually more than a few minutes, since you need to wait for infrastructure to update.

Figure 6.11. You can run integration tests against a testing environment with long-lived resources to isolate the changes from production and reduce the dependencies you need to create for the test.

Why run the DNS test in a *persistent* testing environment? First, it can take a long time to create a testing environment. As a high-level resource, DNS depends on many low-level ones. Second, you want an accurate representation of how the change behaves before you update production.

The testing environment captures a subset of dependencies and complexities of the production system so you can check that your configuration works as expected. Keeping similar testing and production environments means that a change in testing provides an accurate perspective of its behavior in production. You want to aim for early detection of problems in the testing environment.

6.4.3 Testing challenges

Without the integration tests, you would not know if a server module or DNS record updates successfully until you manually check it. They expedite the process of verifying that your infrastructure as code works. However, you will encounter a few challenges with integration testing.

You might have difficulty determining which configuration parameters to test. Should you write integration tests to verify every configuration parameter you've configured in infrastructure as code matches the live resource? Not necessarily!

Most tools already have *acceptance tests* that create a resource, update its configuration, and destroy the resource. Acceptance tests certify that the tool can release new code changes. These tests must pass in order for the tool to support changes to infrastructure.

You don't want to spend additional time or effort writing tests that match the acceptance tests. As a result, your integration tests should cover whether or not *multiple* resources have the correct configuration and dependencies. If you write custom automation, you will need to write integration tests to create, update, and delete resources.

Another challenge involves deciding whether or not you should create or delete resources during each test or run a persistent testing environment. Figure 6.12 shows a decision tree for whether or not you should create, delete, or use a persistent testing environment for an integration test.

In general, if a configuration or module does not have too many dependencies, you can create, test, and delete it. However, if your configuration or module takes time to create or requires the existence of many other resources, you will need to use a persistent testing environment.

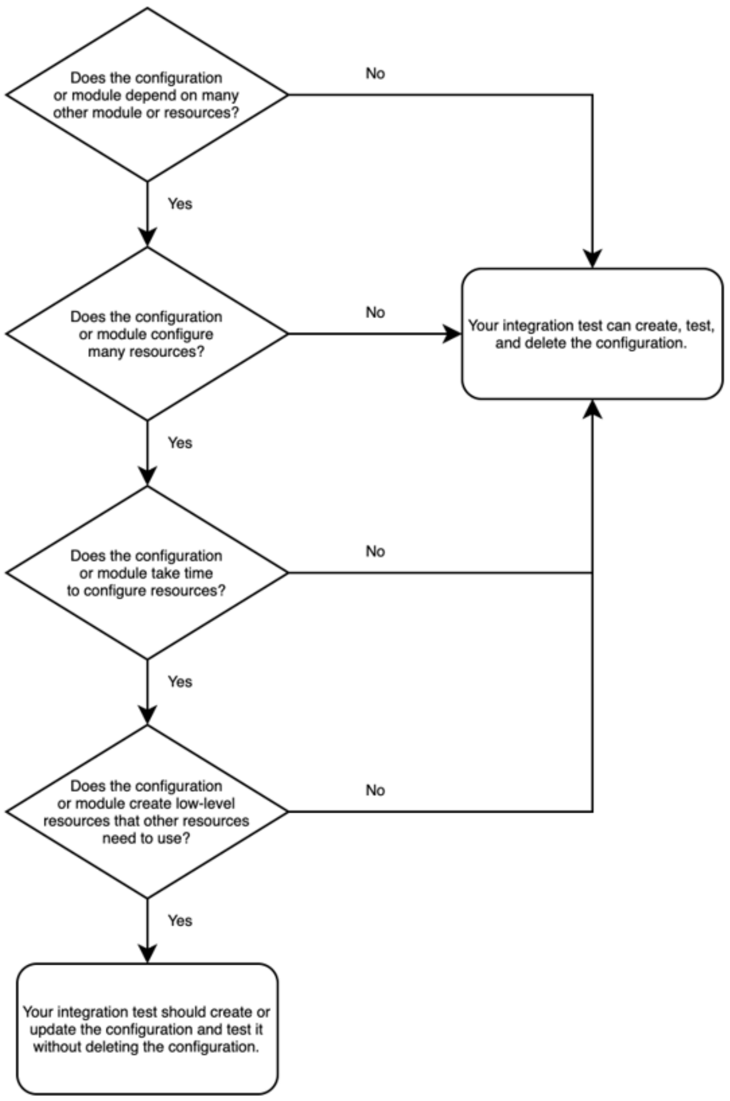

Does the configuration or module depend on many other module or resources?
No
Yes
Does the configuration or module configure many resources?
No
Your integration test can create, test, and delete the configuration.
Yes
Does the configuration or module take time to configure resources?
No
Yes
Does the configuration or module create low-level resources that other resources need to use?
No
Yes
Your integration test should create or update the configuration and test it without deleting the configuration.

Figure 6.12. Your integration test should create and delete resources based on module or configuration type and dependencies.

Not all modules benefit from a create-and-delete approach in integration testing. I recommend running integration tests for low-level modules, such as networks or DNS, and avoid removing the resources. These modules usually require in-place updates in environments with a minimal financial cost. I often find it more realistic to test the update instead of creating and deleting the resource.

Resources created by integration tests for mid-level modules, such as workload orchestrators, may be persistent or temporary depending on the size of the module and resource. The larger the module, the more likely it will need to be long-lived. You can run integration tests for high-level modules, such as application deployments or SaaS, and create and delete the resources each time.

A persistent testing environment does have its limits. Integration tests tend to take a long time to run because it takes time to create or update resources. As a rule, keep modules smaller with fewer resources. This practice reduces the amount of time you need for a module integration test.

Even if you keep configurations and modules small with few resources, integration tests often become the culprit of your infrastructure provider bill's increasing cost. A number of tests need long-lived resources like networks, gateways, and more. Weigh the cost of running an integration test and catching problems against the cost of misconfiguration or a broken infrastructure resource.

You may consider using infrastructure mocks to lower the cost of running an integration test (or any test). Some frameworks replicate an infrastructure provider's APIs for local testing. I do not recommend relying heavily on mocks. Infrastructure providers change APIs frequently and often have complex errors and behaviors, which mocks do not often capture. In chapter 12, I discuss some techniques to manage the cost of testing environments and avoid mocks.

6.5 End-to-end tests

While integration tests dynamically analyze configuration and catch errors during resource creation or update, they do not indicate whether or not an infrastructure resource is *usable*. Usability requires that you or a team member use the resource as intended.

For example, you might use a module to create an application, called a service, on Google Cloud Platform (GCP) Cloud Run. GCP Cloud Run deploys any service in a container and returns a URL endpoint. Your integration tests pass, indicating that your module correctly creates the service resource and permissions to access the service.

How do you know if someone can access the application URL? Figure 6.13 shows how to check if the service endpoint works. First, you write a test to retrieve the application URL as an output from your infrastructure configuration. Then, you make an HTTP request to the URL. The total run time takes a few minutes, most of it from creating the service.

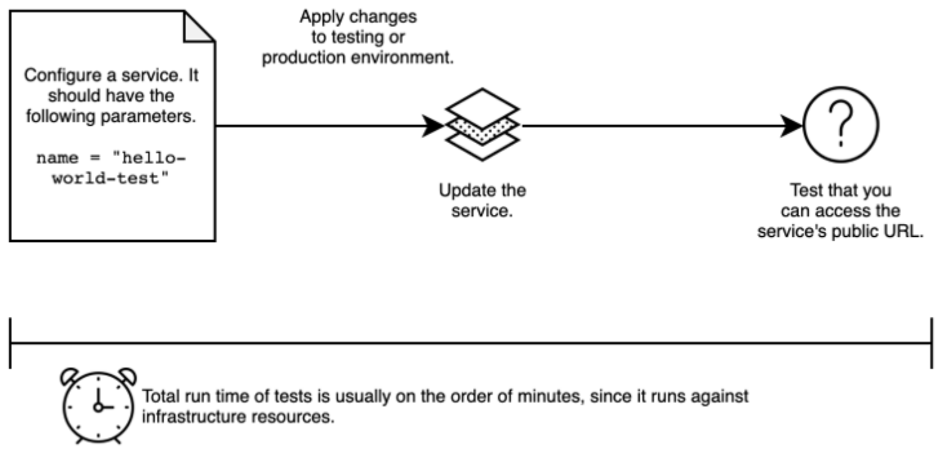

Configure a service. It should have the following parameters.
name = "hello-world-test"
Apply changes to testing or production environment.
Update the service.
Test that you can access the service's public URL.
Total run time of tests is usually on the order of minutes, since it runs against infrastructure resources.

Figure 6.13. End-to-end tests verify the end user's workflow by accessing the webpage at the application's URL.

You created a test for dynamic analysis that differs from an integration test called an ***end-to-end test***. It verifies the end-user functionality of the infrastructure.

> **DEFINITION** End-to-end tests dynamically analyze infrastructure resources and end-to-end system functionality to verify if they are affected by infrastructure as code changes.

The example end-to-end test verifies the end-to-end workflow of the end user accessing the page. It does *not* check for the successful configuration of infrastructure.

End-to-end tests become vital for ensuring that your changes don't break upstream functionality. For example, I might accidentally update a configuration that allows authenticated users to access the GCP Cloud Run service URL. My end-to-end test fails after applying the change, indicating that someone may no longer access the service.

Let's implement an end-to-end test for the application URL in Python. The test for this example needs to make an API request to the service's public URL. It uses a `pytest` fixture to create the GCP Cloud Run service, test the URL for the running page, and delete the service from a testing environment.

Listing 6.5 End-to-end test for GCP Cloud Run service

```
from main import generate_json, SERVICE_CONFIGURATION_FILE
import os
import pytest
import requests    #C
import test_utils

TEST_SERVICE_NAME = 'hello-world-test'


@pytest.fixture(scope='session')     #A
def apply_changes():     #A
   generate_json(TEST_SERVICE_NAME)     #B
   assert os.path.exists(SERVICE_CONFIGURATION_FILE)     #B
   assert test_utils.initialize() == 0     #C
   yield test_utils.apply()     #C
   assert test_utils.destroy() == 0     #G
   os.remove(SERVICE_CONFIGURATION_FILE)     #G


@pytest.fixture     #D
def url():     #D
   output, error = test_utils.output('url')     #D
   assert error == b''     #D
   service_url = output.decode(encoding='utf-8').split('\n')[0]     #D
   return service_url     #D


def test_url_for_service_returns_running_page(apply_changes, url):    #E
   response = requests.get(url)     #E
   assert "It's running!" in response.text     #F
```

#A During the test session, use a pytest test fixture to apply the configuration and create a test service on Google Cloud Platform.
#B Generate a Terraform JSON file that uses the GCP Cloud Run module.
#C Using Terraform, initialize and deploy the service using the Terraform JSON file.
#D Use a pytest fixture to parse the output of the configuration for the service's URL.
#E In the test, make an API request to the service's URL using Python's requests library.
#F In the test, check the service's URL response containing a specific string to indicate the service is running.
#G Destroy the GCP Cloud Run service in the testing environment, so you do not have a persistent service in your GCP project.

AWS equivalents

AWS Fargate with EKS roughly equates to GCP Cloud Run.

Note that if you want to run an end-to-end test in production, you do not want to delete the service. You usually run end-to-end tests against existing environments without creating new or test resources. You apply changes to the existing system and run the tests against the active infrastructure resources.

Smoke tests

As a type of end-to-end test, a smoke test provides rapid feedback on whether or not a change has broken critical business functionality. Running all end-to-end tests can take some time, and you need to fix a change failure quickly. If you can run a smoke test first, you can verify that the change has not gone catastrophically and proceed with further testing. As a quality assurance analyst once told me, “If you power up some hardware and it smokes, you know something is wrong. It’s not worth your time to test it further.”

More complex infrastructure systems benefit from end-to-end tests because they become the primary indicator of whether or not a change has affected critical business functionality. As a result, they help test composite or singleton configurations. You do not usually run end-to-end tests on modules unless they have a large number of resources and dependencies.

I write most of my end-to-end tests for network or compute resources. For example, you can write a few tests to check network peering. The tests provision a server on each network and check if the servers can connect.

Another use case for end-to-end tests involves submitting a job to a workload orchestrator and completing it. This test determines whether or not the workload orchestrator functions properly for application deployment. I once included end-to-end tests that issued HTTP requests with varying payloads to ensure upstream services could call each other without disruption, no matter the payload size or protocol.

Outside of network or compute use cases, end-to-end tests can verify the expected behavior of any system. If you use configuration management with a provisioning tool, your end-to-end tests verify that you can connect to the server and run the expected functionality. For monitoring and alerts, you can run end-to-end tests to simulate the expected system behavior, verify that metrics have been collected, and test the triggering of the alert.

However, end-to-end tests are the most expensive tests to execute in terms of time and resources. Most end-to-end tests need every infrastructure resource available to fully evaluate the system. As a result, you may only run end-to-end tests against production infrastructure. You may not run them in a testing environment because it often costs too much money to procure enough resources for the test.

6.6 Other tests

You may encounter other types of tests outside of unit, contract, integration, and end-to-end tests. For example, you want to roll out a configuration change to a production server that reduces memory. However, you don’t know if the memory reduction will affect the overall system.

Figure 6.14 shows that you can check if your change affected the system using system monitoring. Monitoring continuously aggregates metrics on the server’s memory. If you receive an alert that the server’s memory reaches a percentage of its capacity, you know that you may affect the overall system.

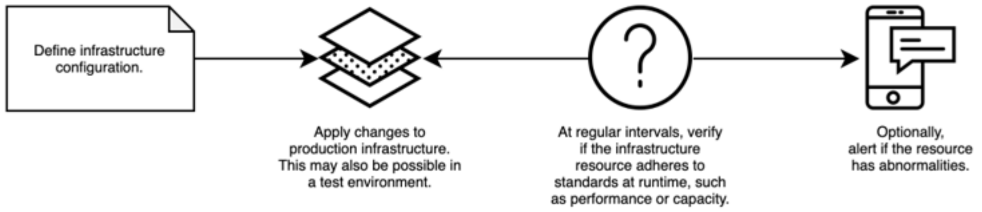

Define infrastructure configuration.
Apply changes to production infrastructure. This may also be possible in a test environment.
At regular intervals, verify if the infrastructure resource adheres to standards at runtime, such as performance or capacity.
Optionally, alert if the resource has abnormalities.

Figure 6.14. Continuous tests run at short intervals to verify that a set of metrics do not exceed a threshold.

Monitoring implements ***continuous testing*** with "tests" to check that metrics do not exceed thresholds run at regular, frequent intervals.

> **DEFINITION** Continuous tests (such as monitoring) run at regular, frequent intervals to check that the current value matches an expected value.

Continuous testing includes monitoring system metrics and security events (when the root user logs into a server). They offer dynamic analysis on an active infrastructure environment. Most continuous tests take the form of alerts, which notify you of any problems.

You may encounter another type of test called a regression test. For example, you may run a test over a period of time to check if your server configuration conforms to your organization's expectations. ***Regression tests*** run regularly but do not have the frequency of monitoring or other forms of continuous testing. You may choose to run them every few weeks or months to check for out-of-band, manual changes.

> **DEFINITION** Regression tests run periodically over an extended period of time to check if infrastructure configuration conforms to the expected state or functionality. They can help mitigate configuration drift.

Continuous and regression tests often require special software or systems to run. They ensure that running infrastructure behaves with expected functionality and performance. These tests also set a foundation for automating a system to respond to anomalies.

For example, systems configured with infrastructure as code and continuous tests can use autoscaling to adjust resources based on the metrics such as CPU or memory. These systems can also implement other self-healing mechanisms, such as diverting traffic to an older version of an application upon errors.

6.7 Choosing tests

I explained some of the most common tests in infrastructure, from unit tests to end-to-end tests. However, do you need to write all of them? Where should you spend your time and effort in writing them? Your *infrastructure testing strategy* will evolve, depending on the complexity and growth of your system. As a result, you will constantly be assessing which tests will help you catch configuration issues before production.

I use a pyramid shape as a *guideline* for infrastructure testing strategy. In figure 6.15, the widest part of the pyramid indicates you should have more of that type of test, while the narrowest part indicates that you should have fewer. At the top of the pyramid are end-to-end tests, which may cost more time and money because they require active infrastructure systems. At the bottom of the pyramid are unit tests, which run in seconds and do not require entire infrastructure systems.

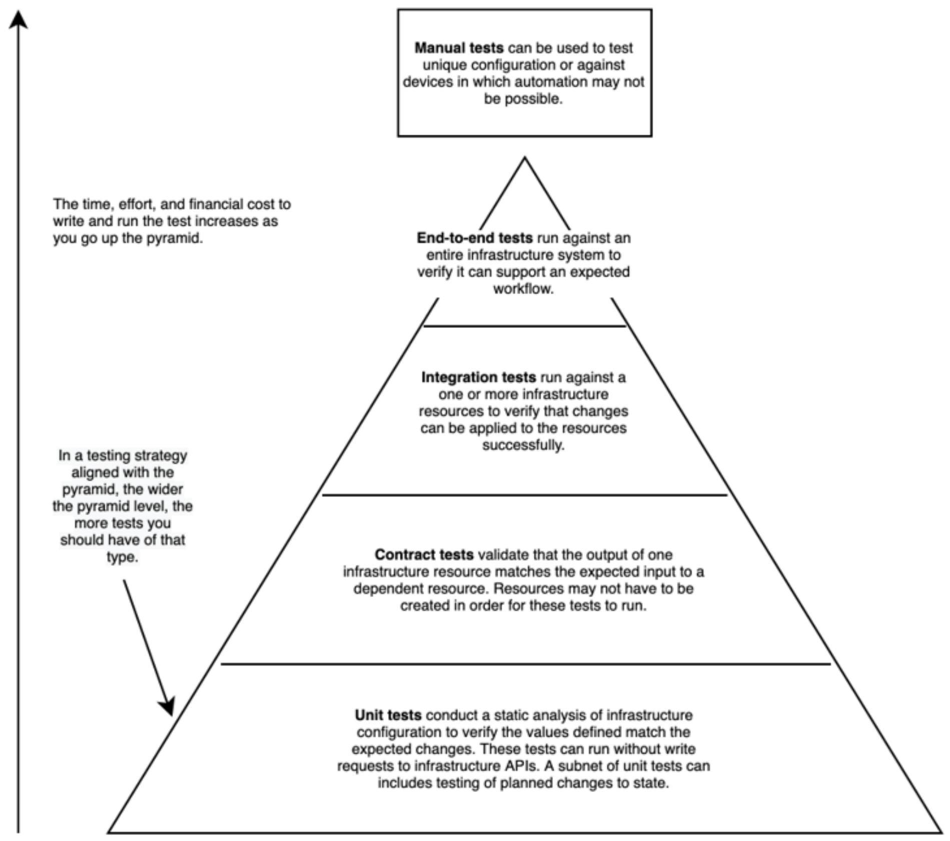

Manual tests can be used to test unique configuration or against devices in which automation may not be possible.
The time, effort, and financial cost to write and run the test increases as you go up the pyramid.
End-to-end tests run against an entire infrastructure system to verify it can support an expected workflow.
Integration tests run against a one or more infrastructure resources to verify that changes can be applied to the resources successfully.
In a testing strategy aligned with the pyramid, the wider the pyramid level, the more tests you should have of that type.
Contract tests validate that the output of one infrastructure resource matches the expected input to a dependent resource. Resources may not have to be created in order for these tests to run.
Unit tests conduct a static analysis of infrastructure configuration to verify the values defined match the expected changes. These tests can run without write requests to infrastructure APIs. A subnet of unit tests can includes testing of planned changes to state.

Figure 6.15. Based on the test pyramid, you should have more unit tests than end-to-end tests because it costs less time, money, and resources to run them.

This guideline, called the ***test pyramid***, provides a framework for different types of tests, their scope, and frequency. I adapted the test pyramid from software testing to infrastructure, modifying it to infrastructure tools and constraints.

> **DEFINITION** The test pyramid serves as a guideline for your overall testing strategy. As you go up the pyramid, the type of test will cost more time and money.

In reality, your test pyramid may be shaped more like a rectangular or pear, sometimes with missing levels. You *will not and should not* write every type of test for every infrastructure configuration. At some point, the tests become redundant and impractical to maintain.

Depending on the system you want to test, it may not be practical to adhere to the test pyramid in its ideal. However, avoid what I jokingly call the "test signpost". A signpost favors many manual tests and not much of anything else.

6.7.1 Module testing strategy

I alluded to the practice of testing modules before releasing them in Chapter 5. Let's return to that example, where you updated a database module to PostgreSQL 12. Rather than manually creating the module and testing to see if it works, you add a series of automated tests. They check for the module's formatting and create a database in an isolated module testing environment.

Figure 6.16 updates the module release workflow with the unit, contract, and integration tests you can add to check that your module works. After the contract tests pass, you run an integration test that sets up the database module on a network and checks if the database runs. After completing the integration test, you delete the test database created by the module and release the module.

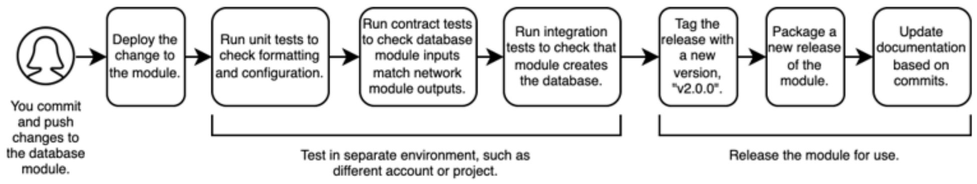

You commit and push changes to the database module.
Deploy the change to the module.
Run unit tests to check formatting and configuration.
Run contract tests to check database module inputs match network module outputs.
Run integration tests to check that module creates the database.
Tag the release with a new version, "v2.0.0".
Package a new release of the module.
Update documentation based on commits.
Test in separate environment, such as different account or project.
Release the module for use.

Figure 6.16. You can break down the testing stage of your module release workflow to include unit, contract, and integration tests.

A combination of unit, contract, and integration tests adequately represents whether or not a module will work correctly. Unit tests check for module formatting and your team's standard configurations. You run them first, so you get fast feedback on any violations in formatting or configurations.

Next, you run a few contract tests. In the case of the database module, you check if the network ID input to the database module matches the output of the network ID from the network module. Catching these mistakes will identify problems between dependencies earlier in your deployment process.

Focus on unit or contract testing to enforce proper configuration, correct module logic, and specific inputs and outputs. The testing workflow outlined in Figure 6.16 works best for modules that use the factory, builder, or prototype patterns. These patterns isolate the smallest subset of infrastructure components and provide a flexible set of variables for your teammates to customize.

Depending on the cost of your development environment, you can write a few integration tests to run against temporary infrastructure resources, which you delete at the end of the test. By investing some time and effort into writing tests for modules with many inputs and outputs, you ensure that changes do not affect upstream configuration and that the module can run successfully on its own.

Exercise 6.1

You notice that a new version of a load balancer module is breaking your DNS configuration. A teammate updated the module to output private IP addresses instead of public IP addresses. What can you do to help your team better remember the module needs public IP addresses?

A. Create a separate load balancer module for private IP addresses.
B. Add module contract tests to verify the module outputs both private and public IP addresses.
C. Update the module's documentation with a note that it needs public IP addresses.
D. Run integration tests on the module and check the IP address is publicly accessible.

Find answers and explanations at the end of the chapter.

6.7.2 Configuration testing strategy

Infrastructure configurations for active environments use more complex patterns like singleton or composite. A singleton or composite configuration has many infrastructure dependencies and often references other modules. Adding end-to-end tests to your testing workflow can help identify issues between infrastructure and modules.

Imagine you have a singleton configuration with an application server on a network. Figure 6.17 outlines each step after you update the size of the server. After pushing the change to version control, you deploy the change to a testing environment. Your testing workflow begins with unit tests to verify formatting and configuration quickly.

Next, you run integration tests to apply changes and verify that the server still runs and has a new size. You complete your verification by testing the entire system using an end-to-end test. The end-to-end test issues an HTTP GET to the application endpoint. Figure 6.17 repeats the process in production to ensure that the system did not break.

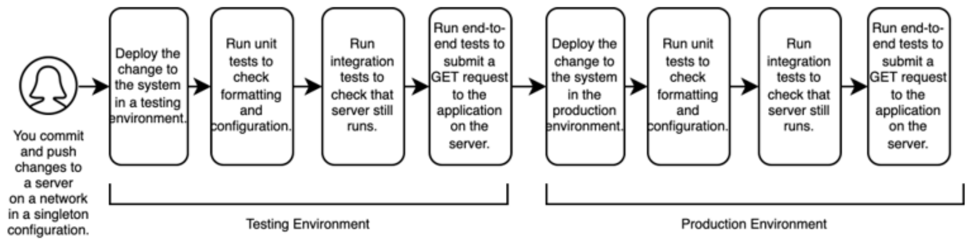

You commit and push changes to a server on a network in a singleton configuration.
Deploy the change to the system in a testing environment.
Run unit tests to check formatting and configuration.
Run integration tests to check that server still runs.
Run end-to-end tests to submit a GET request to the application on the server.
Deploy the change to the system in the production environment.
Run unit tests to check formatting and configuration.
Run integration tests to check that server still runs.
Run end-to-end tests to submit a GET request to the application on the server.
Testing Environment
Production Environment

Figure 6.17. Infrastructure as code using the singleton and composite patterns should run unit, integration, and end-to-end tests in a testing environment before deploying the changes to production.

Just because you created or updated a server successfully does not mean the application it hosts can serve requests! With a complex infrastructure system, you need additional tests to verify dependencies or communication between infrastructure. End-to-end tests can help preserve the functionality of the system.

Repeating the same tests between testing and production environments offers quality control. If you have any configuration drift between testing and production environments, your tests may reflect those differences. You can enable or disable specific tests depending on the environment.

Image building and configuration management

Tests for image building and configuration management tools follow a similar approach to testing the configuration for provisioning tools. Unit testing image building or configuration management metadata involves checking configuration. You do not need contract tests unless you modularize your configuration management, in which you follow the testing approach for modules. Integration tests should run in a testing environment, either to test that the server successfully starts with a new image or applies the correct configuration. End-to-end tests ensure that your new images and configurations do not impact the functionality of the system.

Exercise 6.2

You add some firewall rules to allow an application to access a new queue. Which combination of tests would be most valuable to your team for the change?

A. Unit and integration tests
B. Contract and end-to-end tests
C. Contract and integration tests
D. D. Unit and end-to-end tests

Find answers and explanations at the end of the chapter.

6.7.3 Identifying useful tests

The testing strategies for modules and configurations can help guide your initial approach to writing valuable tests. Figure 6.18 summarizes the types of tests you might consider for modules and configurations. Modules rely on unit, contract, and integration tests while configurations rely on unit, integration, and end-to-end tests.

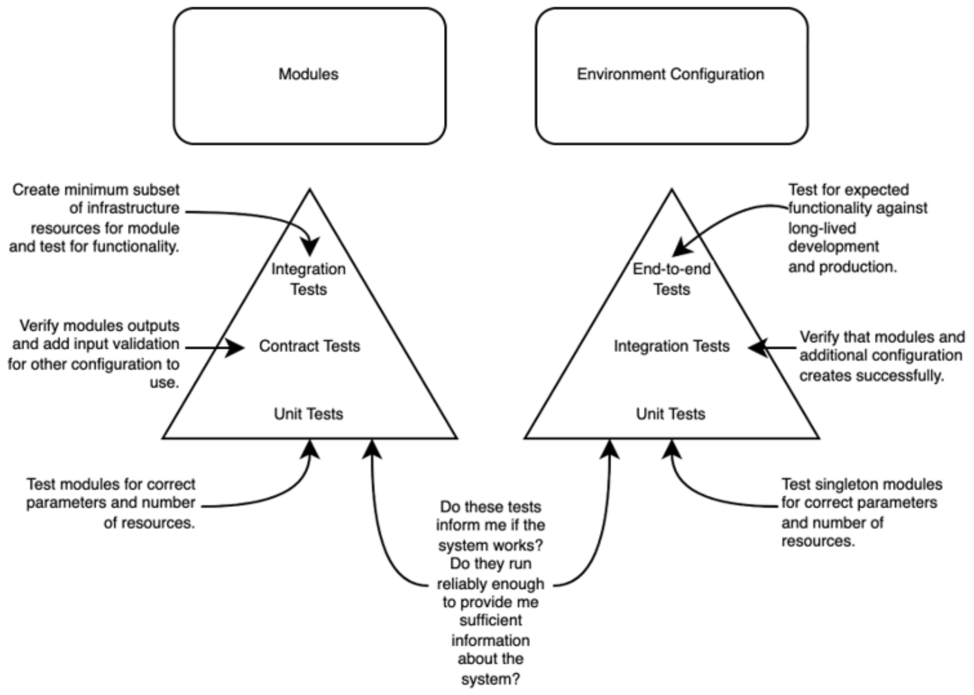

Modules
Environment Configuration
Create minimum subset of infrastructure resources for module and test for functionality.
Integration Tests
Verify modules outputs and add input validation for other configuration to use.
Contract Tests
Unit Tests
Test modules for correct parameters and number of resources.
Test for expected functionality against long-lived development and production.
End-to-end Tests
Integration Tests
Verify that modules and additional configuration creates successfully.
Unit Tests
Test singleton modules for correct parameters and number of resources.
Do these tests inform me if the system works? Do they run reliably enough to provide me sufficient information about the system?

Figure 6.18. Your testing approach will differ depending on whether or not you write a module or environment configuration.

How do you know *when* to write a test? Imagine your teammate might know that a database password needs to have alphanumeric characters with a 16-character limit. However, you might know this fact until you update a 24-character password, deploy the change, and wait five minutes for the change to fail.

I consider the practice of updating your tests a matter of turning unknown knowns into known knowns in your system. After all, you use observability to debug unknown unknowns and monitoring to track the known unknowns. In figure 6.17, you convert siloed knowledge (unknown knowns) that someone else knows into tests (known knowns) for team knowledge. New tests often reflect siloed knowledge that the team should know and acknowledge.

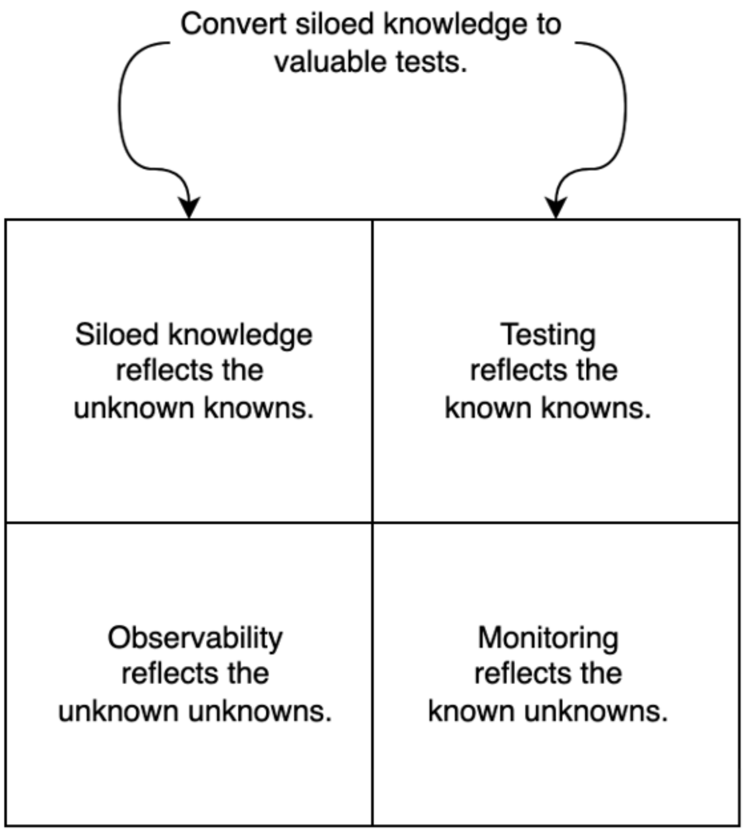

Convert siloed knowledge to valuable tests.
Siloed knowledge reflects the unknown knowns.
Testing reflects the known knowns.
Observability reflects the unknown unknowns.
Monitoring reflects the known unknowns.

Figure 6.19. Infrastructure testing converts siloed knowledge, something someone else might know, into a test to reflect the team's knowledge.

A good test shared knowledge to the rest of the team. You don't always need to build a new test. Instead, you might find an existing test that doesn't check for everything. Use a test to prevent your team from repeating problems.

Besides adding tests, you'll remove tests. You might write a test and discover that it fails half the time. It does not provide helpful information or increase your confidence in the system because of its unreliability. Removing the test cleans up your testing suite and helps eliminate false positives from flakiness.

Furthermore, you'll remove tests because you don't need them. For example, you might not need contract tests for every module or integration tests for every environment configuration. Always ask yourself if the tests provide value and if they run reliably enough for you to get sufficient information about the system!

The next chapter will show how to add tests to a delivery pipeline for your infrastructure as code. Even if you do not choose to automate the testing workflow, you have an opportunity to examine how changes could potentially affect your infrastructure.

6.8 Exercises and Solutions

Exercise 6.1

You notice that a new version of a load balancer module is breaking your DNS configuration. A teammate updated the module to output private IP addresses instead of public IP addresses. What can you do to help your team better remember the module needs public IP addresses?

A. Create a separate load balancer module for private IP addresses.
B. Add module contract tests to verify the module outputs both private and public IP addresses.
C. Update the module's documentation with a note that it needs public IP addresses.
D. Run integration tests on the module and check the IP address is publicly accessible.

Answer:

Rather than creating a new module, you can add a contract test to help the team remember that high-level resources need both private and public IP addresses (B). You could create a separate load balancer module (A). However, it may not help your team remember that specific modules must output specific variables. Updating the module's documentation (C) means that your team must remember to read the documentation first. Integration tests have a time and financial cost for running (D) when contract tests adequately solve the problem.

Exercise 6.2

You add some firewall rules to allow an application to access a new queue. Which combination of tests would be most valuable to your team for the change?

A. Unit and integration tests
B. Contract and end-to-end tests
C. Contract and integration tests
D. Unit and end-to-end tests

Answer:

The two most valuable tests would be unit and end-to-end tests (D). Unit tests will help ensure that someone doesn't remove the new rules going forward. End-to-end tests check that the application can access the queue successfully. Contract tests would not provide any help because you do not need to test inputs and outputs for firewall rules.

6.9 Summary

- The test pyramid outlines an approach to testing. The higher the test level in the pyramid, the more costly the test.
- Unit tests verify static parameters in modules or configurations.
- Contract tests verify that the inputs and outputs of a module matches expected values and formats.
- Integration tests create test resources, verify their configuration and creation, and delete them.
- End-to-end tests verify that the end user of an infrastructure system can run the expected functionality.
- Modules using factory, builder, or prototype patterns benefit from unit, contract, and integration tests.
- Configurations using composite or singleton patterns applied to environments benefit from unit, integration, and end-to-end tests.
- Other tests include monitoring for continuously testing system metrics, regression tests for out-of-band manual changes, or security tests for misconfigurations.

7 Continuous delivery and branching models

This chapter covers

- **Designing delivery pipelines to avoid pushing failures to production systems**
- **Choosing a branching model for infrastructure configuration for team collaboration**
- **Reviewing and managing changes to infrastructure resources within your team**

In the previous chapters, you learned how to write patterns for modules and dependencies. You also applied some general practices for writing infrastructure as code and sharing modules. The patterns, practices, and workflows had a lot of steps.

Furthermore, many of the workflows require careful coordination of changes. One day, you might try to make a change only to find out that your teammate's updates might overwrite yours! How do you make sure you manage conflicts during the development process?

One solution involves submitting change requests to a ticketing system. For example, if I wanted to change a server, I needed to fill out a change request in my ticketing system. This change request then gets reviewed by my peers (usually my team) and the change advisory board (on behalf of the company).

Most companies use this process, called change management, to figure out which changes conflict. Infrastructure ***change management*** involves submitting a change request detailing rollout and rollback steps for peer reviewers' approval.

DEFINITION **Change management for infrastructure is a process that facilitates a change to a system. It often involves detailing and reviewing the change across a company before approving it for production.**

Change management depends on peer review to prevent changes from overwriting each other. The peer review and change advisory board in the example serve as quality gates. ***Quality gates*** verify that the change request does not compromise a system's security or availability. Once my change passes the gates, I can schedule it and update the server accordingly.

> **DEFINITION** Quality gates for infrastructure as code is a step to enforce a system's security, availability, and resiliency through review or tests.

How does change management and quality gates help solve conflicts between changes? Imagine you applied the change management process to you and your teammate's conflicting changes. In figure 7.1, you and your teammate submit change requests to the ticketing system. Peers across the organization manually review each change and determine that your teammate's change has a minimal impact on users. They reschedule your change for another day to avoid conflict with your teammate's change.

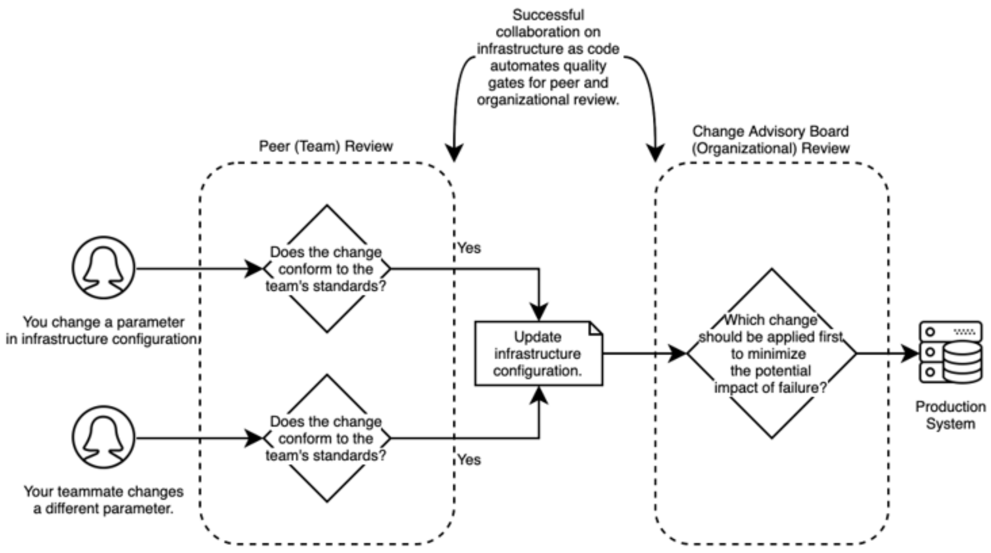

Successful collaboration on infrastructure as code automates quality gates for peer and organizational review.
Peer (Team) Review
Change Advisory Board (Organizational) Review
You change a parameter in infrastructure configuration.
Your teammate changes a different parameter.
Does the change conform to the team's standards?
Does the change conform to the team's standards?
Yes
Yes
Update infrastructure configuration.
Which change should be applied first to minimize the potential impact of failure?
Production System

Figure 7.1. Collaboration on infrastructure as code involves streamlining the process of peer and organizational review of changes.

Change management can take weeks to complete. Manual review doesn't catch every problem or prevent every infrastructure change conflict! You still need to understand how to revert changes. You'll learn about fixing change failures in Chapter 11.

Rather than rely on change management, you can use infrastructure as code to *communicate* changes through code and *automate* change management. This chapter focuses on streamlining change management by scaling and automating infrastructure as code development processes across your team and company. You'll face the challenge of people working on the same or dependent resources while trying not to break a production system.

Image building and configuration management .

Throughout this book, I focus on the use case of provisioning infrastructure. Use cases for image building and configuration management should follow the general pattern for delivery pipelines in this chapter. The patterns and practices to assess infrastructure changes and incorporate automated testing remain consistent for all use cases.

7.1 Delivering Changes to Production

How do you control infrastructure as code changes to a production environment? You apply software development practices like continuous integration, delivery, or deployment to organize code changes from different collaborators and prepare to release infrastructure as code to a production environment.

Continuous integration or delivery requires automated testing to automate the release and management of changes. I'll explain how you can automate infrastructure changes and make the most of its benefits. It uses the testing practices you learned from the last chapter with the delivery pipeline patterns in this chapter.

7.1.1 Continuous integration

Recall the infrastructure as code conflict you ran into with your teammate. You did not know your teammate's change would affect yours, and vice versa. How do you automatically identify the conflicts before your peers review your change?

One solution in Figure 7.2 involves asking your team to regularly merge their changes into the main infrastructure as code. If your team continuously *integrates* their changes into the main configuration, you and your teammate can identify the conflicts earlier before they overwrite your updates.

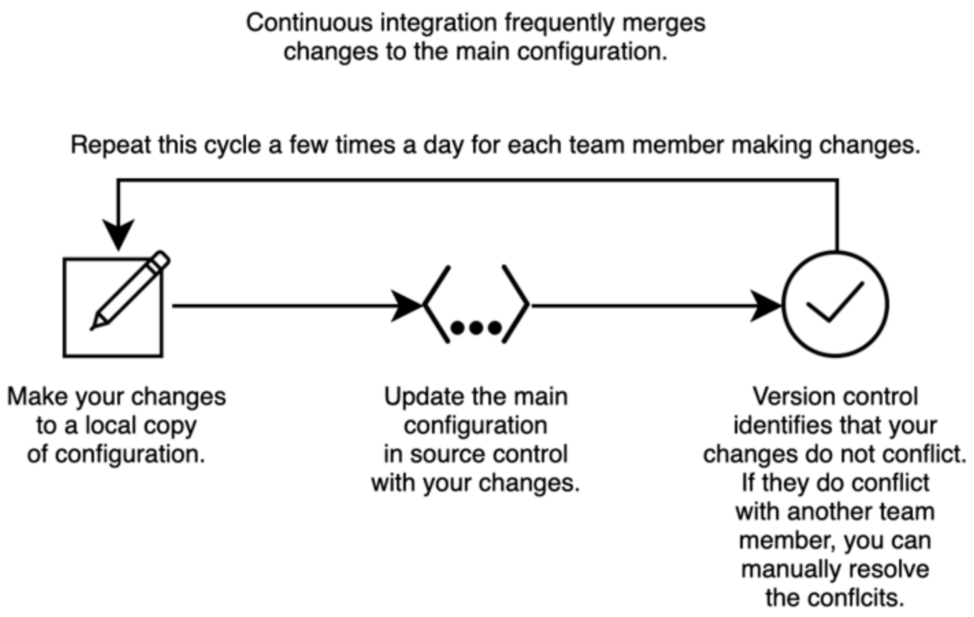

Continuous integration frequently merges changes to the main configuration.
Repeat this cycle a few times a day for each team member making changes.
Make your changes to a local copy of configuration.
Update the main configuration in source control with your changes.
Version control identifies that your changes do not conflict. If they do conflict with another team member, you can manually resolve the conflcits.

Figure 7.2. Continuous integration involves merging changes to the main configuration often, allowing earlier identification of conflicts in changes.

You can apply the practice of ***continuous integration*** to merge your changes into the main configuration multiple times a day and check if they conflict with collaborators.

> **DEFINITION** Continuous integration for infrastructure as code is a practice of regularly and frequently merging changes into your repository after verifying the change in a testing environment.

How often should you merge? Knowing when to merge requires a bit of experience and depends on the type of change you want to make. As a general rule, I merge when I accumulate a few lines of configuration change that (likely) will not break the system. Sometimes, this means I merge several times a day. Other times, for a difficult change, I might merge once or twice a day. The rest of the team also continues to work on and merge their changes a few times a day!

Each time a team member merges their changes, a build tool (such as a continuous integration framework) should start a workflow to test the changes and deploy them. Figure 7.3 shows an example workflow that a build tool might run. The workflow checks infrastructure as code for merge conflicts, runs unit tests to verify formatting, and pauses for peer review. Once it passes peer review, the build tool deploys the changes to production.

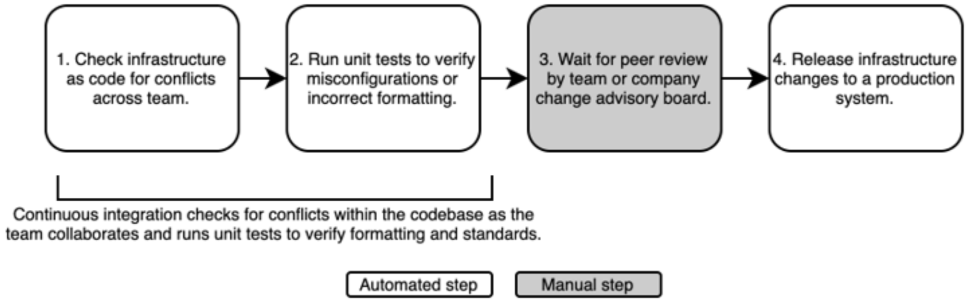

1. Check infrastructure as code for conflicts across team.
2. Run unit tests to verify misconfigurations or incorrect formatting.
3. Wait for peer review by team or company change advisory board.
4. Release infrastructure changes to a production system.
Continuous integration checks for conflicts within the codebase as the team collaborates and runs unit tests to verify formatting and standards.
Automated step
Manual step

Figure 7.3. Continuous integration in a delivery pipeline includes automated unit tests before waiting for manual approval to production.

You can express this workflow as part of a ***delivery pipeline***. A pipeline organizes and automates a set of stages to build, test, deploy, and release your infrastructure as code.

> **DEFINITION** A delivery pipeline for infrastructure as code expresses and automates a workflow to build, test, deploy, and release a change to infrastructure.

An infrastructure delivery pipeline starts by checking configuration conflicts or syntax problems. Unit testing in a continuous integration pipeline offers you some confidence that you don't have change conflicts. You can then submit the change to your team or company for review. The pipeline automatically releases (or applies) it to production.

Why should you design a delivery pipeline and add it to your build tool? You might not remember all the steps you need to release a change to production. A delivery pipeline codifies the process so you don't have to remember it. Agreeing on a delivery pipeline for infrastructure helps you scale infrastructure changes consistently and reproducibly, no matter the infrastructure resource.

7.1.2 Continuous delivery

You used continuous integration to merge changes and check for conflicts, but how do you know if the system functions as you expect? Continuous integration validates formatting and standards, but you don't know if the configuration works until release. You have some confidence in your unit test, but you need more tests to feel comfortable with the updates.

Figure 7.4 re-imagines the continuous integration workflow you started for infrastructure as code. You update your delivery pipeline with extra stages after unit tests. Before you submit the change for peer review, you deploy the configuration to a testing environment and test it with integration and end-to-end tests. After peer review, you *deliver* the changes to production and re-run the end-to-end tests to verify production functionality.

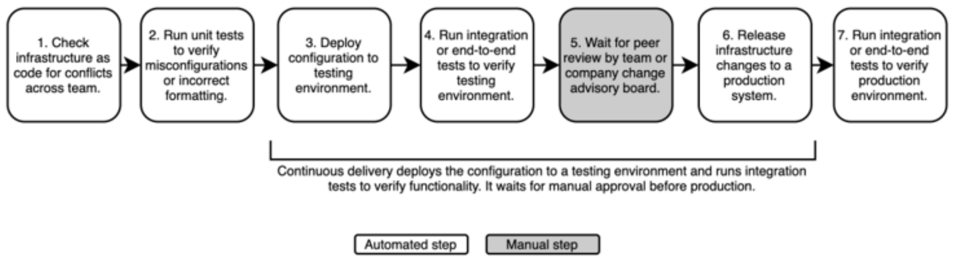

1. Check infrastructure as code for conflicts across team.
2. Run unit tests to verify misconfigurations or incorrect formatting.
3. Deploy configuration to testing environment.
4. Run integration or end-to-end tests to verify testing environment.
5. Wait for peer review by team or company change advisory board.
6. Release infrastructure changes to a production system.
7. Run integration or end-to-end tests to verify production environment.
Continuous delivery deploys the configuration to a testing environment and runs integration tests to verify functionality. It waits for manual approval before production.
Automated step
Manual step

Figure 7.4. Continuous delivery automatically deploys and runs infrastructure changes in a testing environment and waits for manual approval for production.

You expanded your delivery pipeline with the practice of ***continuous delivery***. Continuous delivery adds a step to your delivery pipeline that deploys your infrastructure configuration to a testing environment for integration or end-to-end testing after passing unit tests.

> **DEFINITION** Continuous delivery for infrastructure as code deploys your infrastructure change to a testing environment for integration or end-to-end testing after merging your changes to a repository. It can involve a manual quality gate before releasing the changes to production.

Whenever someone pushes a change to source control, it starts the pipeline's workflow to verify the changes in a testing environment. Once the integration and end-to-end tests pass, the pipeline can wait for manual approval before deploying the changes to production.

Why use continuous delivery over continuous integration? Continuous delivery incorporates all of the automated tests you worked hard to write in chapter 6. Furthermore, its delivery pipeline includes *tests as quality gates*. A teammate reviewing your change might feel more confident about if the tests verify the change implementation.

For more on continuous delivery

Continuous delivery requires an entire book! I applied it directly to infrastructure and tried to cover it in one section. If you want to review a more practical example, I created an example pipeline at https://github.com/joatmon08/manning-book/blob/main/.github/workflows/pipeline.yml. The pipeline uses GitHub Actions to deploy the "hello-world" service to Google Cloud Run. Its stages include unit tests, testing environment deployment, and integration tests.

Continuous delivery should involve small and frequent changes to code. You push these changes automatically to a testing environment and wait for manual approval before pushing them to production. However, changes waiting for manual approval accumulate like a traffic jam. A few cars that slow down can cascade to many cars, ultimately affecting your expected arrival time!

A manual approval step builds a batch of changes, which introduces some problems. When you push a large batch of changes into production in figure 7.5, you wait for the system to process and deploy the changes. Unfortunately, you also introduce an unintended failure because some changes conflict. Your team spends days tracking down which combination of changes caused the failure.

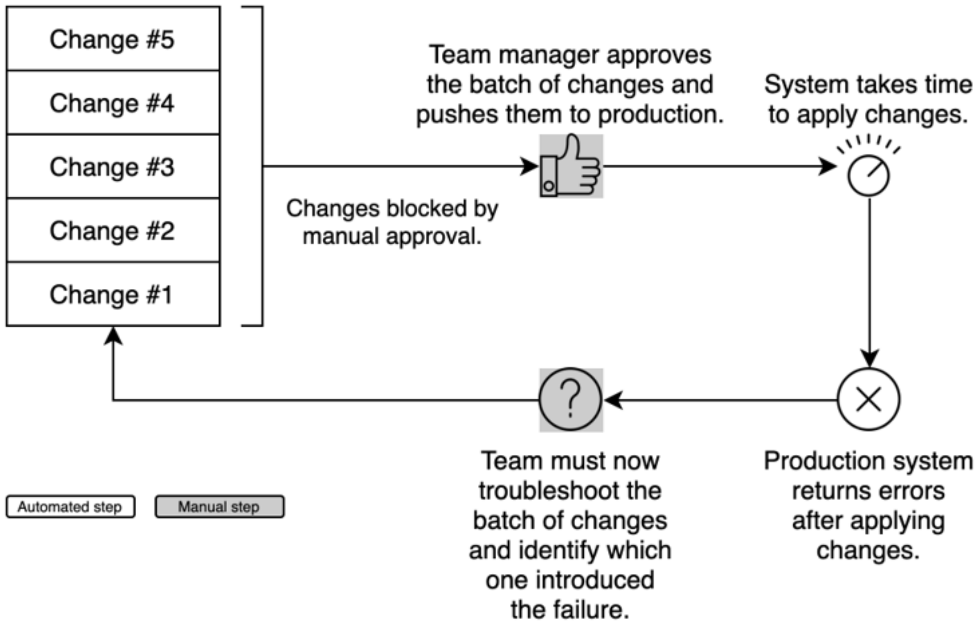

Change #5
Change #4
Change #3
Change #2
Change #1
Team manager approves the batch of changes and pushes them to production.
System takes time to apply changes.
Changes blocked by manual approval.
Team must now troubleshoot the batch of changes and identify which one introduced the failure.
Production system returns errors after applying changes.
Automated step
Manual step

Figure 7.5. Avoid pushing large batches of changes to production whenever you introduce a manual approval to prevent complex troubleshooting.

When you use continuous delivery, approve changes as quickly as possible. Implement a shorter feedback cycle for manual approval. You can also limit how many changes you approve at once. Both solutions mitigate some of the risk introduced by manual approval. I'll cover another solution in the next section that entirely omits the manual approval process.

7.1.3 Continuous deployment

Can you prevent large batches of changes by eliminating manual stages in your delivery pipeline? You can! However, you must practice continuous integration and delivery before removing manual approval.

Removing manual approval from your pipeline means you must have confidence in your *testing*. The pipeline in figure 7.6 adds more integration and end-to-end tests to verify the system and automatically pushes the changes to production. You feel confident that your tests sufficiently check system functionality *and* you can easily revert changes. You remove the manual approval and promote changes immediately to a production environment.

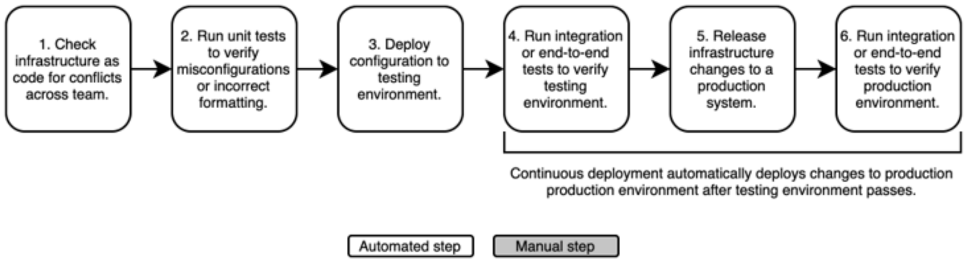

1. Check infrastructure as code for conflicts across team.
2. Run unit tests to verify misconfigurations or incorrect formatting.
3. Deploy configuration to testing environment.
4. Run integration or end-to-end tests to verify testing environment.
5. Release infrastructure changes to a production system.
6. Run integration or end-to-end tests to verify production environment.
Continuous deployment automatically deploys changes to production production environment after testing environment passes.
Automated step
Manual step

Figure 7.6. Continuous deployment fully automates the testing and applying of changes to production.

Continuous deployment removes the manual approval step and promotes changes directly from a testing environment into production.

> **DEFINITION** Continuous deployment for infrastructure as code deploys and tests your infrastructure change to a testing environment and automatically promotes the change to production when the tests pass.

Automatic deployment prevents a traffic jam of changes. Pushing infrastructure changes often takes hours and affects unknown dependencies. You can use continuous deployment for infrastructure if you have a thorough testing strategy and familiarity in fixing failures.

Using the techniques in Chapter 11 to fix failures can help you practice continuous deployment. However, most organizations do not fully embrace continuous deployment for their infrastructure because they do not feel confident in testing or reverting changes. Investing time and practice in these patterns can help you move closer to a continuous deployment model.

7.1.4 Choosing a delivery approach

Continuous delivery and deployment create a workflow to test and deliver infrastructure as code to a production environment. However, you cannot expect your organization to feel comfortable automating all changes directly to production! I recommend applying a combination of continuous delivery and deployment to infrastructure change management. First, you must categorize the type of changes you implement before choosing a delivery approach.

TYPES OF INFRASTRUCTURE CHANGE

The type of change affects how you deliver it to production. You need to work with your organization's change review board to classify the type of change and automate the testing and review for each one! Otherwise, you may find yourself non-compliant with an audit requirement.

Imagine you make a routine change every week to a server. You update the server's infrastructure as code with a new tag. The automation never changes and does not fail often. When it does fail, you know exactly how to fix it. The server's routine change becomes a good candidate for continuous deployment.

In figure 7.7, you continuously deploy the server change directly to production without manual approval. The pipeline replaces a manual approval step after the testing environment with a test to check for a prefix in the commit message. Your server change has the commit message of a "standard" change, so the pipeline bypasses manual approval.

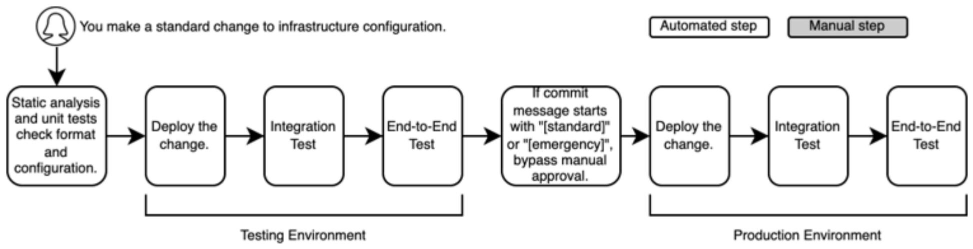

You make a standard change to infrastructure configuration.
Automated step
Manual step
Static analysis and unit tests check format and configuration.
Deploy the change.
Integration Test
End-to-End Test
If commit message starts with "[standard]" or "[emergency]", bypass manual approval.
Deploy the change.
Integration Test
End-to-End Test
Testing Environment
Production Environment

Figure 7.7. Standard or emergency change can have an initial peer review before your delivery pipeline automatically pushes it to production.

You routinely make ***standard changes*** to your infrastructure. Examples of standard changes include upgrading container images in an orchestrator, deploying a new queue, or adding a new alert into your monitoring system. If the change fails, you can reference a runbook to roll back the change without impacting anything.

> **DEFINITION** A standard change to infrastructure is a commonly implemented change with well-defined behavior and rollback plan.

Why should you consider continuous deployment for standard changes? A standard change often involves a commonly automated, well-defined fix. You don't want your team pausing to review and approve a repetitive change. The standard change distracts them from more important changes.

Other types of changes benefit from continuous deployment. Imagine you discover that an application on a server stopped running. You need the application up and running again quickly.

Rather than run a standard change, you implement the fix in infrastructure as code and update its commit message with "emergency". After pushing the change, your build system bypasses the manual approval stage because the commit message identifies the change as an emergency.

Besides continuously delivering standard changes, you can opt to continuously deliver ***emergency changes*** for emergency scenarios to fix production. When possible, push fixes using infrastructure as code and the commit message to identify the emergency change.

> **DEFINITION** **An emergency change to infrastructure is a change that you must quickly implement in production to fix system functionality.**

Emergency changes go directly to production without manual approval because you often need to fix the system quickly. A manual approval may hinder the resolution of a problem. As a result, adding a bypass for emergency changes helps you quickly resolve the problem and record your resolution history.

To continuously deploy standard and emergency changes, you *must* have automated testing in your pipeline before adding the ability to bypass the manual approval step. Furthermore, you need to standardize the bypass commit message structure. A bypass allows engineers to deploy fixes without a backlog of changes. It also allows compliance and security teams to audit the sequence of changes.

Can't I just run an emergency change manually?

I highly recommend you use infrastructure as code and your delivery pipeline to make emergency changes. The commits record a history of your resolution steps, and your pipeline tests your changes before you run automation that makes the system worse.

However, you might find the deployment pipeline can take too long to run when you're trying to push a fix out quickly. You may consider making a manual change, recognizing that you may not benefit from the automated tests and checks addressed by the pipeline.

After you make a manual change, reconcile the actual infrastructure state with the expected infrastructure as code. The practice of reconciliation involves manually updating the configuration to match the infrastructure resource (review this technique in Chapter 2).

Other changes should not use continuous deployment. Imagine you've been assigned a new project. You need to enable IPv6 on all networks. By making this networking change, you could affect every application and system in the network!

For this new and major change, you *do not* want to skip manual approval. You want some skilled network engineers reviewing your infrastructure as code. In figure 7.8, you update your networks' infrastructure as code with IPv6 and wait for manual approval before production. The manual approval step communicates to the other application and engineering teams that your changes may have a large blast radius if they fail.

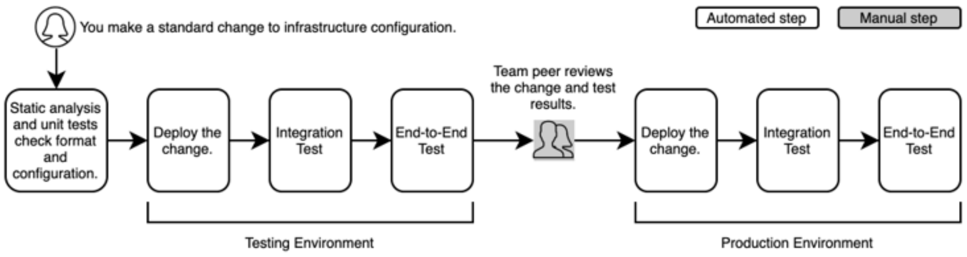

You make a standard change to infrastructure configuration.
Automated step
Manual step
Static analysis and unit tests check format and configuration.
Deploy the change.
Integration Test
End-to-End Test
Team peer reviews the change and test results.
Deploy the change.
Integration Test
End-to-End Test
Testing Environment
Production Environment

Figure 7.8. New or major changes should have manual peer approval before applying to production.

New or major changes can affect a system's architecture, security, or availability. These changes should require filing an issue or ticket with some justification and discussion. They also involve a manual change review from your peers in the team or company.

> **DEFINITION** A new or major change to infrastructure is a change that potentially affects a system's architecture, security, or availability. They do not have well-defined implementation or rollback plans.

Major changes have the potential for significant impact or a high risk of failure. Similarly, new or unknown changes can cause unpredictable results and complex rollback steps. Asking for manual approval signals that you may need some help if the changes fail. Other examples of new or major changes often include updates to network CIDR blocks (if they affect other allocations), DNS, certificate changes, upgrades to workload orchestrators, or platform refactoring (such as adding a secrets manager to applications).

DELIVERY APPROACH BASED ON CHANGE TYPE

After categorizing the changes you make and their types, you can decide on your delivery approach. For the most part, standard and emergency changes use continuous deployment while new and major changes use continuous delivery.

Table 7.1 outlines some of the types of changes, their delivery approach, and an example. However, you will have some exceptions to the general practices. Some standard changes may require continuous delivery because they impact other resources. By contrast, changes related to greenfield (new) environments could implement a continuous deployment approach because it does not impact other systems.

Table 7.1. Types of changes and delivery approach

Type of Change	Delivery Approach	Manual Approval Before Production?	Example
Standard	Continuous Deployment	No	Adding servers to a scaling group
Emergency	Continuous Deployment or manual change	No	Rolling back an operating system image to a previous version
Major	Continuous Delivery	Yes	Enabling SSL for all services and infrastructure
New	Continuous Delivery	Yes	Deploy a new type of infrastructure component

Continuous integration, delivery, and deployment also apply to the software development lifecycle. However, applying the concepts to the infrastructure lifecycle pushes the limits of your organization's change and review processes. Regularly categorizing your changes and evaluating your change and review process can help balance productivity and governance, something we mentioned as part of module sharing practices.

Configuration management

Configuration management tools should follow a similar approach for assessing change types and applying continuous delivery or deployment.

As a general rule, make sure to quickly review and approve changes and *push them into production as soon as possible*. A larger batch of changes has a larger blast radius. If you push every change in one batch and affect a business-critical application, you must troubleshoot each change in the batch. The complexity of troubleshooting grows when you need to identify which change affected the system.

Exercise 7.1

Choose a standard infrastructure change in your organization. What do you need in order to confidently continuously deliver the change to production? What about continuous deployment? Outline or diagram stages in your delivery pipelines for both.

Find answers and explanations at the end of the chapter.

7.1.5 Modules

What about delivery pipelines for modules? You learned about sharing, releasing, and managing versions of infrastructure modules in Chapter 5 and testing their functionality in Chapter 6. I alluded to the idea of automating the process of testing and releasing modules but did not fully explain their delivery pipelines.

Delivery pipelines for infrastructure modules differ slightly from the examples I outlined for production configuration. You alter the delivery pipeline in figure 7.9 to release a module after testing instead of delivering to production. You keep a manual approval step for your team to review the module.

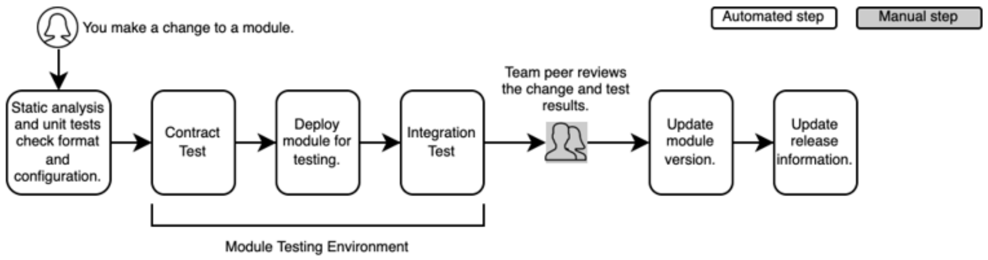

You make a change to a module.
Automated step
Manual step
Static analysis and unit tests check format and configuration.
Contract Test
Deploy module for testing.
Integration Test
Team peer reviews the change and test results.
Update module version.
Update release information.
Module Testing Environment

Figure 7.9. After testing the module, wait for the team to review the change and test results before updating and releasing the new module version.

You can categorize module changes with the same types of changes as configurations, including standard, emergency, major, and new changes. Table 7.2 outlines the types of changes, their delivery approach, and an example. For the most part, they match the approach recommended for configurations.

Table 7.2. Types of module changes and delivery approach

Type of Change	Delivery Approach	Manual Approval Before Production?	Example
Standard	Continuous Deployment	No	Enable an override for an existing default parameter.
Emergency	Continuous Deployment or branch	No	Rolling back an operating system image to a previous version
Major	Continuous Delivery	Yes	Update database or infrastructure using data.
New	Continuous Delivery	Yes	Deploy a new server module.

Some module changes benefit from review or pair programming with a subject matter expert, such as database configurations or other specialized infrastructure involving data. However, emergency changes on modules may have a different delivery approach.

Emergency changes on modules means isolating a quick fix to a different version of the module. You can achieve isolation in one of two ways. You could implement the fix and continuously deploy and release a new version of the module with the changes. Alternatively, you could create a ***branch*** of a module repository and update your infrastructure configuration to reference the branch.

> **DEFINITION** A branch in version control is a pointer to a snapshot of code. It allows you to separately implement changes based on that snapshot.

After validating the branch, you can update the main branch of the module with a standard change. Branching the module helps you quickly implement the emergency module change and reconcile the module changes later.

If I know other teams pin their versions, I prefer to continuously deploy a new version of the module with the fix. While a branch can isolate the emergency change, I have to remember to merge it back into the main release of the module. In the next section, you'll learn about branching models and how to apply them to infrastructure as code changes.

Image building and configuration management

Delivery pipelines for image building and configuration management modules follow a similar approach to modules for provisioning tools. Make sure to version and test changes to images before deploying them to production.

7.2 Branching models

Besides implementing continuous delivery or deployment, you need to standardize how changes merge to your main configuration. The main branch in version control serves as a source of truth for configuration. Updating the configuration requires some additional coordination and collaboration within your team.

Imagine you want to reduce access for a firewall while your teammate refreshes its license. Your team has a continuous delivery pipeline to test and manually approve changes to the production firewall. However, you and your teammate's changes present two problems.

First, how can both of you work on and test your changes in isolation? Second, how do you control which change should go first? Figure 7.10 outlines you and your teammate's dilemma of who should deploy their changes first. You want to avoid pushing both changes at the same time. If the firewall causes network access failure, you won't know which change caused the problem.

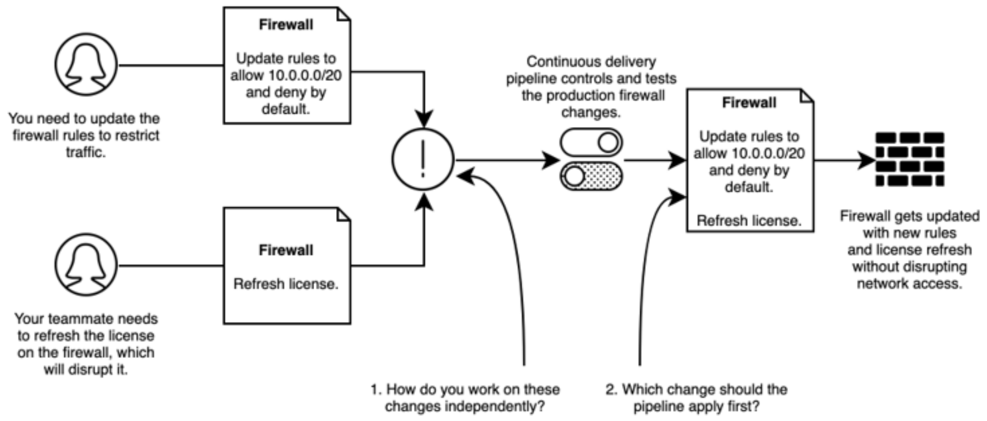

Firewall
Update rules to allow 10.0.0.0/20 and deny by default.
You need to update the firewall rules to restrict traffic.
Firewall
Refresh license.
Your teammate needs to refresh the license on the firewall, which will disrupt it.
Continuous delivery pipeline controls and tests the production firewall changes.
Firewall
Update rules to allow 10.0.0.0/20 and deny by default.
Refresh license.
Firewall gets updated with new rules and license refresh without disrupting network access.
1. How do you work on these changes independently?
2. Which change should the pipeline apply first?

Figure 7.10. Even if you have a continuous delivery pipeline to deliver changes to the firewall, you need additional development coordination to identify which change must apply first.

A ***branching mode***l coordinates how your team uses version control to enable parallel work while minimizing disruption and troubleshooting complexity. You can choose from two types of branching models, feature-based or trunk-based development.

DEFINITION A branching model defines how your team uses version control to enable parallel work and resolve conflicts in their efforts.

Each branching model comes with its complexities in implementation, especially in infrastructure as code. I'll describe how you can apply both development models to coordinate the firewall rule and license changes between you and your teammate. Then, I'll discuss some limitations to each approach and how your teams can choose.

7.2.1 Feature-based development

What if you and your teammate could work on your changes in isolation before combining them? If your teammate creates a *branch* with their license change and you create a branch with your firewall changes, you isolate your changes from each other. When you both finish, you merge your changes to the main branch and resolve conflicts with each other.

Figure 7.11 demonstrates how you and your teammate choreograph your changes on different branches. You name your branch "TICKET-002" for the firewall rule, and your teammate names their branch "TICKET-005" for the license update. Your firewall rule changes get approved first, so you put them into the main branch and deploy them to production. Your teammate continues working on the license update. They retrieve your firewall rule updates into their branch for further testing before merging their changes back to main.

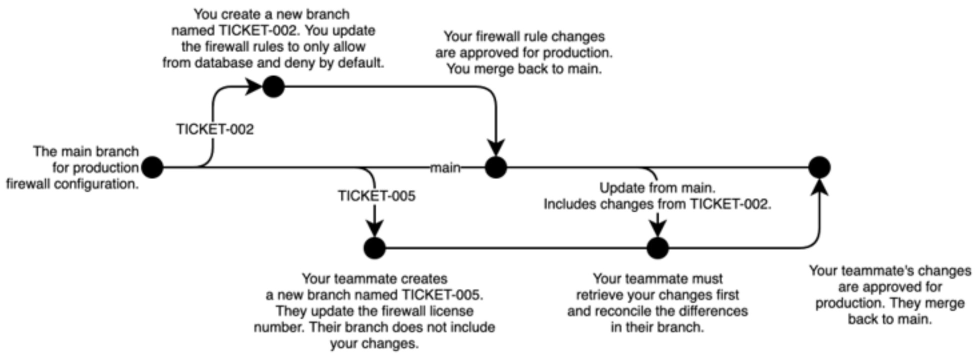

You create a new branch named TICKET-002. You update the firewall rules to only allow from database and deny by default.
Your firewall rule changes are approved for production. You merge back to main.
TICKET-002
The main branch for production firewall configuration.
main
TICKET-005
Update from main. Includes changes from TICKET-002.
Your teammate creates a new branch named TICKET-005. They update the firewall license number. Their branch does not include your changes.
Your teammate must retrieve your changes first and reconcile the differences in their branch.
Your teammate's changes are approved for production. They merge back to main.

Figure 7.11. When you use feature-based development, you isolate your changes to your branch and reconcile conflicts with the main configuration.

Feature-based development allows you to evolve your changes independent of your teammates by isolating them to a branch.

DEFINITION **Feature-based development (also known as feature branching or Git Flow) is a branching pattern that separates different changes into individual branches. You merge the changes on a specific branch to the main configuration after testing.**

The flow of feature-based development helps you focus on the composability of your change independent of others. However, you need to constantly fetch changes from the main branch and reconcile them with your branch. Feature-based development works best when each team member diligently updates and tests their branches.

Let’s examine feature-based development in action. Imagine you start your feature-based development workflow for the firewall by cloning a local copy of the configuration from version control.

```
$ git clone git@github.com:myorganization/firewall.git
```

You create a branch, which creates a pointer for your updates. I recommend naming the branch after the ticket number (such as “TICKET-002”) associated with the change, although you can also use a descriptive dash-delimited name.

```
$ git checkout -b TICKET-002
```

You make your changes to the firewall rules on the branch. Then, you use your command line to commit your changes to your local branch.

```
$ git commit -m “TICKET-002 Only allow traffic from database”
[TICKET-002 cdc9056] TICKET-002 Only allow traffic from database
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 firewall.py
```

You have your changes locally, but you want others to review your change. You push the changes to a remote branch.

```
$ git push --set-upstream origin TICKET-002
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 1.06 KiB | 1.06 MiB/s, done.
Total 5 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:myorganization/firewall.git
 * [new branch]      TICKET-002 -> TICKET-002
Branch 'TICKET-002' set up to track remote branch 'TICKET-002' from 'origin'.
```

Meanwhile, your teammate works on “TICKET-005”, which updates the license. They create a new branch called “TICKET-005” with changes that *do not* include your firewall rule updates. Notice that your branch does not include their updated license, and their branch does not include your updated firewall rules. You can review the difference between the two branches.

```
$ git diff TICKET-002..TICKET-005
diff --git a/firewall.py b/firewall.py
index 74daecd..aaf6cf4 100644
--- firewall.py
+++ firewall.py
@@ -1,3 +1,3 @@
-print("License number is 1234")
+print("License number is 5678")

-print("Firewall rules should allow from database and deny by default.")
\ No newline at end of file
+print("Firewall rules allow all.")
\ No newline at end of file
```

You open a ***pull request***, notifying your team that you finished your changes.

> **DEFINITION** A pull request notifies the maintainers of a repository that you have external changes you would like to merge into the main configuration.

You add members of the change advisory board to review your pull request. They approve the changes, and you merge your changes back to the main branch.

Your teammate has not received approval to update the license yet. To ensure they don't affect the production configuration, they need to retrieve all changes from the main branch, including yours in "TICKET-002".

```
$ git checkout main
Switched to branch 'main'
Your branch is behind 'origin/main' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)

$ git pull --rebase
Updating 22280e7..084855a
Fast-forward
 firewall.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Then, they return to their branch named "TICKET-005" and merge changes from the main branch into the "TICKET-005" branch.

```
$ git checkout TICKET-005
Switched to branch 'TICKET-005'
Your branch is up to date with 'origin/TICKET-005'.

$ git merge main         [9:50:55]
Auto-merging firewall.py
Merge made by the 'recursive' strategy.
 firewall.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

When your teammate reviews the firewall configuration, they'll find your changes from "TICKET-002". They can update their branch with the changes from main.

```
$ git push --set-upstream origin TICKET-005
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 1.06 KiB | 1.06 MiB/s, done.
Total 5 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:myorganization/firewall.git
 * [new branch]      TICKET-005 -> TICKET-005
Branch 'TICKET-005' set up to track remote branch 'TICKET-005' from 'origin'.
```

Once they receive approval, they can merge the new firewall license to the main branch.

Feature-based development requires a number of steps for each team member. You can simplify the workflow by automating the testing and merging process. Use a delivery pipeline to organize the changes across branches.

Figure 7.12 organizes the feature-based development workflow of the delivery pipeline for your and your teammate. You and your teammate each get a branch pipeline with its own testing environment. For example, you have a branch named "TICKET-002" and a new firewall environment that isolates your changes. You run unit tests, deploy the change, and run integration and end-to-end tests in the "TICKET-002" firewall environment.

Once your branch tests pass, you merge the changes to the main branch. While you worked on your change, your teammate separately created their own branch and firewall environment named "TICKET-005". Your teammate realizes that you recently made updates to the firewall configuration. In figure 7.12, they retrieve changes from main and make sure it still works with their branch and environment. Once your teammate runs the same unit, integration, and end-to-end tests on their branch, they merge "TICKET-005" changes to the main branch for production deployment.

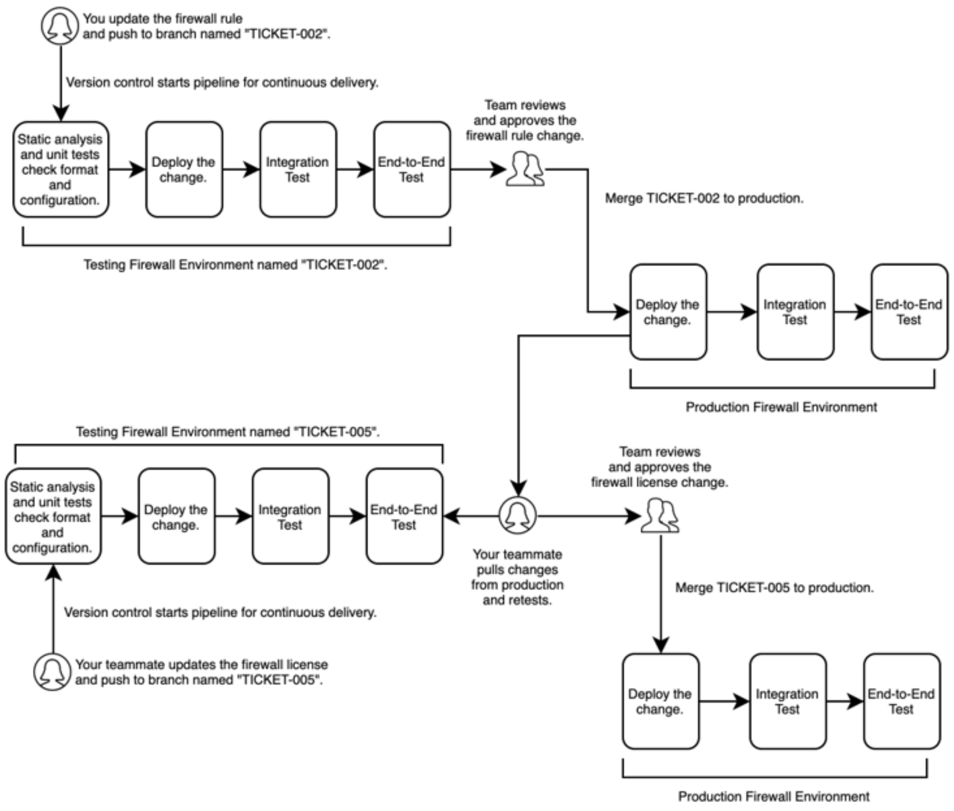

You update the firewall rule and push to branch named "TICKET-002".
Version control starts pipeline for continuous delivery.
Static analysis and unit tests check format and configuration.
Deploy the change.
Integration Test
End-to-End Test
Testing Firewall Environment named "TICKET-002".
Team reviews and approves the firewall rule change.
Merge TICKET-002 to production.
Deploy the change.
Integration Test
End-to-End Test
Production Firewall Environment
Testing Firewall Environment named "TICKET-005".
Static analysis and unit tests check format and configuration.
Deploy the change.
Integration Test
End-to-End Test
Your teammate pulls changes from production and retests.
Team reviews and approves the firewall license change.
Merge TICKET-005 to production.
Version control starts pipeline for continuous delivery.
Your teammate updates the firewall license and push to branch named "TICKET-005".
Deploy the change.
Integration Test
End-to-End Test
Production Firewall Environment

Figure 7.12. You can use feature-based development to isolate testing of your changes on the branch.

Why create a testing environment for each branch? A testing environment per branch isolates your changes and tests them relative to the main branch. As a temporary environment, the branch's testing environment can minimize the need for a persistent testing environment and reduce the overall cost of infrastructure. However, a testing environment may take some time to create.

Your team gets a few benefits from feature-based development, including the following:

- Ability to isolate changes to a branch
- Ability to test changes within the branch.
- Implicit step for peer review. You can only merge your changes to production if someone approves the change.
- Separation of emergency changes on a branch. After validating the emergency change, you can merge it to the main branch.

Fortunately, repository hosting services (e.g., GitHub or GitLab) have functionality that can help you automate a feature-based development model. Such functionality includes labels for tracking specific features, status checks with integration tests before merging your branch, and automated deletion of old branches. You can also define a list of reviewers and automatically add them to pull requests.

7.2.2 Trunk-based development

Imagine your organization does not want to create a testing environment per branch, and many of the engineers do not feel comfortable with a feature-based development workflow. Instead of creating a branch, you and your teammate work together on the main branch.

Figure 7.13 shows how you and your teammate can collaborate on the main branch. You update the firewall rule and push the changes first. Then, your teammate updates their local repository to include your changes and pushes their changes to the main branch.

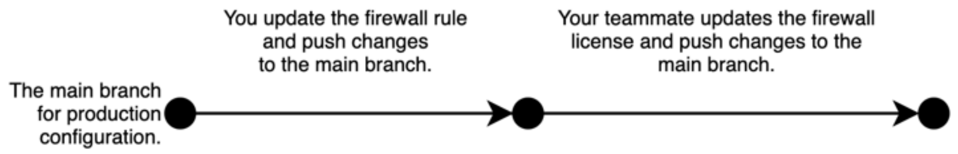

You update the firewall rule and push changes to the main branch.
Your teammate updates the firewall license and push changes to the main branch.
The main branch for production configuration.

Figure 7.13. When you use trunk development, you maintain one main branch and update the production configuration directly.

The workflow seems more streamlined since you both push to one branch. ***Trunk-based development*** means that you push changes directly to the main branch without isolating your changes in version control.

> **DEFINITION** Trunk-based development (also known as pushing to main) is a branching pattern that pushes all changes directory to the main branch. It favors making small changes and using testing environments to verify changes succeed.

Trunk-based development does not allow you to evolve your changes independently of your teammates. However, this limitation becomes a benefit. Trunk-based development forces you to implement changes in a *specific order*. You can quickly identify which commit causes the change and resolve it. The pattern offers an opinionated way to orchestrate and apply infrastructure as code changes.

Let’s apply trunk-based development to your firewall rule and your teammate’s firewall license updates. You start by cloning a local copy of the firewall configuration from version control. When you clone the configuration, you can check the main branch.

```
$ git clone git@github.com:myorganization/firewall.git
       $ git branch --show-current
       main
```

You make the firewall rule changes to the main branch. Commit your changes.

```
$ git commit -m “TICKET-002 Only allow traffic from database”
[TICKET-002 cdc9056] TICKET-002 Only allow traffic from database
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 firewall.py
```

Update your local copy to make sure you retrieve new changes from main. Use `git pull --rebase` to fetch the changes from the remote repository, merge them into your local copy, and rebase the local history with the remote one.

```
$ git pull --rebase
Already up to date.
```

Now, you can push the changes to the main branch. Your push should start the delivery pipeline in figure 7.14. Your pipeline runs unit and integration tests against a testing environment. After all testing stages pass, the pipeline waits for manual approval from your team. Your teammate can review your changes and approve them. Once they approve your change, the pipeline deploys your firewall rule change to production.

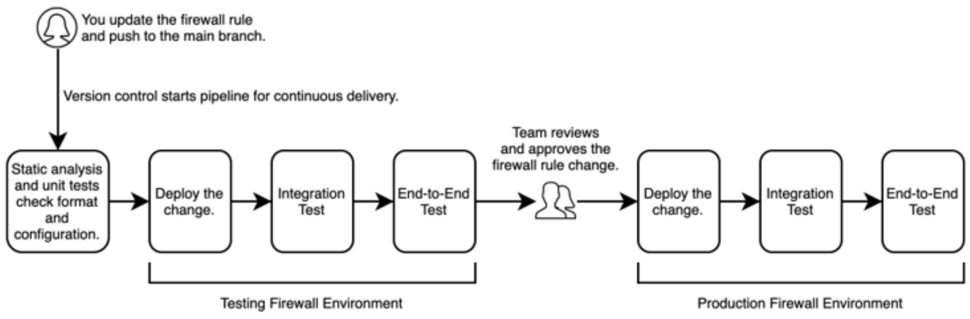

You update the firewall rule and push to the main branch.
Version control starts pipeline for continuous delivery.
Static analysis and unit tests check format and configuration.
Deploy the change.
Integration Test
End-to-End Test
Team reviews and approves the firewall rule change.
Deploy the change.
Integration Test
End-to-End Test
Testing Firewall Environment
Production Firewall Environment

Figure 7.14. Trunk-based development requires a pipeline to deliver it to production continuously.

If your firewall rule change fails, it stops the delivery pipeline before it goes to production. You’ll notice the failure in the testing environment stage and revert the change. Everyone else can proceed with their changes to production while you implement a fix.

Trunk-based development heavily depends on delivery pipelines to test and deploy changes. The delivery pipeline should include a persistent testing environment to evaluate conflicts between changes. While a persistent testing environment incurs a cost, the environment more accurately reflects the change’s behavior in production.

The workflow for trunk-based development has very few steps. Most infrastructure teams find it helpful in making changes because it sequences the changes in some order. Trunk-based development creates a continuous feedback loop of how different changes affect each other. It also promotes a practice of making minor changes, resolving updates from the main configuration, and pushing the changes to production. When your team's changes conflict, you can quickly identify which dependencies affected the testing environment.

However, trunk-based development does require practice to resolve changes and discipline to reconcile changes. You do not isolate your changes, and working on one branch can make it challenging to collaborate. Once you work through the initial collaboration conflicts, you might find that trunk-based development provides better visibility into infrastructure as code changes across your team.

7.2.3 Choosing a branching model

I've spent hours with software and infrastructure teams debating the merits of feature-based or trunk-based development. At the conclusion of those meetings, I always came to the realization that the branching model choice depended on the team's comfort level, size, and environment setup. I will cover some limitations and concerns when applying both branching models to infrastructure.

CHALLENGES OF FEATURE-BASED DEVELOPMENT

Many open-source projects for applications and infrastructure successfully use feature-based development. Feature-based development provides the framework to test and assess critical changes independent of each other. It separates changes across many collaborators and enforces a manual review stage before merging to the main branch. Source control or continuous integration frameworks offer native integrations to support feature-based development.

Infrastructure as code gets the same benefit from feature-based development. Teams enjoy isolating infrastructure changes to a branch before pushing it to production. Figure 2.15 tests your firewall rule changes in the "TICKET-002" environment from your teammate's license changes in the "TICKET-005" environment. You can apply changes on top of your branch without conflicting with someone else.

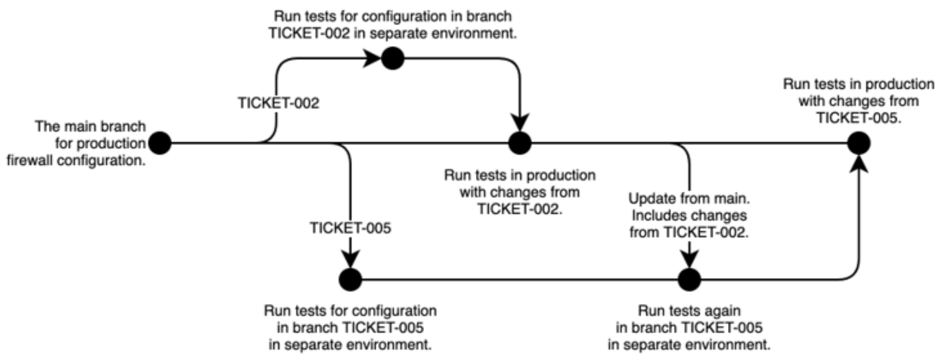

Run tests for configuration in branch TICKET-002 in separate environment.
Run tests in production with changes from TICKET-005.
TICKET-002
The main branch for production firewall configuration.
Run tests in production with changes from TICKET-002.
Update from main. Includes changes from TICKET-002.
TICKET-005
Run tests for configuration in branch TICKET-005 in separate environment.
Run tests again in branch TICKET-005 in separate environment.

Figure 7.15. You can create a new testing environment for each feature branch to verify individual changes.

However, feature-based development has a few challenges. First, new environments can cost time and money to create (refer to Chapter 12 for cost management). Your pipeline might struggle to spin up new environments when multiple team members work on the configuration with many branches.

To speed up the creation of testing environments, you could invest in runners for your pipeline framework to run tests in parallel. Alternatively, you could also create one persistent testing environment for all branches to use. However, feature-based development might cause conflicts in a persistent testing environment because each branch applies changes asynchronously.

Rather than create a persistent testing environment, you could also omit the integration and end-to-end tests from every branch except the main one to optimize costs. For example, your firewall change might only require static analysis, unit tests, and team review before merging to production in Figure 7.16. You do not need to create a unique testing environment for the branch and merge branch to production after manual review.

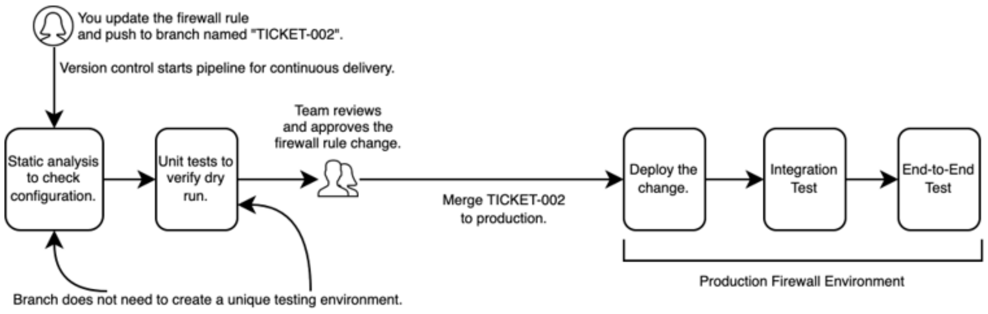

You update the firewall rule and push to branch named "TICKET-002".
Version control starts pipeline for continuous delivery.
Static analysis to check configuration.
Unit tests to verify dry run.
Team reviews and approves the firewall rule change.
Merge TICKET-002 to production.
Deploy the change.
Integration Test
End-to-End Test
Production Firewall Environment
Branch does not need to create a unique testing environment.

Figure 7.16. You can omit your branches' integration and end-to-end tests to mitigate the cost of multiple environments and pipeline concurrency.

Another challenge you face with feature-based development involves the discipline and familiarity with version control. If you haven't used version control, you need to get used to the feature branching workflow. The workflow adds the challenge of reverse-engineering merged changes and troubleshooting conflicts.

For example, someone could make a branch to fix the firewall rule over the weekend. You might not know they changed and forgot to merge the hotfix's branch. You overwrite their configuration by accident when updating the firewall rules! Over time, you accumulate a lot of branches and must troubleshoot which ones you've applied.

You'll also encounter the challenge of long-lived branches. Imagine your teammate has been working on updating the license for a month. They create a new branch named "TICKET-005" in figure 7.17. Every few days, they need to check for updates in the main branch and add them to their fix.

One day, you need to make a change that depends on your teammate's license update. Figure 7.17 shows that you start working on your changes on a branch named "TICKET-002". You finish but realize that your teammate still has work to do on "TICKET-005"! You wait two more months for your teammate to finish their firewall license update. Once they finish, you spend hours updating your "TICKET-002" branch so you can finally deploy your changes to production.

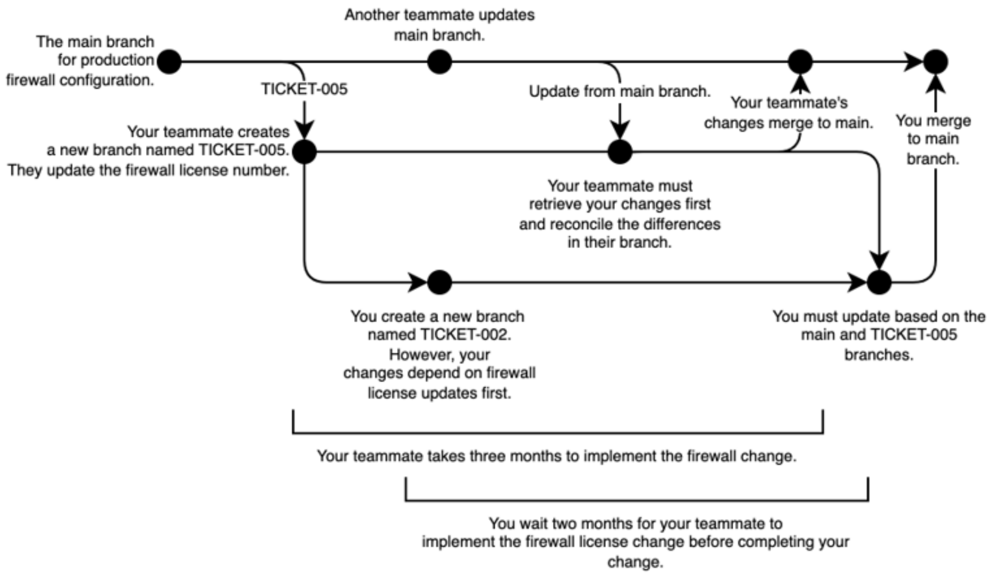

Another teammate updates main branch.
The main branch for production firewall configuration.
TICKET-005
Update from main branch.
Your teammate's changes merge to main.
You merge to main branch.
Your teammate creates a new branch named TICKET-005. They update the firewall license number.
Your teammate must retrieve your changes first and reconcile the differences in their branch.
You create a new branch named TICKET-002. However, your changes depend on firewall license updates first.
You must update based on the main and TICKET-005 branches.
Your teammate takes three months to implement the firewall change.
You wait two months for your teammate to implement the firewall license change before completing your change.

Figure 7.17. Branches with long lifetimes can prevent other changes from production and introduce complexity during the branching process.

Feature-based development encourages you to keep branches for a long time. You have to vigilantly update long-lived branches to keep up with the main one. Otherwise, you'll encounter conflicts you cannot easily resolve. On occasion, your only solution involves deleting your abandoned branch and restarting your changes on a new, updated branch.

CHALLENGES OF TRUNK-BASED DEVELOPMENT

Trunk-based development works well with infrastructure changes and the need to mitigate configuration drift between environments and states. You omit the complexities of merging and managing feature branches, especially if you need to build confidence in git skills.

Trunk-based development favors small changes instead of large, significant ones. You implement changes progressively instead of testing them in one batch. In Chapter 10, I'll cover the use of feature toggling to gradually implement a set of changes and mitigate risk to infrastructure.

Trunk-based development has a few disadvantages. It requires a dedicated testing environment before pushing changes to production. Figure 7.18 outlines the ideal workflow for trunk-based development. After you run unit tests, you deploy the change to a long-lived testing environment for integration and end-to-end tests. If the change passes tests in the testing environment, it can undergo review from your teammates. Once they approve the change, it goes to the production environment for integration and end-to-end testing.

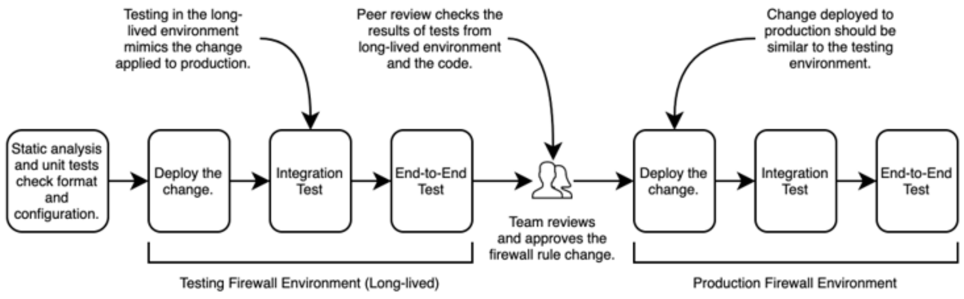

Testing in the long-lived environment mimics the change applied to production.
Peer review checks the results of tests from long-lived environment and the code.
Change deployed to production should be similar to the testing environment.
Static analysis and unit tests check format and configuration.
Deploy the change.
Integration Test
End-to-End Test
Team reviews and approves the firewall rule change.
Deploy the change.
Integration Test
End-to-End Test
Testing Firewall Environment (Long-lived)
Production Firewall Environment

Figure 7.18. Trunk-based development requires a dedicated testing environment to mimic the changes to production and help build confidence for peer review.

You need thorough unit tests to format and lint for your team's standards and integration tests to verify functionality. A persistent testing environment increases the overall cost of trunk-based development. Smaller, more modular infrastructure configurations can reduce the conflicts within a resource or module and lower the overall cost of required infrastructure for tests.

You may find that trunk-based development conflicts with manual change approval. Manual change approval can only happen after someone pushes the changes to the main branch. Your reviewer needs to know if your change worked before they can verify its formatting and configuration. If you push a broken configuration to the testing environment, you identify and revert quickly before someone else reviews it.

Table 7.3 summarizes the benefits and limitations of feature-based and trunk-based development. The choice depends on the type of infrastructure your team configures and their familiarity with version control.

Table 7.3. Comparison of feature-based and trunk-based development

Development model	Benefits	Limitations
Feature-based development	Isolates changes using branches Isolates tests using branches Organizes manual review of code Scales across multiple teams and collaborators	Requires diligence and familiarity with updating branches Encourages long-lived branches Increases cost in money and time
Trunk-based development	Provides better representation of change behavior Use one version control workflow for all changes Encourages incremental infrastructure changes to reduce blast radius	Requires long-lived testing environment Does not include a stage for manual review Requires discipline and organization to scale across multiple teams and collaborators

You must establish and agree on a development model within your team. Agreement on a development model helps promote reproducibility of changes and the overall availability of the system. Beware of the limitations with each approach, and always keep your changes as small as possible. No matter which model your team adopts, apply changes to production as frequently as possible to reduce the blast radius of changes.

7.3 Peer review

Throughout this chapter and chapter 5, I emphasized the importance of including a review step in delivery pipelines and module changes. Why should you take the time to review your teammate's infrastructure as code? What should you look for as you review it?

Peer review allows your teammates to examine your infrastructure configuration for recommendations, standards, and formatting.

> **DEFINITION** Peer review is a practice that allows your teammates or other teams to examine your infrastructure configuration for recommendations, standards, and formatting.

As a reviewer, I focus on whether a configuration will scale across teams, remain secure, or affect higher-level infrastructure dependencies. This perspective on review sometimes blocks the merging of the change to production. However, the peer review process serves as a team education opportunity for standardized practices and new patterns. You and your team may need to spend time debating the merits or drawbacks of a design or implementation.

To understand the importance and drawbacks of peer review, imagine a new inventory team needs read access to a Google Cloud Platform (GCP) project. In listing 7.1, you update the code for access management rules to read a list of users from a JSON object. The new code passes all the tests, and you wait for a few days for your teammate to review the change.

Listing 7.1 First implementation to add a new team to a GCP project

```
import json

GCP_PROJECT_USERS = [    #A
   (
       'operations',
       'group:team-operations@example.com',
       'roles/editor'
   ),
   (
       'inventory',    #B
       'group:inventory@example.com',    #B
       'roles/viewer'    #B
   )
]

class GCPProjectUsers:     #C
   def __init__(self, project, users):
       self._project = project
       self._users = users
       self.resources = self._build()    #E

   def _build(self):    #E
       resources = []
       for user, member, role in self._users:    #D
           resources.append({    #D
               'google_project_iam_member': [{    #D
                   user: [{    #D
                       'role': role,    #D
                       'member': member,    #D
                       'project': self._project    #D
                   }]    #D
               }]    #D
           })    #D
       return {
           'resource': resources
       }

if __name__ == "__main__":
   with open('main.tf.json', 'w') as outfile:    #F
       json.dump(GCPProjectUsers(    #F
           'infrastructure-as-code-book',    #F
           GCP_PROJECT_USERS).resources, outfile,    #F
           sort_keys=True, indent=2)    #G
```

#A Define a list of users and groups to add to the GCP project.
#B Add the inventory team as a read-only group to the project.
#C Create a module for the GCP project users, which uses the factory pattern to attach users to roles.
#D For each group in the list, create a Google project IAM member with the user attached to their assigned role. This resource appends the user to the roles in GCP.
#E Use the module to create the JSON configuration for the list of users to append to GCP roles.
#F Write it out to a JSON file to be executed by Terraform later.
#G When you write out the JSON file to be executed by Terraform, use an indentation of two spaces.

You wait for three days before your teammate returns with the following feedback.

- You must indent your JSON infrastructure configuration with four spaces.
- You must rename the group to "team-inventory@example.com."
- You must add the inventory team to the list of users for the "viewer" role instead of defining the role for the group.

Your teammate explains that the first two conform to team standards. The last requirement conforms to a security standard for authoritative bindings for access control (it defines a list of users for the role instead of adding a role to the user). You already delayed your change by three days waiting for peer review! Now, you need to fix it and wait another few days to get approval.

Remember in chapter 6 that you want to capture the unknown-knowns of siloed knowledge into tests. Your teammate had some knowledge that you did not know! You decide to add some unit tests to help you remember the team standards.

The new code includes some new unit tests (linting rules) to validate your team's configuration and security standards. One test checks for the correct indentation of four spaces in the JSON. Another checks that all groups conform to a naming standard. The last test checks that you use the correct resource to bind users to roles.

Listing 7.2 Adding unit tests to lint for team development standards

```
import pytest
from main import GCP_PROJECT_USERS, GCPProjectUsers    #B

GROUP_CONFIGURATION_FILE = 'main.tf.json'    #A

@pytest.fixture    #A
def json():    #A
   with open(GROUP_CONFIGURATION_FILE, 'r') as f:    #A
       return f.readlines()    #A

@pytest.fixture    #B
def users():    #B
   return GCP_PROJECT_USERS    #B

@pytest.fixture    #C
def binding():   #C
   return GCPProjectUsers(   #C
       'testing',   #C
       [('test', 'test', 'roles/test')]).resources['resource'][0]   #C

def test_json_configuration_for_indentation(json):    #A
   assert len(json[1]) - len(json[1].lstrip()) == 4, \    #A
       "output JSON with indent of 4"    #A

def test_user_configuration_for_standard_team_name(users):    #B
   for _, member, _ in GCP_PROJECT_USERS:    #B
       assert member.startswith('team-'), \    #B
           "group should always start with `team-`"    #B

def test_authoritative_project_iam_binding(binding):    #C
   assert 'google_project_iam_binding' in binding.keys(), \    #D
       "use `google_project_iam_binding` to add team members to roles"    #D
```

#A Use Python to read in the Terraform JSON configuration file. The test uses this fixture to verify that the JSON has an indentation of 4 spaces.
#B Import the list of GCP users and roles, including the inventory team, as a fixture to the test. The test checks that each user has a prefix of “team-” to identify it as a group.
#C Use a fixture to create a sample GCP project user using the factory module.
#D Check that the factory module uses the correct Terraform resource of Google project IAM binding and not members. This uses an authoritative binding to add team members to a specific role.

You correct your mistakes before peer review and shorten the feedback loop thanks to automated linting and unit tests. Your teammate doesn’t have to “nitpick” for formatting and standards. However, you and your teammate still debate whether or not you should add the user to the role or the role to the user. You decide to raise this architectural decision to the broader team for consideration.

Figure 7.19 demonstrates that effective peer review follows the example’s workflow of combining automated testing with broader architectural discussions. Between automated tests, peer review, and collaborating together, you maintain secure, resilient, and scalable infrastructure as code.

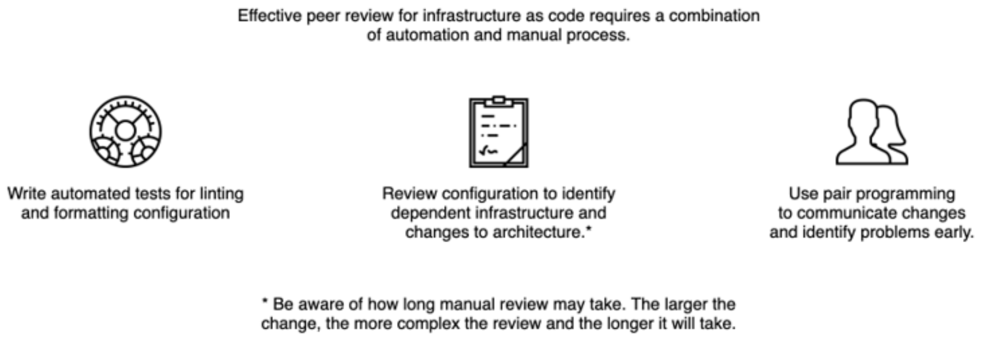

Effective peer review for infrastructure as code requires a combination of automation and manual process.
Write automated tests for linting and formatting configuration
Review configuration to identify dependent infrastructure and changes to architecture.*
Use pair programming to communicate changes and identify problems early.
* Be aware of how long manual review may take. The larger the change, the more complex the review and the longer it will take.

Figure 7.19. Automating some of the checks and maintaining awareness of any manual review processes will help expedite peer review.

However, test automation and reviewers do not catch everything. Peer review later in your development process can get frustrating as well. To address any gaps and raise architectural concerns earlier in the infrastructure as code writing process, you can program with a teammate. This technique, called ***pair programming***, uses two engineers to mitigate the friction of peer review.

> **DEFINITION** Pair programming is the practice of two programmers working together at one workstation.

One engineer may catch something the other does not realize, and vice versa. Pair programming has many challenges, including resource constraints and personality conflicts. Most companies don’t adopt it because it initially slows down the delivery pace and affects team capacity. Some individuals dislike it because their pairing partner may work at a different pace. Pair programming takes self-awareness and discipline!

Try to pair program infrastructure as code when you can. Infrastructure often includes particular terminology and institutional knowledge. For instance, current and future team members must understand *why* someone used an authoritative binding for project access control. Pair programming facilitates knowledge sharing and bakes in change review during development. Over time, your team becomes more proficient at delivering infrastructure changes quickly without the friction of manual change review.

For more on peer review and pair programming

Peer (or code) review and pair programming should promote a safe space for everyone on the team to learn how to code with best practices in mind. I do not detail how to code review or program as a pair since they require entire books. I recommend examining Google's Engineering practices at https://google.github.io/eng-practices/review/reviewer/ for more information on code review.

For more on pair programming, I recommend https://martinfowler.com/articles/on-pair-programming.html. You can use various techniques to balance the pairing relationship, such as switching the keyboard every 30 minutes or assigning a driver and navigator.

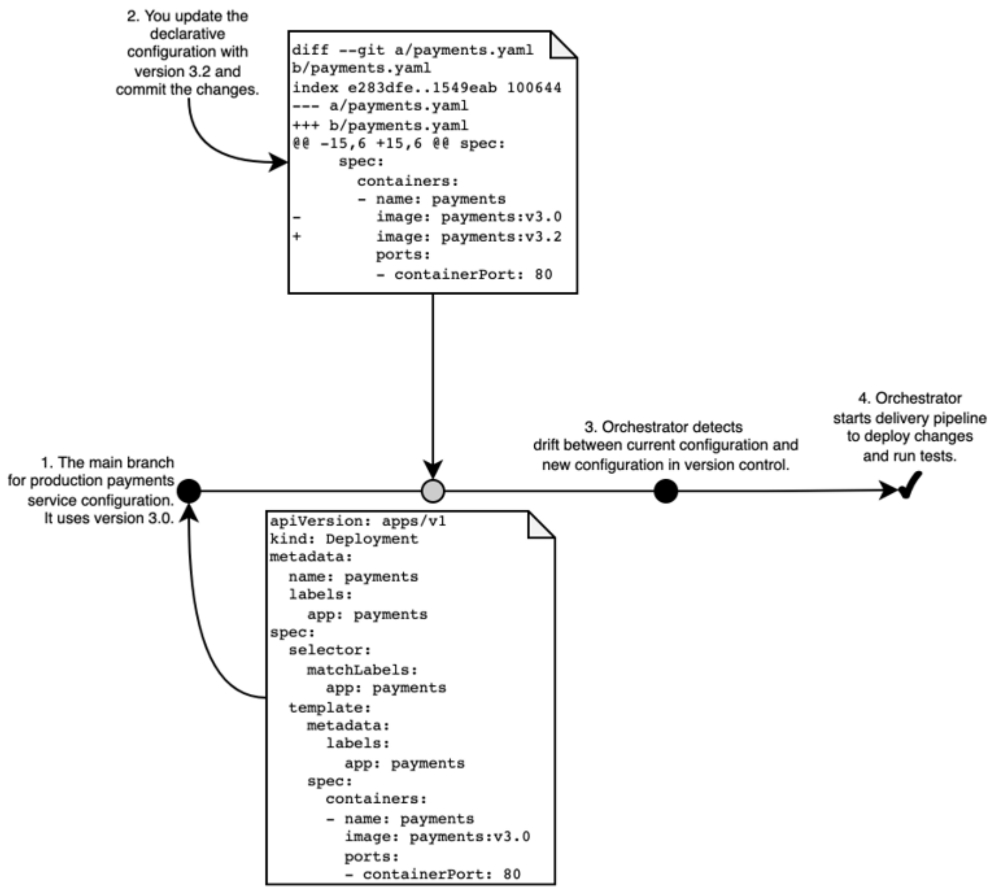
A change to infrastructure can affect the availability of business-critical systems. Batching many changes can exacerbate the failure by making it challenging to troubleshoot to one root cause. If you can shorten the peer-review process with a combination of pair programming and test automation, you can focus on reviewing the architecture and impact of infrastructure changes.

7.4 GitOps

What happens when you combine continuous deployment, declarative configuration, drift detection, and version control? All of these patterns seem fairly disparate but using them together offers an opinionated approach to managing infrastructure. You declare the configuration you want for your infrastructure, add it to version control, and deploy it to production.

Imagine you want to update a payments service from version 3.0 to 3.2. The payments service runs on a workload orchestrator (e.g., Kubernetes). The orchestrator offers a declarative configuration interface using a domain-specific language (DSL). You can pass YAML files to configure resources in the orchestrator.

Figure 7.20 implements a workflow that responds to changes by combining continuous deployment, the declarative configuration, and version control. You update the declarative configuration with version 3.2. The orchestrator detects drift between the current configuration and the one in version control. It starts a delivery pipeline to deploy the new version and run tests to check its functionality.


2. You update the declarative configuration with version 3.2 and commit the changes.
1. The main branch for production payments service configuration. It uses version 3.0.
3. Orchestrator detects drift between current configuration and new configuration in version control.
4. Orchestrator starts delivery pipeline to deploy changes and run tests.

```
diff --git a/payments.yaml
b/payments.yaml
index e283dfe..1549eab 100644
--- a/payments.yaml
+++ b/payments.yaml
@@ -15,6 +15,6 @@ spec:
     spec:
       containers:
       - name: payments
-        image: payments:v3.0
+        image: payments:v3.2
         ports:
         - containerPort: 80
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payments
  labels:
    app: payments
spec:
  selector:
    matchLabels:
      app: payments
  template:
    metadata:
      labels:
        app: payments
    spec:
      containers:
      - name: payments
        image: payments:v3.0
        ports:
        - containerPort: 80
```

Figure 7.20. An implementation of GitOps can use feature-based development to open a pull request, test changes on a branch, add reviewers, and merge the changes.

Why do you have an orchestrator detect and apply changes continuously? The orchestrator reduces the drift between the expected configuration and actual state. This ensures your system stays updated.

You might get a sense of deja vu from this workflow. After all, it combines all the practices on writing infrastructure as code, testing, and delivery. This book takes a very opinionated stance on infrastructure as code, aligning with a concept called GitOps. ***GitOps*** defines an approach that allows teams to manage infrastructure changes in version control, make declarative changes via infrastructure as code, and continuously deploy updates to infrastructure.

DEFINITION GitOps is an approach that uses declarative infrastructure as code to manage infrastructure changes through version control and continuously deploy them to production.

You'll associate GitOps most frequently with the Kubernetes ecosystem. However, GitOps offers an opinionated paradigm for scaling infrastructure as code practices across an organization. You no longer implement changes by filling tickets with the relevant details.

Instead, anyone in the organization can branch infrastructure as code and commit changes. Continuous deployment reduces drift, keeps infrastructure updated, and always runs the tests. You can track who requested and made the change through the pull request and commit history.

7.5 Exercises and Solutions

Exercise 7.1

Choose a standard infrastructure change in your organization. What do you need in order to confidently continuously deliver the change to production? What about continuous deployment? Outline or diagram stages in your delivery pipelines for both.

Answer:

- As you go through this exercise, consider the following:
- Do you have unit, integration, or end-to-end tests?
- What kind of branching model do you use?
- Does your company have compliance requirements? For example, two people must approve the change before production.

What happens when someone needs to make a change?

7.6 Summary

- Delivering infrastructure changes to production usually involves a change review process, which manually verifies the architecture and impact of the change.
- Continuous integration involves frequently merging changes to the main branch of infrastructure configuration.
- Continuous delivery deploys changes to a testing environment for automated testing and waits for manual approval before pushing them to production.
- Continuous deployment directly deploys changes to production without a manual approval stage.
- You can use continuous integration, delivery, or deployment to push infrastructure as code changes with automated tests, depending on the type of change and its frequency.
- Your team can collaborate on infrastructure as code using feature-based development or trunk-based development.
- Feature-based development creates a branch for each change, enabling isolated testing but requiring familiarity with version control practices.
- Trunk-based development applies all changes to the main branch, which identifies conflicts between changes but requires a testing environment before production.
- You can automate checks for formatting and standards and manually review the configuration for architecture and dependencies.
- Pair programming can help identify conflicts and problems with a change earlier in the development process.
- GitOps incorporates version control, declarative infrastructure configuration, and continuous deployment to empower anyone to automate infrastructure changes through code commits.

8 Security and compliance

This chapter covers

- **Choosing protections for credentials and secrets in infrastructure as code**
- **Implementing policies to enforce compliant and secure infrastructure**
- **Preparing end-to-end tests for security and compliance**

In previous chapters, I alluded to the importance of securing infrastructure as code and checking its conformance with your organization's security and compliance requirements. Oftentimes, you don't address these requirements until later in your engineering process. By that point, you may have already deployed an insecure configuration or violated a compliance requirement about data privacy!

For example, imagine you work for a retail company called uDress. Your team has six months to build a new frontend application on Google Cloud Platform (GCP). The company needs it available by the holiday season. Your team works very hard and develops enough functionality to go live. However, a month before you deploy and test the new application, the compliance and security team perform an audit - and you fail.

Now, you have new items in your backlog to fix the security and compliance issues and adhere to company policy. Unfortunately, these fixes delay your delivery timeline or, at worst, break functionality. You might wish that you knew about these from the very beginning, at least so you could plan for them!

Your company's ***policy*** ensures that systems comply with security, audit, and organizational requirements. In addition, your security or compliance teams often define policies based on industry, country, and more.

> **DEFINITION** **Policy is a set of rules and standards in your organization to ensure compliance to security, industry, or regulatory requirements.**

This chapter will teach you to protect credentials and secrets and write tests to enforce policies for security and compliance. If you think about these practices before you write infrastructure as code, you build secure, compliant infrastructure and avoid delays in your delivery timeline. Inspired by a manager I used to work with, "We're baking security into the infrastructure instead of icing it later."

8.1 Managing access and secrets

We already introduced the idea of "baking" security into infrastructure as code in Chapter 2. Infrastructure as code uses two sets of secrets. You use API credentials to automate infrastructure and sensitive variables such as passwords to pass to resources. You can store both secrets in secrets managers to handle their protection and rotation.

In this section, I focus on securing infrastructure as code delivery pipelines. Infrastructure as code expresses the expected state of the infrastructure, which often includes root passwords, usernames, private keys, and other sensitive information. Infrastructure delivery pipelines control the deployment and release of infrastructure that needs this information.

Let's imagine you built delivery pipelines for the new uDress system to deploy infrastructure. The pipelines use a set of infrastructure provider credentials to create and update resources. It also reads a database password from a secrets manager and passes it as an attribute to create the database.

Your security team points out two problems with your approach. First, the infrastructure delivery pipeline uses full administrative credentials to configure GCP. Second, your team's delivery pipeline accidentally prints out the root database password in its logs!

Your delivery pipeline just increased the ***attack surface*** (sum of the different points of attack) of your system.

DEFINITION **The attack surface describes the sum of different points of attack where an unauthorized user can compromise a system.**

Anyone can use the administrative credentials or root database password to gain information and compromise your system. You need a solution to better secure the credentials and the database password. The solution should hopefully minimize the attack surface.

8.1.1 Principle of least privilege

Infrastructure as code delivery pipelines have points of attack that allow unauthorized users to use credentials with elevated access. For example, uDress used Chapter 7 to build a pipeline that continuously delivers infrastructure changes to production. The pipeline needs some permissions to change infrastructure in GCP.

Initially, the team gave the pipeline full administrative credentials so it could create all of the resources in GCP. If someone accessed those credentials, they could create and update anything in the uDress system. Someone could exploit your team's pipeline to run machine learning models or access other customer data!

The pipeline does not *need* access to every resource. You decide to update the credentials so it only uses the minimal set of permissions it needs to update specific resources. You determine that infrastructure as code only creates network, App Engine, and Cloud SQL resources. You remove administrative access from the credentials and replace them with write access to the three resources.

When the pipeline runs in figure 8.1, the new credentials have just enough access to update the three sets of resources. It also retrieves the database password from the secrets manager before deploying updates to the network, application, and database. After deploying the changes to a testing environment, you add a unit test to verify that the credentials no longer have administrative access.

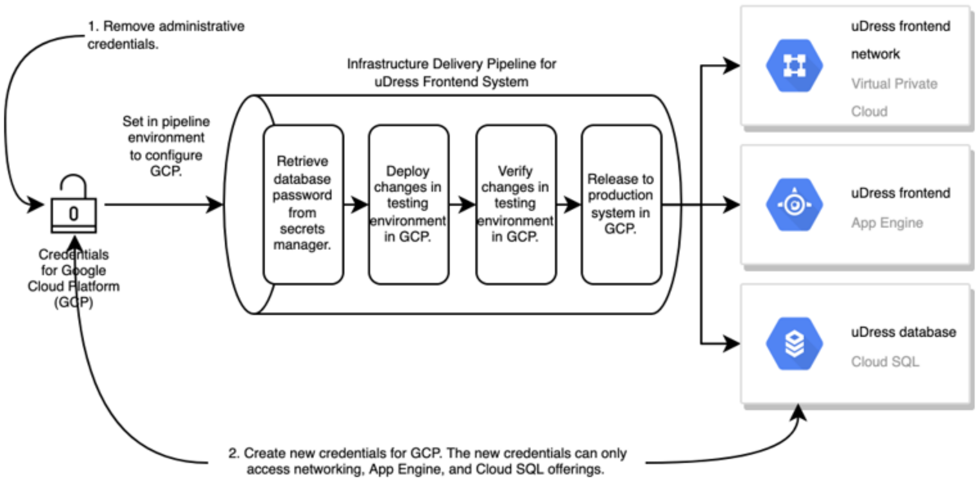

1. Remove administrative credentials.
Infrastructure Delivery Pipeline for uDress Frontend System
Set in pipeline environment to configure GCP.
0
Credentials for Google Cloud Platform (GCP)
Retrieve database password from secrets manager.
Deploy changes in testing environment in GCP.
Verify changes in testing environment in GCP.
Release to production system in GCP.
uDress frontend network
Virtual Private Cloud
uDress frontend
App Engine
uDress database
Cloud SQL
2. Create new credentials for GCP. The new credentials can only access networking, App Engine, and Cloud SQL offerings.

Figure 8.1. Remove administrative credentials from the uDress frontend delivery pipeline and limit them to network, application, and database access.

You remediated the security concern of pipeline credentials using the principle of least privilege. The ***principle of least privilege*** ensures that a user or service account only gets the minimum access they require to complete their task.

> **DEFINITION** The principle of least privilege means that users or service accounts should have minimum access requirements to a system. They should only have enough to complete their tasks.

Maintaining the principle of least privilege takes some time and effort. You usually change access as you add new resources to infrastructure as code. In general, attach roles to delivery pipeline credentials. Grouping access permissions into roles helps promote composability so you can add and remove access as needed.

Apply the module practices from chapter 3 to offer modules of permission sets. For example, you may offer a factory module for uDress's web applications to customize network, application, and database write access. Any web application can use the module and properly reproduce the minimum set of privileges it needs.

Let's use the access management module to implement least privilege access management for uDress's frontend delivery pipeline. You limit the pipeline to network, application, and Cloud SQL administrative credentials. These credentials allow the pipeline to create, delete, and update the network, application, and database but not update any other resource types.

Listing 8.1 Least privilege access management policy for frontend

```
import json
import iam    #A


def build_frontend_configuration():
   name = 'frontend'
   roles = [
       'roles/compute.networkAdmin',    #B
       'roles/appengine.appAdmin',    #B
       'roles/cloudsql.admin'    #B
   ]

   frontend = iam.ApplicationFactoryModule(name, roles)    #A
   resources = {
       'resource': frontend._build()    #C
   }
   return resources


if __name__ == "__main__":
   resources = build_frontend_configuration()    #C

   with open('main.tf.json', 'w') as outfile:    #D
       json.dump(resources, outfile, sort_keys=True, indent=4)   #D
```

#A Import the application access management factory module to create access management roles for the frontend application.
#B Create the role configuration based on a list of roles for a service account, including networking, App Engine, and Cloud SQL.
#C Use the method to create the JSON configuration for the pipeline's access permissions.
#D Write it out to a JSON file to be executed by Terraform later.

AWS equivalent

Google App Engine is similar to AWS Elastic Beanstalk, which deploys web applications and services to provider-managed infrastructure. Google Cloud SQL is similar to AWS relational database service, which deploys managed databases to providers.

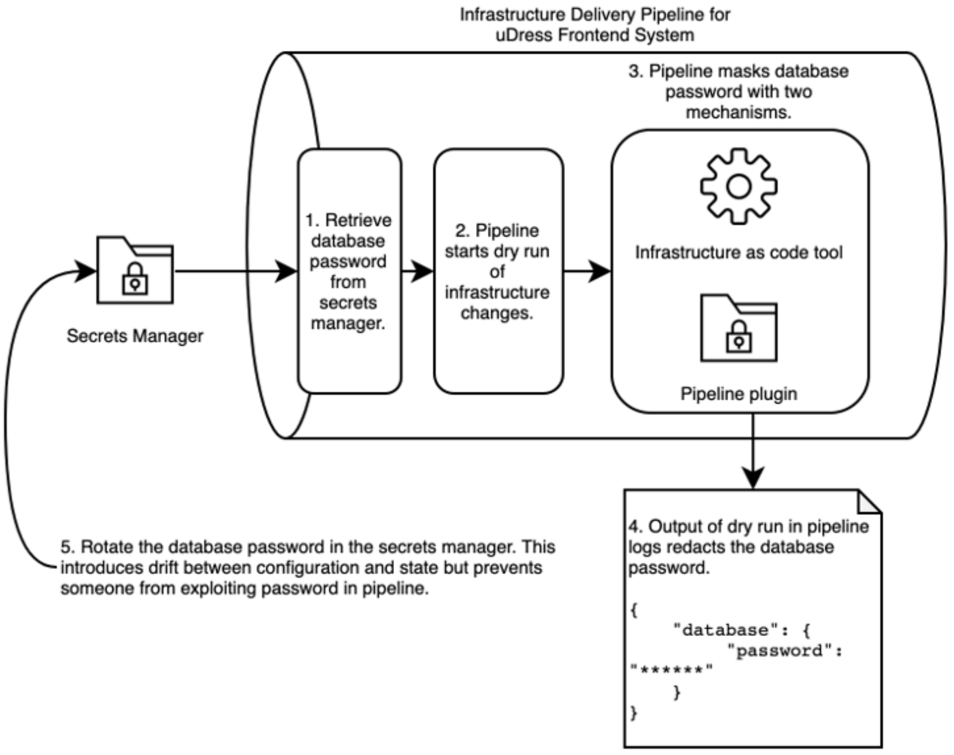
As you adhere to the principle of least privilege, take care when you remove permissions. Sometimes, a pipeline needs more specific permissions to read or update dependencies. You can break infrastructure or applications if they do not have sufficient permissions.

Some infrastructure providers, including GCP, analyze the permissions used for a service account or user and output a set of excess permissions. You can also run other third-party tools to analyze access and identify unused permissions. I recommend using these tools to check and update your access control each time you add a new infrastructure resource.

8.1.2 Protecting secrets in configuration

Besides using administrative credentials from a pipeline to access an infrastructure provider, someone could alter the pipeline to print out sensitive information about infrastructure. For example, the frontend delivery pipeline outputs the root database password in the logs. Anyone accessing the logs from the pipeline can use the root password to log into the database!

To address this security concern, you decide to mark the password as a *sensitive variable* using your infrastructure as code tool. The tool redacts the password in the logs. You also *install a plugin* in your pipeline tool to identify and redact any sensitive information, such as the password. You add these two configurations to your pipeline in figure 8.2 to avoid compromising the database password in the pipeline logs. As a safety precaution, you rotate the database password in the secrets manager.


Infrastructure Delivery Pipeline for uDress Frontend System
3. Pipeline masks database password with two mechanisms.
Secrets Manager
1. Retrieve database password from secrets manager.
2. Pipeline starts dry run of infrastructure changes.
Infrastructure as code tool
Pipeline plugin
4. Output of dry run in pipeline logs redacts the database password.
5. Rotate the database password in the secrets manager. This introduces drift between configuration and state but prevents someone from exploiting password in pipeline.

```
{
    "database": {
          "password":
"******"
    }
}
```

Figure 8.2. You can protect the root database password by using tools to mask the value and rotating the credential after applying changes to infrastructure as code.

You can use tools to ***mask*** the password in the delivery pipeline by either suppressing or redacting the plaintext information.

> **DEFINITION** Masking your sensitive information means suppressing or redacting its plaintext format to prevent someone from reading the information.

Using one or both mechanisms will prevent the sensitive information from appearing in the pipeline logs. Sensitive information can include passwords, encryption keys, or infrastructure identifiers like IP addresses. If you think someone can use the information to gain access to your system, consider redacting or masking the value in your pipeline.

However, masking sensitive information doesn't guarantee protection from unauthorized access. You still need a workflow to *remediate* the exposed credentials as quickly as possible. As a solution, store and rotate the credentials with a secrets manager instead of defining it in your infrastructure as code.

Separately managing secrets introduces mutability, or in-place changes, to your infrastructure as code. While it introduces drift between the actual root database password and the one expressed in infrastructure as code, managing the password mutably prevents someone from exploiting the infrastructure as code pipeline and using the credentials.

As you build infrastructure as code, think about the checklist of security requirements in your delivery pipeline to minimize its attack surface.

1. Check for *least privilege access* for infrastructure provider credentials from the beginning. You should provide enough permissions to apply and secure your infrastructure as code.
2. Generate a secret *using a function* to generate a random string or read the secret from a secrets manager. Avoid passing secrets as static variables to your configuration.
3. Check that your pipeline *masks sensitive configuration data* in its dry run capability or command outputs.
4. Provide a mechanism to *revoke and rotate compromised credentials* or data quickly.

You can solve many of the requirements in the checklist with a secrets manager. The secrets manager can commit the need for statically defining secrets in configuration. While some requirements serve as general security practices for delivery pipelines, they also apply to secure infrastructure as code. You can review Chapter 2 for the pattern of securing secrets with a secrets manager.

8.2 Tagging infrastructure

After securing your infrastructure, you have the challenge of running and supporting it. Operating infrastructure requires a set of troubleshooting and auditing patterns and practices. As you continue to add infrastructure to your system, you need a way to identify the purpose and life cycle of resources.

Imagine the uDress frontend application goes live! However, your team gets a message from the finance team. Your infrastructure provider billing has exceeded the expected budget for the past two or three months. You search in the provider’s interface to determine which resources have contributed most to the cost. How do you know the owner and environment of each resource?

GCP offers the use of labels, which allows you to add metadata to your resources for identification and audit purposes. You update these labels to include owner and environment. In figure 8.3, uDress includes identification of owner and environment, standards for tag format, and automation metadata. You decide to dash-delimit tag names and values so the tags work with GCP.

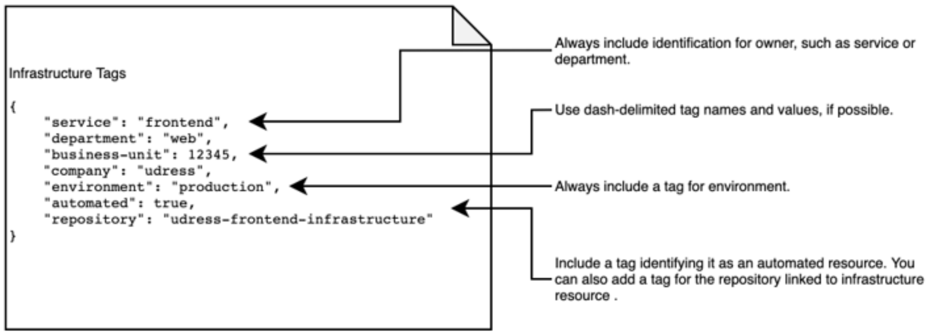

Infrastructure Tags
{
"service": "frontend",
"department": "web",
"business-unit": 12345,
"company": "udress",
"environment": "production",
"automated": true,
"repository": "udress-frontend-infrastructure"
}
Always include identification for owner, such as service or department.
Use dash-delimited tag names and values, if possible.
Always include a tag for environment.
Include a tag identifying it as an automated resource. You can also add a tag for the repository linked to infrastructure resource .

Figure 8.3. Tags should include identification of owner, environment, and automation for easy troubleshooting.

Outside of GCP, other infrastructure providers allow you to add metadata to identify resources. In your organization, you'll develop a ***tagging strategy*** to define a standard set of metadata used for auditing your infrastructure system.

DEFINITION A tagging strategy defines a set of metadata (also known as tags) used for auditing, managing, and securing infrastructure resources in your organization.

Why use metadata in the form of tags? Tags help you search and audit resources, actions necessary for billing and compliance. You can also use tags to do bulk automation of infrastructure resources. Bulk automation includes clean-up or break-glass (manual changes to stabilize or fix system failures) updates to a subset of resources.

Let's implement standard tags for uDress. From Chapter 3, you apply the prototype pattern to define a list of standard tags for your uDress. You reference the uDress tag module to create a list of labels for a Google Cloud Platform server in your code.

Listing 8.2 Use tags module to set standard tags for server

```
class TagsPrototypeModule():    #A
   def __init__(
           self, service, department,
           business_unit, company, team_email,
           environment):
       self.resource = {    #B
           'service': service,    #B
           'department': department,    #B
           'business-unit': business_unit,    #B
           'company': company,    #B
           'email': team_email,    #B
           'environment': environment,    #B
           'automated': True,    #B
           'repository': f"${company}-${service}-infrastructure"    #B
       }    #B
```

```
class ServerFactory:
   def __init__(self, name, network, zone='us-central1-a', tags={}):
       self.name = name
       self.network = network
       self.zone = zone
       self.tags = TagsPrototypeModule(    #C
           'frontend', 'web', 12345, 'udress',    #C
           'frontend@udress.net', 'production')    #C
       self.resource = self._build()

   def _build(self):     #D
       return {
           'resource': [
               {
                   'google_compute_instance': [     #E
                       {
                           self.name: [
                               {
                                   'allow_stopping_for_update': True,
                                   'boot_disk': [
                                       {
                                           'initialize_params': [
                                               {
                                                   'image': 'ubuntu-1804-lts'
                                               }
                                           ]
                                       }
                                   ],
                                   'machine_type': 'f1-micro',
                                   'name': self.name,
                                   'network_interface': [
                                       {
                                           'network': self.network
                                       }
                                   ],
                                   'zone': self.zone,
                                   'labels': self.tags     #E
                               }
                           ]
                       }
                   ]
               }
           ]
       }
```

#A The tag module uses the prototype pattern to define a standard set of tags.
#B Set tags to identify owner, department, business unit for billing, and repository for the resource.
#C Pass the required parameters to set tags for the frontend application.
#D Use the module to create the JSON configuration for the server.
#E Create the Google compute instance (server) using a Terraform resource and include the tags from the tag module as labels.

How do you know which tags to add? Recall from Chapter 2 that you must standardize the naming *and* tagging of your infrastructure resources. Discuss these considerations with compliance, security, and finance teams. It will help determine which tags you need and how to use them. At a minimum, I *always* have a tag for the following:

- Service or team
- Team email or communication channel
- Environment (development or production)

For example, let’s say the uDress security team audits the frontend resources and discovers some misconfigured infrastructure. They can check the tags, identify the service and environment with the problem, and reach out to the team email.

You may also include tags for:

- Automation, which helps you identify manually created resources from automated ones
- Repository, which allows me to correlate the resource with its original configuration in version control
- The business unit, which identifies the billing or chargeback identifier for accounting
- Compliance, which identifies if the resource has compliance or policy requirements for handling personal information.

As you decide on your tagging, make sure it conforms to a general set of constraints so you can apply the same tags across any infrastructure provider. Most infrastructure providers have character restrictions on tags. I usually prefer *dash-case*, which uses lowercase tag names and values split with hyphens. While you can use camel case (stylistically, camelCase), not all providers have case-sensitive tagging.

Tag character limits also vary depending on the infrastructure provider. Most providers support a *maximum length* of 128 characters for the tag key and 256 characters for the tag value. You will have to balance the verbosity of descriptive names (from Chapter 2) with the provider’s tag limits!

Another part of your tagging strategy involves deciding whether or not you *delete untagged resources*. Consider enforcing tags for all resources in the production environment. The testing environment can support untagged resources for manual testing. In general, I do not recommend immediately deleting untagged resources without careful examination. You don’t want to delete an essential resource by accident.

8.3 Policy as code

Securing access and secrets in infrastructure delivery pipelines and managing tags in infrastructure providers can improve security and compliance practices. However, you might wish to identify insecure or non-compliant infrastructure configuration *before* it goes to production. You’d like to catch a problem before someone finds it in your production system.

Imagine connecting the uDress frontend application to another database. You open a firewall rule to allow all traffic inbound to a managed database for testing. After testing, you expect to remove the database, so you do not tag it.

You forget about the firewall and tag configuration and send it off for review. Unfortunately, your teammate misses them in code review and pushes the changes to production. Two weeks later, you discover that an unknown entity has accessed some data! However, you have no tags to identify the compromised database.

What could you have done differently? Recall the importance of unit tests or static analysis of infrastructure configuration in Chapter 6. You can apply the *same* techniques to write tests specifically for security and policy.

Rather than depend on a teammate to catch the problem, you can express policy *as code* to statically analyze the configuration for the permissive firewall rule or lack of tags. ***Policy as code*** tests some infrastructure metadata and verifies if it complies with security or compliance requirements.

> **DEFINITION** Policy as code (also known as shift-left security testing or static analysis of infrastructure as code) infrastructure metadata and verifies if values comply with security or compliance requirements before pushing changes to production. Policy as code includes the rules you write for dynamic analysis tools or vulnerability scanning.

I discussed the long-term benefit of automating and testing infrastructure as code in chapter 1 and 6. You similarly have an initial short-term time investment for writing policy as code. The policy checks continuously verify the compliance of each change you want to make to production. You minimize the surprises after the compliance and security teams audit your system. Over time, you decrease the long-term time investment with a shorter time to production.

8.3.1 Policy engines and standards

Tools can help run policy as code by evaluating metadata based on a set of rules. Most testing tools in this space use a policy engine. A ***policy engine*** takes policies as input and evaluates infrastructure resources for compliance.

> **DEFINITION** A policy engine takes policies as the input and evaluates resource metadata for compliance to policies.

Many policy engines parse and check fields in infrastructure configuration or state. In figure 8.4, a policy engine extracts JSON or other metadata from the infrastructure as code or system state. Then, it passes the metadata to a security or policy test. The engine runs the test to parse fields, check their values, and fail if the actual values do not match the expected values.

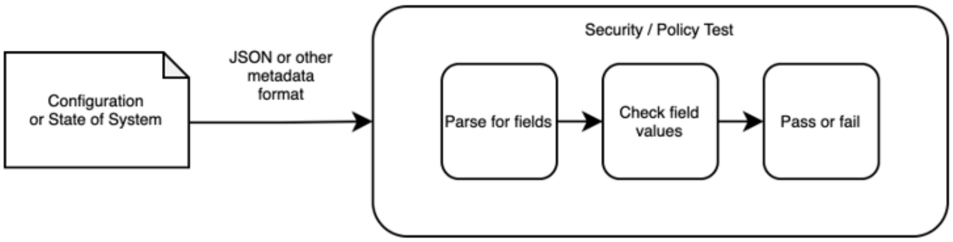


Figure 8.4. Tests for security and policy parse the configuration or state of the system for the correct field values and fail if they do not match an expected value.

This workflow applies to policy as code tools *and* any tests you write yourself. Policy as code tools make it more straightforward to test for values because it abstracts the complexity of parsing for fields and checking the values. However, they don't cover every value you want to test.

As a result, you usually write your own policy engine to suit your purposes. In the examples for this chapter, I use pytest, a Python testing framework, as a primitive "policy engine" to check for a secure and compliant configuration.

POLICY ENGINES

The policy as a code ecosystem has different tools for different purposes. Most tools fall into one of three use cases, all of which address very different functions and vary widely in behavior, including the following:

1. Security tests for specific platforms
2. Policy tests for industry or regulatory standards
3. Custom policies

Table 8.1 includes a non-exhaustive list of policy engines for *provisioning* tools, both vendor and open source. I outlined a few of the technology integrations and the use case category for each tool.

Table 8.1. Examples of policy engines for provisioning tools

Tool	Use Case(s)	Technology Integration(s)
Amazon Web Services CloudFormation Guard	Security tests for specific platforms Custom policies	Amazon Web Services CloudFormation
HashiCorp Sentinel	Security tests for specific platforms Custom policies	HashiCorp Terraform
Pulumi CrossGuard	Security tests for specific platforms Custom policies	Pulumi SDK
Open Policy Agent (Underlying technology for Fugue, conftest, Kubernetes Gatekeeper, and more)	Security tests for specific platforms (tool-dependent) Policy tests for industry or regulatory standards (tool-dependent) Custom policies	Various[1]
Chef Inspec	Security tests for specific platforms Custom policies	Various[2]
Kyverno	Security tests for specific platforms Custom policies	Kubernetes

You often need to *mix and match* tools to cover all use cases. No single tool covers all use cases. Some tools offer customization, which you can use to build policies of your own. In general, consider extending an existing tool with custom policies so you can establish opinionated patterns and defaults with your security, compliance, and engineering teams. In reality, you probably adopt five or six different policy engines to cover the tools, platforms, and policies you need.

Note that I do not include any security or policy tooling specific to datacenter appliances, which often depend on your organization's procurement requirements. You may also find some community projects outside of the examples listed in the table! I often find these tools and their integrations replaced by newer ones since the ecosystem changes rapidly.

[1] For a complete list, see https://www.openpolicyagent.org/docs/latest/ecosystem/.
[2] For a complete list, search the Chef marketplace at https://supermarket.chef.io/.

Image building and configuration management

Image building tools do not have too many security or policy tools, as you tend to write your tests for them. Configuration management tools follow a similar approach to provisioning tools. You will need to find community or built-in tools that verify security and policy configuration.

INDUSTRY OR REGULATORY STANDARDS

You might examine table 8.1 and discover that very few tools include policy tests for industry or regulatory standards. Most of these policies exist in documentation form, and you often have to write them yourselves. On occasion, you can find policy test suites created by the community that you'll need to augment with your own.

For example, the National Institute of Standards and Technology (NIST) in the United States publishes a list of security benchmarks as part of the National Checklist Program (https://ncp.nist.gov/repository). A reviewer for this book also recommended Security Technical Implementation Guides (STIGs) from the U.S. Department of Defense, including technical testing and configuration standards.

You're missing a helpful tool or standard!

Yes, I am missing many tools or standards in this section. The standards I included apply to the United States and not necessarily worldwide. By the time you read this, policy engines will have changed features, integrations, or open-source status, and the industry or regulatory standards will have updated drafts. If you'd like to recommend one, please let me know at https://github.com/joatmon08/tdd-infrastructure!

8.3.2 Security tests

What should you test to secure your infrastructure? Some policy as code tools offer opinionated defaults that capture best practices for a secure system. However, you might need to write your own for your company's specific platforms and infrastructure.

Let's start fixing your database security breach. Fortunately, the testing data did not have anything important. However, in the future, you don't want your teammate copying and deploying the configuration to production. To prevent the testing environment's infrastructure as code from deploying to production, you write a test to secure a network for a database.

The database needs a very restrictive, least-privilege (minimum access) firewall rule. Figure 8.5 shows how you implement a test to retrieve the firewall configuration from infrastructure as code. The configuration goes to a test, which parses the source range from the firewall rule. If the range contains a permissive rule, "0.0.0.0/0", the test fails.

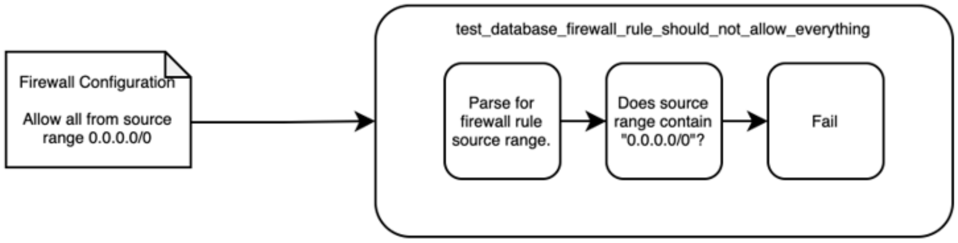

Firewall Configuration
Allow all from source range 0.0.0.0/0
test_database_firewall_rule_should_not_allow_everything
Parse for firewall rule source range.
Does source range contain "0.0.0.0/0"?
Fail

Figure 8.5. Retrieve the source range value from the firewall rule configuration and determine if it contains an overly permissive range.

GCP uses "0.0.0.0/0" to denote that any IP address can access the database. If someone gains access to your network, they can access your database if they have the username and password. Your new test fails before an overly permissive rule like "0.0.0.0/0" goes to production.

Listing 8.3 implements the test for the firewall rule in Python. In your test, you implement code to open the JSON configuration file, retrieve the "source_ranges" list, and check if the list contains "0.0.0.0/0".

Listing 8.3 Using a test to parse the firewall rule for 0.0.0.0

```
import json
import pytest
from main import APP_NAME, CONFIGURATION_FILE


@pytest.fixture(scope="module")
def resources():
   with open(CONFIGURATION_FILE, 'r') as f:    #A
       config = json.load(f)    #A
   return config['resource']     #F


@pytest.fixture
def database_firewall_rule(resources):    #B
   return resources[0][    #B
       'google_compute_firewall'][0][APP_NAME][0]    #B


def test_database_firewall_rule_should_not_allow_everything(    #D
       database_firewall_rule):
   assert '0.0.0.0/0' not in \    #C
       database_firewall_rule['source_ranges'], \    #C
       'database firewall rule must not ' + \    #E
       'allow traffic from 0.0.0.0/0, specify source_ranges ' + \    #E
       'with exact IP address ranges'    #E
```

#A Load the infrastructure configuration from a JSON file.

#B Parse the Google compute firewall resource defined by Terraform from the JSON configuration.
#C Check that “0.0.0.0/0”, or allow all, is not defined in the rule’s source ranges.
#D Use a descriptive test name explaining the policy for the firewall rule, which should not allow all traffic.
#E Use a descriptive error message describing how to correct the firewall rule, such as removing “0.0.0.0/0” from source ranges.
#F Parse the resource block out of the JSON configuration file.

AWS equivalent

A firewall rule in GCP is equivalent to an AWS security group.

Imagine your new teammate wants to run some tests on the database from their laptop. They make a change to open the firewall rule to “0.0.0.0/0” in infrastructure as code. The pipeline runs the Python code to generate JSON.

```
$ python main.py
```

The pipeline runs unit tests checking the JSON file with the configuration. It recognizes the firewall rule contains “0.0.0.0/0” in the list of allowed source ranges and throws an error!

```
$ pytest test_security.py
=========================== short test summary info ===========================
FAILED test_security.py::test_database_firewall_rule_should_not_allow_everything -
       AssertionError: database firewall rule must not allow traffic from 0.0.0.0/0,
       specify source_ranges with exact IP address ranges
=========================== 1 failed in 0.04s ===========================
```

Your teammate reads the error description and realizes that the firewall rule should not allow all traffic. They can correct their configuration to add their laptop IP address to source ranges.

Just like functional tests in chapter 6, security tests *educate* the rest of your team on ideal secure practices for infrastructure. While the tests do not necessarily catch all security violations, they communicate important information about security expectations. Moving the *unknown knowns* of security best practices to *known knowns* eliminates a repeated mistake.

These tests also help scale security practices in your organization. Your teammate feels empowered to correct the configuration. Furthermore, your security team has fewer investigations and follow-ups for security violations. Making security part of everyone’s responsibility reduces the time and effort for future remediation.

Positive versus negative testing

In the example of the database IP address range, you checked that an IP address range does *not* match every IP address (0.0.0.0/0). Called ***negative testing***, it asserts that the value does not match. You can also use ***positive testing*** to assert that attributes do match an expected value.

Some references suggest that you express all security or policy tests with one type. However, I usually write tests with both positive and negative testing assertions. The combination better expresses the intent of the security and policy requirement. For example, you can use the negative test to check for any IP address range against *any* infrastructure configuration written by any team. On the other hand, if you have an IP address range that every firewall rule *must include*, such as a VPN connection, you can use a positive test.

You can write tests to check other secure configurations, including:

- Ports, IP ranges, or protocols on other network policy
- Access control for no administrative or root access of infrastructure resources, servers, or containers
- Metadata configuration to mitigate exploitation of instance metadata
- Access and audit logging configuration for security information and events management (SIEM), such as for load balancers, identity and access management, or storage buckets
- Package or configuration versions for fixed vulnerabilities.

This non-exhaustive list covers some general configurations. However, you should consult with your security team or other industry benchmarks for additional information and tests.

8.3.3 Policy tests

Security tests verify that you minimize the attack surface of misconfiguration in your infrastructure as code. However, you need other tests for auditing, reporting, billing, and troubleshooting. For example, your testing database should have a tag on it so someone can identify its owner and report the security breach.

The uDress compliance team reminds you to add tags to your GCP database so they can identify the database owner. They also notify you that the security breach caused the database resources to scale, which increased your cloud computing bill. Without tags, they had a difficult time identifying who to contact about the security problem and the increased bill.

You add tags to the database configuration. To remind yourself of tagging in the future, you use figure 8.6 to implement a unit test to check for tags. Like the security test for the firewall rule configuration, you parse a JSON file with the database configuration to check that you correctly filled the labels with tags. If the test has empty labels, the test fails.

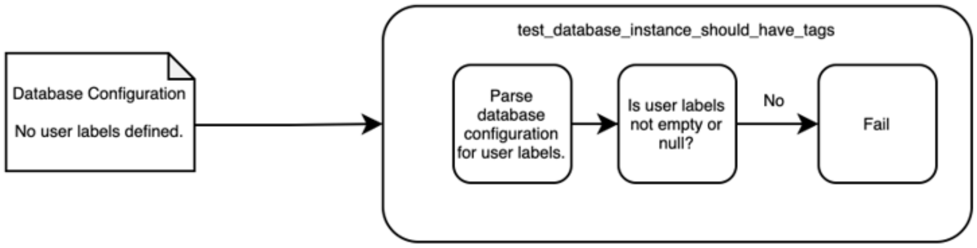

Database Configuration
No user labels defined.
test_database_instance_should_have_tags
Parse database configuration for user labels.
Is user labels not empty or null?
No
Fail

Figure 8.6. You implement a test that parses the database configuration and checks for a list of tags in the database’s user labels.

The policy test behaves similarly to the security test. However, it tests the tags instead of the IP source ranges. While the policy does not better secure the infrastructure, it improves your ability to troubleshoot and identify resources.

Let’s implement the test workflow. You write a test to check that you more than zero tags under the GCP “user_labels” parameter.

Listing 8.4 Using a test to parse the database configuration for tags

```
import json
import pytest
from main import APP_NAME, CONFIGURATION_FILE

@pytest.fixture(scope="module")
def resources():    #A
   with open(CONFIGURATION_FILE, 'r') as f:    #A
       config = json.load(f)    #A
   return config['resource']    #F

@pytest.fixture
def database_instance(resources):
   return resources[2][    #B
       'google_sql_database_instance'][0][APP_NAME][0]    #B


def test_database_instance_should_have_tags(database_instance):    #D
   assert database_instance['settings'][0]['user_labels'] \    #C
       is not None    #C
   assert len(    #E
       database_instance['settings'][0]['user_labels']) > 0, \    #E
       'database instance must have `user_labels`' + \    #E
       'configuration with tags'    #E
```

#A Load the infrastructure configuration from a JSON file.
#B Parse user labels in the Google SQL database instance defined by Terraform from the JSON configuration.
#C Check that the user labels on the database do not have an empty list or null value.
#D Use a descriptive test name explaining the policy for tagging the database.
#E Use a descriptive error message describing adding the tags to GCP user labels.

#F Parse the resource block out of the JSON configuration file.

You add the test to your security tests. The next time your teammate makes a change and forgets tags, the test fails. They read the error message and correct their infrastructure as code to include the tags.

You can implement tests for other organizational policies, including:

- Required tags for all resources
- Number of approvers for a change
- The geographic location of an infrastructure resource
- Log outputs and target servers for auditing
- Separate development data from production data.

This non-exhaustive list covers some general configurations. However, you should consult with your compliance team or other industry benchmarks for additional information and tests. As you write your tests, ensure you include clear error messages outlining which policies the test checks.

8.3.4 Practices and patterns

As you write more security and policy tests, you gain confidence that your configuration remains secure and compliant. How can you teach this across your team and company? You can apply some of the practices and patterns for testing to checking infrastructure security and compliance. I'll cover the practices and patterns for writing security and policy tests in greater detail.

USE DETAILED TEST NAMES AND ERROR MESSAGES

You'll notice *detailed test names and error messages* for the uDress policy and security tests. They seem very verbose but communicate to teammates precisely what the policy looks for and how they should correct it! I introduced a technique in Chapter 2 to verify the quality of your naming and code. Try asking someone else to read the test. If they can understand its purpose, they can update their configuration to conform to the policy as code.

MODULARIZE TESTS

You can apply some of the *module patterns* from Chapter 3 to policy as code. For example, the uDress payments team asks to borrow your security and policy tests for their infrastructure. You divide your database policies into "database-tests" and firewall policies into "firewall-tests".

The security team also asks you to add a Center for Information Security (CIS) benchmark. This industry benchmark includes tests to verify the best practices for secure configuration on GCP. After adding the security benchmark, you realize that you have too many tests to track in multiple repositories.

Figure 8.7 moves all of these tests into a repository named "gcp-security-test". The repository organizes all tests for uDress's GCP infrastructure. The uDress frontend and payments teams can reference a shared repository, import the tests, and run them against their configuration. Meanwhile, the security team can update the security benchmarks in one place in the "gcp-security-tests" repository.

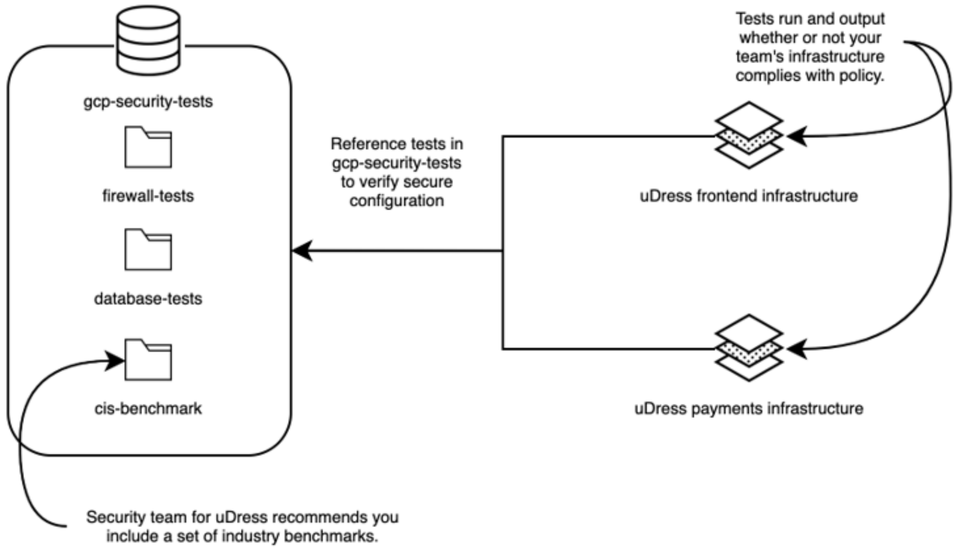

gcp-security-tests
firewall-tests
database-tests
cis-benchmark
Reference tests in gcp-security-tests to verify secure configuration
Tests run and output whether or not your team's infrastructure complies with policy.
uDress frontend infrastructure
uDress payments infrastructure
Security team for uDress recommends you include a set of industry benchmarks.

Figure 8.7. Add policy as code to a shared repository for distribution across all teams creating infrastructure.

Like your infrastructure approach as code repository structures, you can choose to put your organization's policy as code in a single repository or divide them across multiple repositories based on the environment. In either structure, make sure all teams have visibility into security and policy tests for the organization to learn how to deploy compliant infrastructure.

Furthermore, divide the tests based on business unit, function, environment, infrastructure provider, infrastructure resource, stack, or benchmark. You want to evolve the types of tests individually as your business changes. Some business units may need one type of test while another may not. Dividing the tests and running them selectively helps

ADD POLICY AS CODE TO DELIVERY PIPELINES

Your team wants to make sure to run the security and policy tests before pushing to production, so they add them as a *stage of their delivery pipeline*. Policy as code runs *after* deploying the changes to a testing environment but *before* releasing to production. You get fast feedback on infrastructure changes, prioritizing functionality but checking policy before production.

The security team also adds policy as code to scan the *running* production environment. This dynamic analysis continuously verifies the security and compliance of any emergency or break-glass changes to infrastructure.

Figure 8.9 shows the workflow of static analysis in a delivery pipeline and dynamic analysis of running infrastructure to check configuration changes and reactively address issues in resources proactively. After deploying the changes to a testing environment, you run the security and policy tests. They should pass before releasing the changes to production. When the resources get the changes, you scan the running infrastructure with similar tests for runtime security and policy checks.

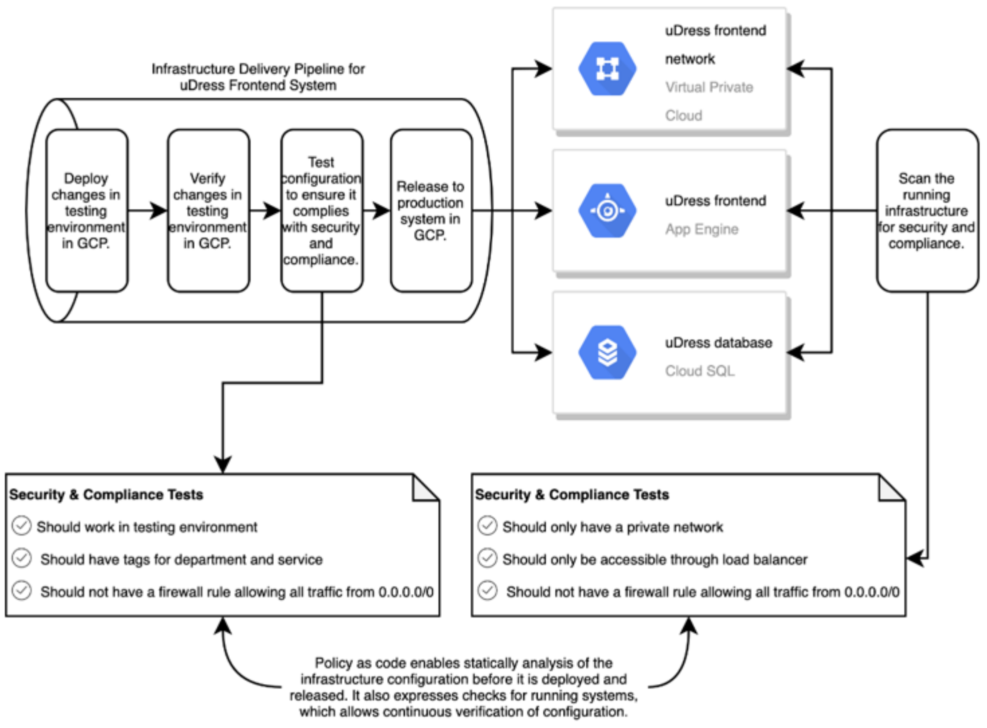

Infrastructure Delivery Pipeline for uDress Frontend System
Deploy changes in testing environment in GCP.
Verify changes in testing environment in GCP.
Test configuration to ensure it complies with security and compliance.
Release to production system in GCP.
uDress frontend network Virtual Private Cloud
uDress frontend App Engine
uDress database Cloud SQL
Scan the running infrastructure for security and compliance.
Security & Compliance Tests
Should work in testing environment
Should have tags for department and service
Should not have a firewall rule allowing all traffic from 0.0.0.0/0
Security & Compliance Tests
Should only have a private network
Should only be accessible through load balancer
Should not have a firewall rule allowing all traffic from 0.0.0.0/0
Policy as code enables statically analysis of the infrastructure configuration before it is deployed and released. It also expresses checks for running systems, which allows continuous verification of configuration.

Figure 8.8. Tests for security and policy check for improper infrastructure configuration and prevent the changes from going to production.

You may have different tests for static and dynamic analysis. Some tests, such as live verification of endpoint access, can only work on running infrastructure. As a result, you want to run some tests *before* and *after* you push to production.

If your static analysis tests take too long, you could run a subset of tests after you push the change to production. However, you will have to remediate any security or compliance violations quickly. As a result, I recommend running the most critical security and policy tests as part of your pipeline.

Image building

You might encounter the practice of building immutable server or container images. By baking the packages you want into a server or container image, you can create new servers with updates without the problems of in-place updates

Use the same workflow of the infrastructure pipeline with policy as code to build the immutable images. The workflow includes unit tests to check the scripts for specific installation requirements, such as a company package registry, and integration tests against a test server to verify that the package versions comply with policy and security.

You can always use dynamic analysis in the form of an agent to scan a server and make sure its configuration complies with rules. For example, Sysdig offers Falco, a runtime security tool that runs on a server and checks for rule compliance.

Most teams *do not* want their security or policy tests to block all changes from going to production. For example, what if customers need to access public endpoints for infrastructure? The test to check for a private network only may not apply. Sometimes, you find exceptions in your security policy.

DEFINE ENFORCEMENT LEVELS

As you build more policy as code, you must identify the most important ones and make exceptions for others. For example, the uDress security team identifies the database tagging policy to be hard mandatory. The delivery pipeline *must fail* if it does not find tags, and someone must add the tags.

In figure 8.9, you define three categories of policy. The security team mandates that you fix the database tags before pushing to production. However, the team makes an exception for your firewall rule because customers need access to your endpoint. The team also includes some advice on more secure infrastructure configurations.

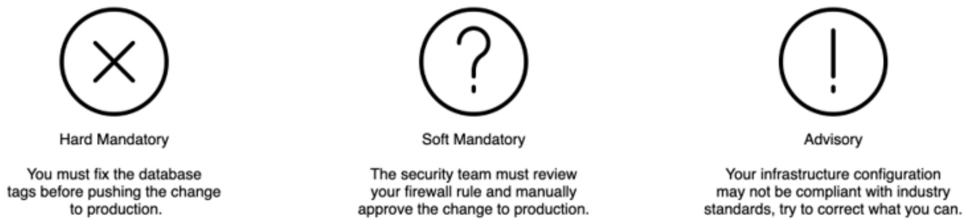

Hard Mandatory
You must fix the database tags before pushing the change to production.
Soft Mandatory
The security team must review your firewall rule and manually approve the change to production.
Advisory
Your infrastructure configuration may not be compliant with industry standards, try to correct what you can.

Figure 8.9. You can divide policy as code into three enforcement categories that gate changes before production.

I classify policy as code into three categories of enforcement (borrowed from HashiCorp Sentinel's terminology):

- *Hard mandatory* for required policies
- *Soft mandatory* for policies that may require manual analysis for an exception
- *Advisory* for knowledge-sharing of best practices.

The security team classifies the firewall rule as a soft mandatory. Some public load balancers must allow access from 0.0.0.0/0. If the firewall rule test fails, someone from the security team must review the rule and manually approve the change to production in the pipeline.

The security team sets the CIS benchmarks as "advisory" for knowledge sharing and best practices. If possible, they ask you to correct the configuration, but they do not require enforcement before production.

Do you have to run security tests before changes go to production? They take a while to run, after all! If you worry that security and policy tests will gate the changes too long, run the *hard mandatory* tests before deploying to production.

You can run the soft mandatory or advisory tests asynchronously, so only the necessary tests block your pipeline. I *do not* recommend running *all* security and policy tests asynchronously because you may temporarily introduce a non-compliant configuration to production, even if you fix it quickly after running asynchronous tests!

Figure 8.10 summarizes testing patterns and practices, such as writing detailed test names and error messages. Similar to infrastructure, you can modularize tests based on function and add tests to delivery pipelines for production.

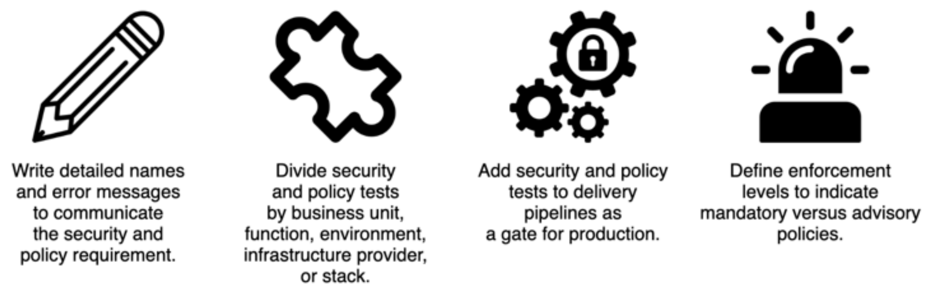

Write detailed names and error messages to communicate the security and policy requirement.
Divide security and policy tests by business unit, function, environment, infrastructure provider, or stack.
Add security and policy tests to delivery pipelines as a gate for production.
Define enforcement levels to indicate mandatory versus advisory policies.

Figure 8.10. Tests for security and policy check for improper infrastructure configuration and prevent the changes from going to production.

No matter the tool, security benchmark, or policy rules, you should express and communicate the practices in the form of tests. Following these patterns and practices will help you and your teammates improve your security and compliance knowledge.

As your organization grows and sets more policies, you expand your security and policy tests with it. Early adoption of policy as code sets a foundation for baking security and compliance practices into your infrastructure as code. If you cannot find a tool to run the tests you need, consider writing your own tests to parse infrastructure as code.

8.4 Summary

- A company policy ensures that systems comply with security, audit, and organizational requirements. Your company defines policies based on industry, country, and other factors.
- The principle of least privilege only gives a user or service account the minimum access they require.
- Ensure your infrastructure as code uses credentials with least privilege access to the infrastructure provider. Least privilege prevents someone from exploiting the credentials and creating unauthorized resources in your environment.
- Use a tool to suppress or redact plaintext, sensitive information in a delivery pipeline.
- Rotate any usernames or passwords generated by infrastructure as code after applying it in the pipeline.
- Tagging infrastructure with service, owner, email, accounting information, environment, and automation details makes it easier to identify and audit security and billing.
- Policy as code tests some infrastructure metadata and verifies if it complies with a secure or compliant configuration.
- Use policy as code to test the security and compliance of your infrastructure before pushing to production but after functional testing of your system.
- Apply clean infrastructure as code and module patterns to managing and scaling policy as code.
- Classify each security and policy test into one of three enforcement categories, such as hard mandatory (must fix), soft mandatory (manual review), and advisory (best practice but not blocking production).

9 Making changes

This chapter covers

- **Determining when to use patterns like blue-green deployments to update infrastructure**
- **Establishing immutability to create environments across multiple regions with infrastructure as code**
- **Determining change strategies for updating stateful infrastructure**

In previous chapters, we started with helpful practices and patterns for modularizing, decoupling, testing, and deploying infrastructure changes. However, you also need to manage changes to infrastructure with infrastructure as code techniques. In this chapter, you'll learn how to change infrastructure as code with strategies that apply immutability to minimize the impact of potential failure.

Let's return to Datacenter for Veggies and its struggles to modularize infrastructure as code from Chapter 5. The company acquires Datacenter for Carnivorous Plants as a subsidiary. Its carnivorous plants, like Venus Flytraps, require particular growing conditions.

As a result, Datacenter for Carnivorous Plants needs global networking and network-optimized servers and components. Most teams configure their infrastructure as code but realize that their code can't handle such a widespread change. They ask you, a Datacenter for Veggies engineer with some experience in infrastructure as code, to help them.

The engineering team directs you to the Cape sundew team to update their infrastructure first. As a hardy carnivorous plant, the Cape sundew can best handle any system downtime that causes fluctuations in temperature and watering. All of the infrastructure resources for the Cape sundew infrastructure exist in a single repository with a few configuration files.

You investigate and diagram the architecture of the sundew system. Figure 9.1 shows a regional forwarding rule (load balancer) that sends traffic to a regional network with a container cluster and three servers. All traffic circulates in the same region and not globally.

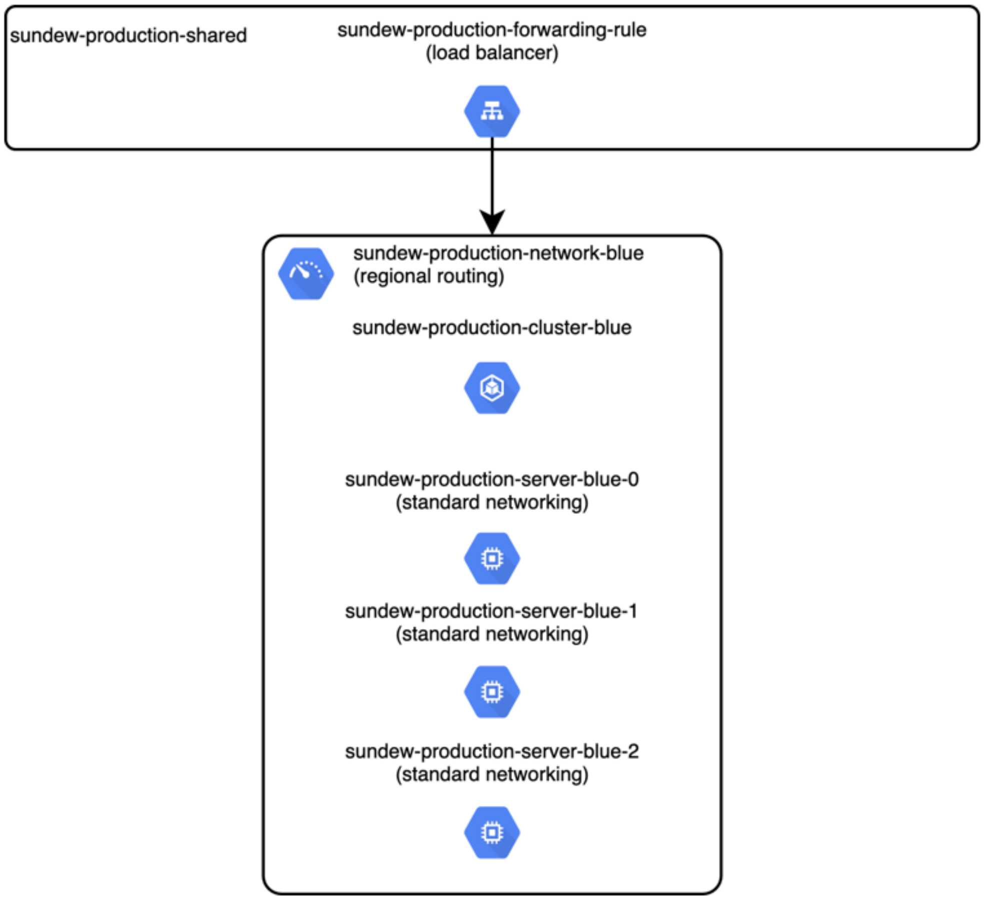

sundew-production-shared
sundew-production-forwarding-rule (load balancer)
sundew-production-network-blue (regional routing)
sundew-production-cluster-blue
sundew-production-server-blue-0 (standard networking)
sundew-production-server-blue-1 (standard networking)
sundew-production-server-blue-2 (standard networking)

Figure 9.1. The Cape sundew applications use an infrastructure system with shared load balancing resources, three servers, and one container cluster.

You want to change all resources to send traffic globally instead of routing in a single region. However, the sundew team defined all of these resources in the same infrastructure as code, also known as the singleton pattern (refer to chapter 3). The resources also share infrastructure state.

How do you roll out the network changes to the system and minimize their impact? You worry about disrupting the watering application if you treat the infrastructure mutably by making the change in-place. For example, changing your network to use global routing may affect the servers supporting the watering application.

You remember from chapter 2 that you can use the idea of immutability to build new infrastructure with new changes. If you can apply these techniques to the system, you can isolate and test the changes in a new environment without affecting the old one. In this chapter, you'll learn how to isolate and make changes to infrastructure as code.

Code examples

Demonstrating change strategies requires a sufficiently large (and complex) example. If you run the complete example, you will incur a cost that exceeds the free tier in Google Cloud Platform (GCP). In this book, I only include the relevant lines of code in the text and omit the rest for readability. For the listing in its entirety, refer to the book's code repository at https://github.com/joatmon08/manning-book/tree/main/ch09.

9.1 Pre-change practices

You jump right into changing the sundew system. Unfortunately, you accidentally delete a configuration attribute that tags a server with "blue," which allows traffic between all blue instances in the network. You push your change to your delivery pipeline to test the configuration and apply it to production.

Testing misses the deleted tag. Fortunately, your monitoring system sends you an alert that the watering application cannot communicate with your new server! You divert all requests to a duplicate server instance to ensure the sundews still get watered while you debug.

You realize that you should *not* start changing the system. The sundew system has existing architecture and tools you need to understand before you start. You also need to know if the system has backups or alternative environments ready for use if you break something. What should you do *before* you make a change?

9.1.1 Checklist

You always run the risk of introducing bugs and other problems when changing infrastructure as code. You need testing, monitoring, and observability (the ability to infer a system's internal state from its outputs) to ensure you haven't affected the system during your change. If you do not have some visibility into your system, you cannot quickly troubleshoot problems from broken changes.

Before you change the sundew system, you decide to review a few things about your system. Figure 9.2 shows what you review. You first add a test to check for your deleted tag. Next, you examine your monitoring system for both system and application health checks and metrics. Finally, you create duplicate servers as backup, just in case you break the existing servers. You can send traffic to the backup server if you accidentally affect the updated server.

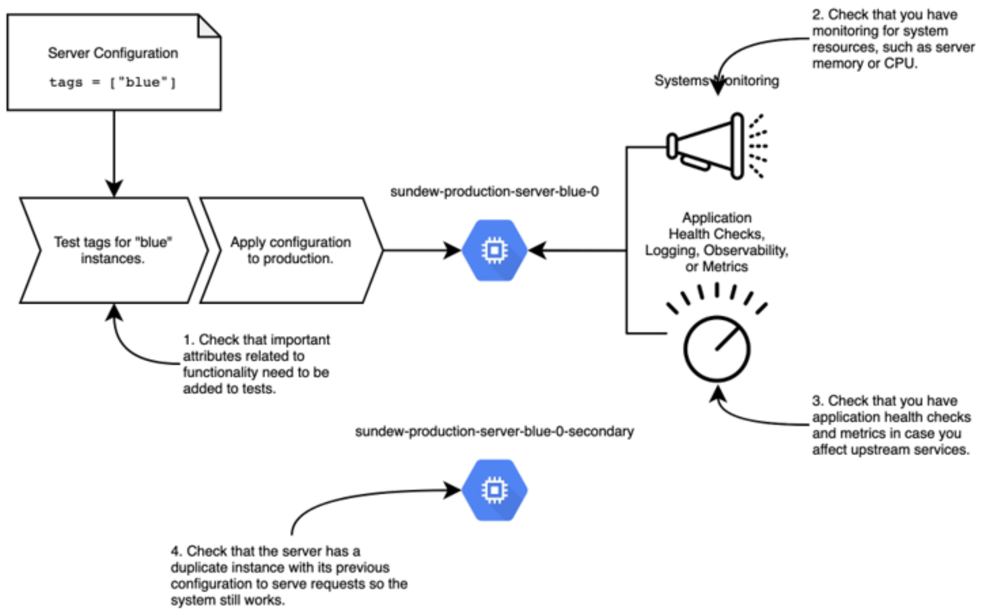

Server Configuration
tags = ["blue"]
Test tags for "blue" instances.
Apply configuration to production.
sundew-production-server-blue-0
Systems Monitoring
2. Check that you have monitoring for system resources, such as server memory or CPU.
Application Health Checks, Logging, Observability, or Metrics
1. Check that important attributes related to functionality need to be added to tests.
3. Check that you have application health checks and metrics in case you affect upstream services.
sundew-production-server-blue-0-secondary
4. Check that the server has a duplicate instance with its previous configuration to serve requests so the system still works.

Figure 9.2. Before updating the network, you need to verify test coverage, systems and application monitoring, and redundancy.

Why did you create the duplicate servers as backup? It helps to have extra resources that duplicate the previous configuration that you only use if the primary resources fail. This ***redundancy*** keeps your system running.

DEFINITION Redundancy is the duplication of resources to improve the system's performance. If updated components fail, the system can use working resources with a previous configuration.

In general, review the following checklist before you make a change:

- Can you preview each change and *test* them in an isolated environment?
- Does the system have *monitoring and alerts* configured to identify any anomalies?
- Does the application track error responses with *health checks, logging, observability, or metrics*?
- Do the applications and their systems have any *redundancy*?

The items on the list focus on visibility and awareness. Without monitoring systems or tests to help identify problems, you may have trouble identifying or resolving broken changes. Once I pushed a change that broke an application and did not find out until *two weeks* after release. It took so long to realize the problem because we did not have alerts on the application!

A pre-change checklist sets the foundation for debugging any problems and establishing any backup plans should the change fail. You can even use the practices from Chapters 6 and 8 to build this checklist into delivery pipelines as quality gates.

9.1.2 Adding reliability

After reviewing the pre-change checklist, you realize that you need a better backup environment in your system. To ensure you don't bring down the entire sundew system in your continued refactoring efforts, you need additional redundancy. When you finally deploy a change to the sundew team's modules, you do not need to worry about disrupting the system.

Unfortunately, the sundew system only exists in `us-central1`. The sundews don't get watered if the region fails! You decide to build an idle production sundew system in another region (`us-west1`) so you can restart the watering application. You use infrastructure as code to *reproduce* the active region in `us-central1` to the passive (idle) region in `us-west1`.

You can now use the environment in the passive region as a backup. In figure 9.3, you update the sundew team's configuration to use a server module and push the change to the active environment. If it does not work, you temporarily send all traffic to the passive environment while you debug the problem. Otherwise, you run tests and update the passive environment with the changes.

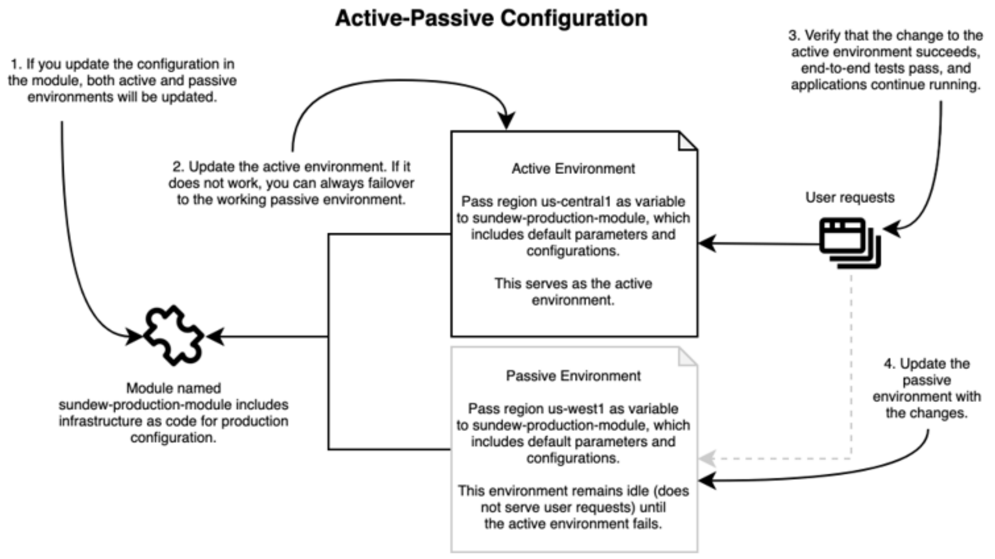

Active-Passive Configuration
1. If you update the configuration in the module, both active and passive environments will be updated.
2. Update the active environment. If it does not work, you can always failover to the working passive environment.
3. Verify that the change to the active environment succeeds, end-to-end tests pass, and applications continue running.
Active Environment
Pass region us-central1 as variable to sundew-production-module, which includes default parameters and configurations.
This serves as the active environment.
User requests
Module named sundew-production-module includes infrastructure as code for production configuration.
Passive Environment
Pass region us-west1 as variable to sundew-production-module, which includes default parameters and configurations.
This environment remains idle (does not serve user requests) until the active environment fails.
4. Update the passive environment with the changes.

Figure 9.3. You use infrastructure as code to implement an active-passive configuration for the sundew system to improve its reliability during changes.

The sundew system now uses an ***active-passive configuration***, where one environment sits idle and serves as a backup.

> **DEFINITION** Active-passive configuration is a model in which one system is the active environment for completing user requests, and the other is the backup environment.

If the environment in `us-central1` stops working, you can always send traffic to the other passive environment in `us-west1`. Switching from a failed active environment to the working passive one follows the process of ***failover***.

> **DEFINITION** Failover is the practice of using passive (or standby) resources to take over when the primary resources fail.

Why would you want an active-passive configuration? Building a passive environment in a second region improves the overall reliability of the system. ***Reliability*** measures how long a system performs correctly within a time period.

> **DEFINITION** Reliability measures how long a system performs correctly within a time period.

You want to keep your system reliable as you make infrastructure as code changes. Improving reliability minimizes disruption to business-critical applications and ultimately, end users. You reduce your blast radius to the active environment with the option to cut over traffic to a working passive environment.

Let's create the active-passive configuration in code. In your terminal, you copy the file named `blue.py` containing the sundew's infrastructure resources to a new file named `passive.py`.

```
$ cp blue.py passive.py
```

In `passive.py`, you update a few variables to create the passive sundew environment, including region and name.

Listing 9.1 Update passive sundew environment for us-west1

```
TEAM = 'sundew'
ENVIRONMENT = 'production'
VERSION = ‘passive’    #A
REGION = 'us-west1'    #B
IP_RANGE = '10.0.1.0/24'    #C

zone = f'{REGION}-a'
network_name = f'{TEAM}-{ENVIRONMENT}-network-{VERSION}'    #D
server_name = f'{TEAM}-{ENVIRONMENT}-server-{VERSION}'    #D

cluster_name = f'{TEAM}-{ENVIRONMENT}-cluster-{VERSION}'    #D
cluster_nodes = f'{TEAM}-{ENVIRONMENT}-cluster-nodes-{VERSION}'    #D
cluster_service_account = f'{TEAM}-{ENVIRONMENT}-sa-{VERSION}'    #D

labels = {    #E
   'team': TEAM,    #E
   'environment': ENVIRONMENT,    #E
   'automated': True    #E
}    #E
```

#A Set the version to identify the passive environment.
#B Change the region from us-central1 to us-west1 for the passive environment.
#C Update a different IP address range for the passive environment to avoid sending requests.
#D Remaining variables and functions reference the constants for the passive environment, including region.
#E Define labels for resources so that you can identify the passive versus active environment.

You now have a backup environment in case your module change goes wrong. Imagine you push the change and break the active environment in `us-central1`. You can update and push the configuration for a production global load balancer to failover and send everything to the passive environment.

In listing 9.2, you change the weight on your global load balancer to send 100% of traffic to the passive environment.

Listing 9.2 Failover to passive sundew environment in us-west1

```
import blue    #A
import passive    #A

services_list = [
   {
       'version': 'blue',    #B
       'zone': blue.zone,
       'name': f'{shared_name}-blue',
       'weight': 0    #C
   }, {
       'version': 'passive',    #B
       'zone': passive.zone,
       'name': f'{shared_name}-passive',
       'weight': 100    #D
   }
]


def _generate_backend_services(services):    #E
   backend_services_list = []
   for service in services:
       version = service['version']
       weight = service['weight']
       backend_services_list.append({
           'backend_service': (     #H
               '${google_compute_backend_service.'     #H
               f'{version}.id}}'     #H
           ),     #H
           'weight': weight,     #H
       })
   return backend_services_list


def load_balancer(name, default_version, services):
   return [{
       'google_compute_url_map': {    #F
           TEAM: [{
               'name': name,
               'path_matcher': [{
                   'name': 'allpaths',
                   'path_rule': [{     #G
                       'paths': [     #G
                           '/*'     #G
                       ],     #G
                       'route_action': {    #E
                           'weighted_backend_services':    #E
                               _generate_backend_services(    #E
                                   services)    #E
                       }    #E
                   }]
               }]
           }]
       }
   }]
```

#A Import infrastructure as code for both blue (active environment) and passive environment.
#B Define a list of versions for each environment to attach to the load balancer, blue and passive.

#C Configure load balancer with a weight to send 0% of traffic to the blue version.
#D Configure load balancer with a weight to send 100% of traffic to the passive version.
#E Add the two versions to the load balancer with their weights as routes to the load balancing rule.
#F Create the Google compute URL map (load balancing rule) using a Terraform resource based on the path, blue (active) and passive servers, and weight.
#G Set up a path rule that directs all paths to the active or passive servers.
#H Define backend services for the blue and passive environments with a weight to direct traffic to each one.

After you failover the system to the passive environment, the sundew team reports the return of the watering application. You have a chance to debug problems with your module in the blue (active) environment. The active-passive configuration will protect from failures in individual regions in the future.

The sundew team tells you that eventually, they want to send traffic to both regions. Both environments in each region process requests. Figure 9.4 shows their dream configuration. The next time, they want to update the module and push it out to one region. If a region breaks, *most* requests will still get processed by the system. You water the sundews less frequently but you have an opportunity to fix the broken region.

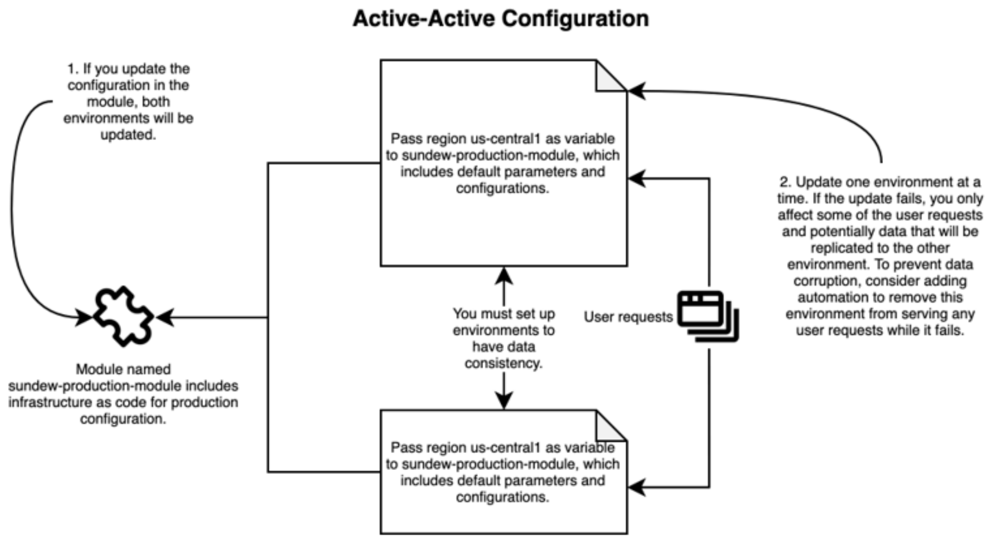

Active-Active Configuration
1. If you update the configuration in the module, both environments will be updated.
Pass region us-central1 as variable to sundew-production-module, which includes default parameters and configurations.
2. Update one environment at a time. If the update fails, you only affect some of the user requests and potentially data that will be replicated to the other environment. To prevent data corruption, consider adding automation to remove this environment from serving any user requests while it fails.
You must set up environments to have data consistency.
User requests
Module named sundew-production-module includes infrastructure as code for production configuration.
Pass region us-central1 as variable to sundew-production-module, which includes default parameters and configurations.

Figure 9.4. In the future, the sundew team will further refactor the system configuration to support an active-active configuration and send requests to both regions.

Why aspire to run two active environments? Many distributed systems run in an ***active-active configuration***, which means both systems process and accept requests and replicate data between them. Public cloud architectures recommend using a multi-region, active-active configuration to improve the system's reliability.

DEFINITION Active-active configuration is a model in which multiple systems are the active environments for completing user requests and replicating data between them.

Your changes to infrastructure as code will differ depending on active-passive or active-active configuration. The sundew team's active-active configuration must refactor the infrastructure as code into more modular components and support data replication between environments. Assuming the sundew team refactors their applications to support an active-active configuration, you will need to implement some form of global load balancing in your infrastructure as code and connect it to each region.

Infrastructure as code conforms to *reproducibility*, which we covered in chapter 1. Thanks to this principle, we can create a new environment in a new region with a few updates to attributes. You do not have to painstakingly rebuild an environment resource-by-resource as we did in chapter 2.

However, you may find yourself copying and pasting a lot of the same configuration. Try to pass regions as inputs and modularize your infrastructure as code to reduce copy-and-paste. Separate shared resources configuration, like global load balancer definitions, away from the environment configurations.

Multi-region environments always incur a cost in terms of money and time but can help improve system reliability. Infrastructure as code expedites creating copies of new environments in other regions and enforces consistent configurations across regions. Inconsistencies between regions can introduce significant system failures and increase maintenance effort! Chapter 12 discusses cost management and its considerations.

9.2 Blue-green deployment

Now that you have another environment to fall back on if you accidentally corrupt the active one, you can start updating the sundew system to use global networking and premium tier network access for servers. The sundew system uses an active-passive configuration, which means duplicating a whole new environment in a new region just to make changes.

You realize that some changes don't require duplicating entire environments across multiple regions. Why have an entire passive environment just to update a server? Can't you just update one server? After all, we want to minimize the blast radius of failures and optimize our resource efficiency. Instead of using active-passive configuration, you can apply the pattern to fewer resources on a much smaller scale.

In figure 9.5, you reproduce a new *network* with global networking instead of the entire environment. You label the new network "green" and deploy a set of three servers and one cluster on it. After testing the new resources, you use your global load balancer to send a small percentage of traffic to the new resources. The requests succeed, indicating the update to global routing worked.

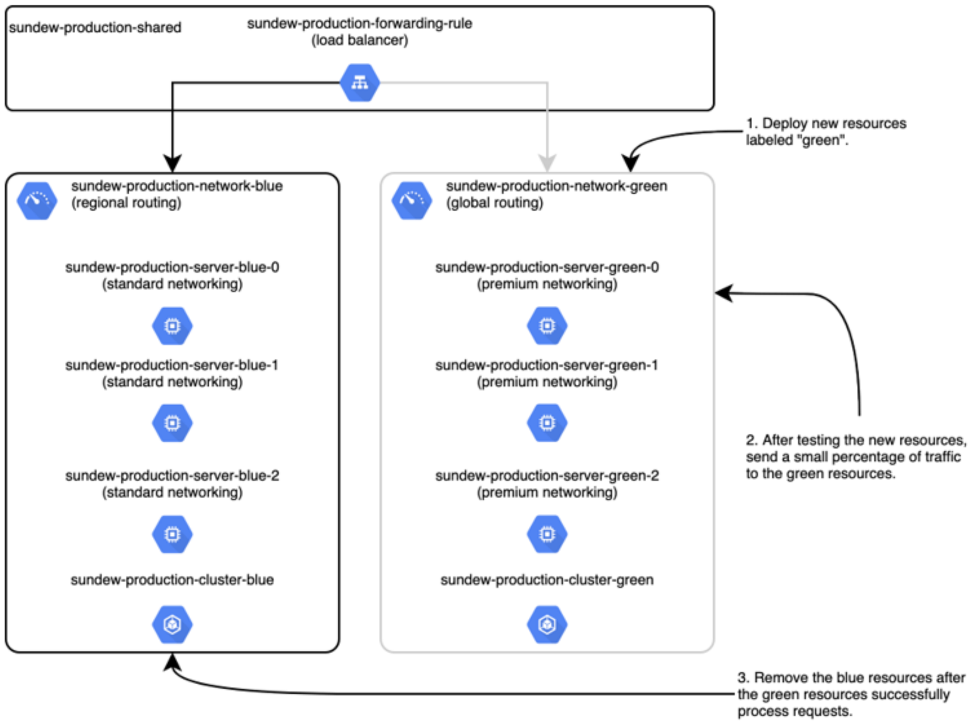

sundew-production-shared
sundew-production-forwarding-rule (load balancer)
1. Deploy new resources labeled "green".
sundew-production-network-blue (regional routing)
sundew-production-server-blue-0 (standard networking)
sundew-production-server-blue-1 (standard networking)
sundew-production-server-blue-2 (standard networking)
sundew-production-cluster-blue
sundew-production-network-green (global routing)
sundew-production-server-green-0 (premium networking)
sundew-production-server-green-1 (premium networking)
sundew-production-server-green-2 (premium networking)
sundew-production-cluster-green
2. After testing the new resources, send a small percentage of traffic to the green resources.
3. Remove the blue resources after the green resources successfully process requests.

Figure 9.5. Use blue-green deployment to create a green environment for the global network, send some traffic to the new resources, and remove the old resources.

The pattern of creating a new set of resources and gradually cutting over to them applies the principles of composability and evolvability of your system. You add a new *green* set of resources to your environment and allow them to evolve independently of the old resources. If you want to make changes to infrastructure, repeat this workflow to reduce your blast radius of changes and test new resources before sending traffic to them.

This pattern, called a ***blue-green deployment***, creates a new subset of infrastructure resources that stages the changes you want to make, labeled "green". You then direct a few requests to the green staging infrastructure resources and ensure everything works. Over time, you send all requests to the green infrastructure resources and remove the old production resources, labeled "blue".

> **DEFINITION** Blue-green deployment is a pattern that creates a new subset of infrastructure resources with the changes you want to make. You gradually shift traffic from the old set of resources (blue) to the new set of resources (green) and eventually remove the old ones.

Blue-green deployment allows you to isolate and test changes in a (temporarily) new staging environment before sending requests to it. After validating the green environment, you can switch the new environment to the production one and delete the old one. You temporarily pay for two environments for a few weeks but minimizes the overall cost of maintaining persistent environments.

Blue/green or red/black?

Blue-green deployment has a few different labels, occasionally with subtle nuances depending on the context. It doesn't matter what color or names you label the environments, as long as you can identify which environment serves as the existing production one or the new staging one. I have also used production/staging and version numbering (v**1**/v**2**) to identify old and new resources during blue-green deployment.

You should use a blue-green deployment pattern to refactor or update your infrastructure as code beyond a few minimal configurations. Blue-green deployment depends on the principle of immutability to create new resources, cut over traffic or functionality to them, and remove old resources. Most of the patterns for refactoring (Chapter 10) and changing infrastructure as code often involve the principle of reproducibility.

Image building and configuration management

You can similarly mitigate the risk of a failed machine image or configuration by applying a blue-green deployment pattern. Isolate machine image or configuration management updates to a new server (green), send traffic to it, and test its functionality before removing the old server (blue).

You already have a global load balancer for the existing "blue" network that you can use later to connect the new "green" network. In the following sections, let's implement each step of a blue-green deployment for the sundew system.

9.2.1 Deploy green infrastructure

To start a blue-green deployment for the sundew system's global networking and premium tier servers, you copy the configuration for the existing blue network. You create the file named `green.py` and paste the blue network configuration. In listing 9.3, you make changes to the network definition so it uses a global routing mode.

Listing 9.3 Create the green network

```
TEAM = 'sundew'
ENVIRONMENT = 'production'
VERSION = 'green'    #A
REGION = 'us-central1'
IP_RANGE = '10.0.0.0/24'    #B

zone = f'{REGION}-a'
network_name = f'{TEAM}-{ENVIRONMENT}-network-{VERSION}'    #C

labels = {
   'team': TEAM,
   'environment': ENVIRONMENT,
   'automated': True
}


def build():    #C
   return network()    #C


def network(name=network_name,    #C
           region=REGION,
           ip_range=IP_RANGE):
   return [
       {
           'google_compute_network': {    #E
               VERSION: [{
                   'name': name,
                   'auto_create_subnetworks': False,
                   'routing_mode': 'GLOBAL'    #D
               }]
           }
       },
       {
           'google_compute_subnetwork': {    #F
               VERSION: [{
                   'name': f'{name}-subnet',
                   'region': region,
                   'network': f'${{google_compute_network.{VERSION}.name}}',
                   'ip_cidr_range': ip_range
               }]
           }
       }
   ]
```

#A Set the name of the new network version “green”.
#B Keep the IP address range for green the same as the blue network. Google Cloud Platform (GCP) allows the two networks to have the same CIDR block if you have not set up peering.
#C Use the module to create the JSON configuration for the network and subnetwork for the “green” network.
#D Update the green network’s routing mode to global to expose routes globally.
#E Create the Google network using a Terraform resource based on the name and a global routing mode.
#F Create the Google subnetwork using a Terraform resource based on the name, region, network, and IP address range.

You want to keep the same configuration for blue and green resources when possible. They should only differ in the changes you want to make to the green sources. However, you might have some differences!

For example, if I had some specific peering configuration for my networks, I *could not* use the blue network’s IP address range for the green network. Instead, I would need a different IP address range, like 10.0.1.0/24, and update any dependencies to communicate to another IP address range.

Blue-green deployment favors immutability, creating new, updated resources and isolating the changes away from the old resources. However, deploying a new version of a low-level resource like networking does not mean you can immediately send live traffic to it! You always start by changing and testing the infrastructure resource you want to update. Then, you must change and test *other* resources that depend on it.

9.2.2 Deploy high-level dependencies to green infrastructure

When you use the blue-green deployment pattern, you always need to deploy a new infrastructure resource with the changes *and* a new set of high-level resources that depend on it. You finished updating the network but cannot use it unless you have servers and applications on it. The new network needs high-level infrastructure that depends on it.

You communicate to the sundew teams to deploy new clusters and servers onto the green network in figure 9.6. The servers must use premium networking on the global network. The sundew team also deploys their applications onto the cluster and servers.

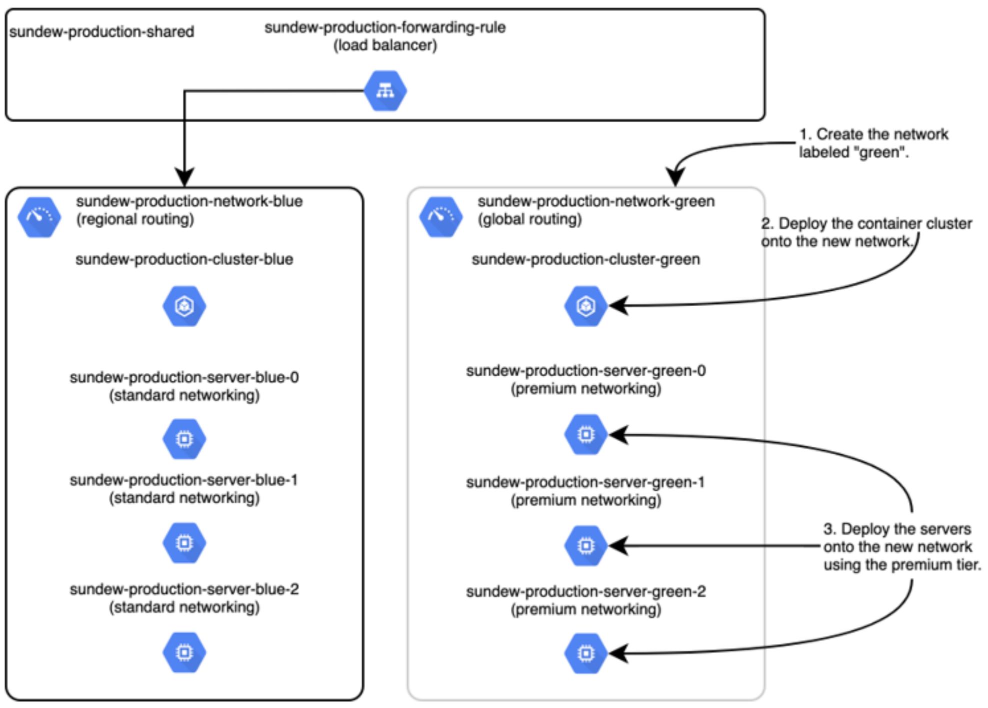

sundew-production-shared
sundew-production-forwarding-rule (load balancer)
1. Create the network labeled "green".
sundew-production-network-blue (regional routing)
sundew-production-cluster-blue
sundew-production-server-blue-0 (standard networking)
sundew-production-server-blue-1 (standard networking)
sundew-production-server-blue-2 (standard networking)
sundew-production-network-green (global routing)
sundew-production-cluster-green
2. Deploy the container cluster onto the new network.
sundew-production-server-green-0 (premium networking)
sundew-production-server-green-1 (premium networking)
3. Deploy the servers onto the new network using the premium tier.
sundew-production-server-green-2 (premium networking)

Figure 9.6. After creating the network, a low-level infrastructure resource, you also need to create new high-level resources like the server and container cluster that depend on it.

In this example, changing the low-level infrastructure resource like the network affects the high-level resources. The servers must run with premium networking. Updating the original "blue" network in-place from regional to global routing would likely have affected the servers and cluster. With a blue-green deployment, you *evolve* the network attributes for the servers without affecting the live environment.

Let's review a sample of the infrastructure as code the sundew team added to deploy the cluster onto the "green" network. They copied the cluster configuration from the blue resource and updated its attributes to run on the green network.

Listing 9.4 Add a new cluster to the green network .

```
VERSION = 'green'    #A

cluster_name = f'{TEAM}-{ENVIRONMENT}-cluster-{VERSION}'   #A
cluster_nodes = f'{TEAM}-{ENVIRONMENT}-cluster-nodes-{VERSION}'   #A
cluster_service_account = f'{TEAM}-{ENVIRONMENT}-sa-{VERSION}'   #A


def build():     #C
   return network() + \
       cluster()    #B

def cluster(name=cluster_name,    #D
           node_name=cluster_nodes,    #D
           service_account=cluster_service_account,    #D
           region=REGION):    #D
   return [
       {
           'google_container_cluster': {    #E
               VERSION: [
                   {
                       'initial_node_count': 1,    #E
                       'location': region,
                       'name': name,
                       'remove_default_node_pool': True,
                       'network': f'${{google_compute_network.{VERSION}.name}}',    #B
                       'subnetwork': \
                         f'${{google_compute_subnetwork.{VERSION}.name}}'    #B
                   }
               ]
           }
       }
   ]
```

#A Label the new version of the cluster “green”.
#B Build the cluster on the green network and subnetwork.
#C Use the module to create the JSON configuration for the network, subnetwork, and cluster for the “green” network.
#D Pass required attributes to the cluster, including name, node names, service accounts for automation, and region.
#E Create the Google container cluster using a Terraform resource with one node and on the green network.

The cluster does not require any changes to adapt to the global networking configuration. However, the servers need premium networking. You copy the server configuration from blue and change it to use premium networking attributes in `green.py`.

Listing 9.5 Add premium networking to servers on the green network

```
VERSION = 'green'    #A

server_name = f'{TEAM}-{ENVIRONMENT}-server-{VERSION}'    #G


def build():    #D
   return network() + \
       cluster() + \
       server0() + \    #C
       server1() + \    #C
       server2()    #C


def server0(name=f'{server_name}-0',    #B
           zone=zone):    #B
   return [
       {
           'google_compute_instance': {    #E
               f'{VERSION}_0': [{
                   'allow_stopping_for_update': True,
                   'boot_disk': [{
                       'initialize_params': [{
                           'image': 'ubuntu-1804-lts'    #E
                       }]
                   }],
                   'machine_type': 'f1-micro',    #E
                   'name': name,
                   'zone': zone,
                   'network_interface': [{
                       'subnetwork': \
                            f'${{google_compute_subnetwork.{VERSION}.name}}',    #E
                       'access_config': {
                           'network_tier': 'PREMIUM'   #F
                       }
                   }]
               }]
           }
       }
   ]
```

#A Label the new version of the network “green”.
#B Copy and paste each server configuration. This code snippet features the first server, “server0”. Other server configurations were omitted for clarity.
#C Build the three servers on the green network with the cluster.
#D Use the module to create the JSON configuration for the network, subnetwork, cluster , and server for the “green” network.
#E Create a small Google compute instance (server) using a Terraform resource on the green network.
#F Set the network tier to use premium networking. This enables compatibility with the underlying subnet, which uses global routing.
#G Create a template for the server name which includes the team, environment, and version (blue or green).

Updating the network tier to premium *should* not affect the functionality of the applications, although you don’t quite know! The green environment allows you to identify and mitigate any problems before affecting sundew growth. After the sundew team makes the updates, they push the changes and check the test results in the delivery pipeline.

The tests include unit, integration, and end-to-end testing to ensure that you can run the applications on the new container cluster and send requests to the new green servers. Fortunately, the tests pass, and you feel ready to send live traffic to the green resources.

9.2.3 Canary deployment to green infrastructure

You could immediately send all traffic to the green network, servers, and cluster. However, you don't want to bring down the sundew system! Ideally, you want to switch all traffic back to blue when you find a problem with your system. In figure 9.7, you adjust the global load balancer to send 90% of traffic to the blue network and 10% of traffic to the services on the green network.

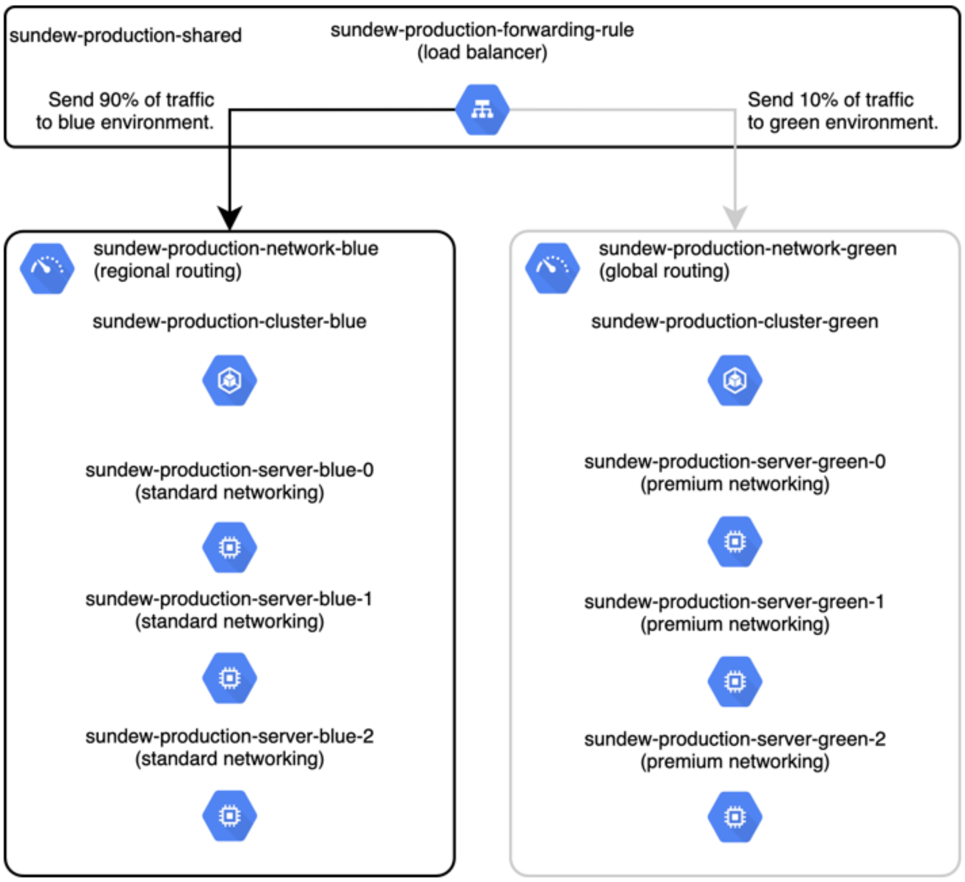

sundew-production-shared
sundew-production-forwarding-rule (load balancer)
Send 90% of traffic to blue environment.
Send 10% of traffic to green environment.
sundew-production-network-blue (regional routing)
sundew-production-cluster-blue
sundew-production-server-blue-0 (standard networking)
sundew-production-server-blue-1 (standard networking)
sundew-production-server-blue-2 (standard networking)
sundew-production-network-green (global routing)
sundew-production-cluster-green
sundew-production-server-green-0 (premium networking)
sundew-production-server-green-1 (premium networking)
sundew-production-server-green-2 (premium networking)

Figure 9.7. Configure the global load balancer to run a canary test and send a small percentage of traffic to green resources.

If this small amount of traffic sent to your system, known as a ***canary deployment***, results in errors in requests, you need to debug and fix your changes.

> **DEFINITION** Canary deployment is a pattern that sends a small percentage of traffic to the updated resources in the system. If the requests complete successfully, you increase the percentage of traffic over time.

Why send a small amount of traffic first? You do not want all of your requests failing. Sending a few requests to the updated resources helps identify critical problems before you affect your entire system.

Canary in a coal mine

A "canary" in software or infrastructure serves as a first indicator of whether or not your new system, feature, or application will work! The term comes from "canary in a coal mine." Miners would bring caged birds with them into mines. If the mine had dangerous gas, the birds would serve as the first indicator.

You'll often find references to canary *testing* in software development, which measures user experience for a new version of an application or feature. I highly recommend canary *deployment*, the technique of sending a small percentage of traffic to new resources, any time you make significant infrastructure changes.

Note that you *do not* have to use load balancers to achieve a canary deployment. You can use any method to send a few requests to updated infrastructure resources. For example, you could add one updated application instance to an existing pool of three application instances. A round-robin load balancing scheme sends about 25% of your requests to the new, updated instance and 75% to old, existing application instances.

For the sundew team, you separate the global load balancer configuration away from the green and blue environments in your configuration. This improves the load balancer's evolvability. You add the green servers as a separate backend service to the load balancer and control requests between green and blue environments.

You define the load balancer in a file named `shared.py`. Let's add the green version of the network (servers and cluster, too!) to the list of versioned environments with a weight of 10.

Listing 9.6 Add a green version to the list of load balancing services

```
import blue    #A
import green    #A

shared_name = f'{TEAM}-{ENVIRONMENT}-shared'    #B

services_list = [    #C
   {
       'version': 'blue',    #D
       'zone': blue.zone,    #D
       'name': f'{shared_name}-blue',    #D
       'weight': 90    #E
   },
   {
       'version': 'green',    #F
       'zone': green.zone,    #F
       'name': f'{shared_name}-green',    #F
       'weight': 10    #B
   }
]


def _generate_backend_services(services):    #H
   backend_services_list = []
   for service in services:    #I
       version = service['version']    #I
       weight = service['weight']    #I
       backend_services_list.append({
           'backend_service': (    #I
               '${google_compute_backend_service.'    #I
               f'{version}.id}}'    #I
           ),    #J
           'weight': weight,    #I
       })
   return backend_services_list
```

#A Import infrastructure as code for both blue and green environments.
#B Create the name for the shared global load balancer based on the team and environment.
#C Define a list of versions for each environment to attach to the load balancer, blue and green environments.
#D Add the blue network, servers, and cluster to the load balancer list. Retrieve the availability zone of the blue environment from its infrastructure as code.
#E Set the weight of traffic to the blue server instances to 90, representing 90% of requests.
#F Add the green network, servers, and cluster to the load balancer list. Retrieve the availability zone of the green environment from its infrastructure as code.
#G Set the weight of traffic to the green server instances to 10, representing 10% of requests.
#H Create a function to generate a list of backend services for a load balancer.
#I For each environment, define a Google load balancing backend service with a version and weight.

The load balancer in `shared.py` already accepts a list of backend services with differing weights. Once you deploy the list of weights and services, the load balancer configuration starts sending 10% of traffic to the green network.

Listing 9.7 Update the load balancer to send traffic to green

```
default_version = 'blue'    #A

def load_balancer(name, default_version, services):    #D
   return [{
       'google_compute_url_map': {    #C
           TEAM: [{
               'default_service': (    #A
                   '${google_compute_backend_service.'    #A
                   f'{default_version}.id}}'    #A
               ),    #A
               'description': f'URL Map for {TEAM}',
               'host_rule': [{
                   'hosts': [
                       f'{TEAM}.{COMPANY}.com'
                   ],
                   'path_matcher': 'allpaths'
               }],
               'name': name,
               'path_matcher': [{
                   'default_service': (    #A
                       '${google_compute_backend_service.'    #A
                       f'{default_version}.id}}'    #A
                   ),    #A
                   'name': 'allpaths',
                   'path_rule': [{
                       'paths': [
                           '/*'
                       ],
                       'route_action': {
                           'weighted_backend_services':    #B
                               _generate_backend_services(    #B
                                   services)    #B
                       }
                   }]
               }]
           }]
       }
   }]
```

#A Send all traffic from the load balancer to the blue environment by default.
#B Set routing rules on the load balancer to send 10% of traffic to green and 90% of traffic to blue.
#C Create the Google compute URL map (load balancing rule) using a Terraform resource based on the path, blue and green environments, and weight.
#D Use the module to create the JSON configuration for the load balancer to send 10% of traffic to green and 90% of traffic to blue.

You run Python to build the Terraform JSON configuration for review. The JSON configuration for the load balancer includes the instance groups that organize the blue servers and green servers, backend services to target the blue and green instances' groups, and the weighted routing actions.

Listing 9.8 JSON configuration for the load balancer

```
{
   "resource": [
       {
           "google_compute_url_map": {    #C
               "sundew": [
                   {
                       "default_service": \    #D
                           "${google_compute_backend_service.blue.id}",    #D
                       "description": "URL Map for sundew",
                       "host_rule": [
                           {
                               "hosts": [
                                   "sundew.dc4plants.com"
                               ],
                               "path_matcher": "allpaths"
                           }
                       ],
                       "name": "sundew-production-shared",
                       "path_matcher": [
                           {
                               "default_service":
                                 "${google_compute_backend_service.blue.id}",
                               "name": "allpaths",
                               "path_rule": [
                                   {
                                       "paths": [    #E
                                           "/*"    #E
                                       ],    #E
                                       "route_action": {
                                           "weighted_backend_services": [
                                               {
                                                   "backend_service":
                                   "${google_compute_backend_service.blue.id}",    #A
                                                   "weight": 90,    #A
                                               },
                                               {
                                                   "backend_service":
                                   "${google_compute_backend_service.green.id}",    #B
                                                   "weight": 10,    #B
                                               }
                                           ]
                                       }
                                   }
                               ]
                           }
                       ]
                   }
               ]
           }
       }
   ]
}
```

#A Send 90% of traffic to blue backend services, which use the blue network.
#B Send 10% of traffic to green backend services, which use the green network.
#C Define the Google compute URL map (load balancing rule) using a Terraform resource based on the path, blue and green environments, and weight.

#D Define the default service for the Google compute URL map (load balancing rule) to the blue environment.
#E Send all requests to the blue or green environments based on weight.

Why send all traffic by default to the blue environment? You know the blue environment processes requests successfully. If your green environment breaks, you can quickly switch the load balancer to send traffic to the default blue environment.

In general, copy, paste, and update the "green" resources. If you express the "blue" resources in a module, you only need to change the attributes passed to the module. I separate the green and blue environment definitions in separate folders or files, when possible. This makes it easier to identify the environments later.

You may notice some Python code in `shared.py` that makes it easier to evolve the list of environments and the default environment attached to the load balancer. I usually define a list of environments and a variable for the default environment. Then, I iterate over the list of environments and attach the attributes to a load balancer. This ensures that the high-level load balancer can evolve to accommodate the different resources and environments.

As you add new resources, you can adjust your load balancer to send traffic to additional environments. You may find yourself updating your load balancer's infrastructure as code each time you want to run a blue-green deployment. Taking the time and effort to configure the load balancer helps mitigate any problems from changes and controls the rollout of potentially disruptive updates.

9.2.4 Regression testing

If you immediately send all traffic to the green network and it fails, you could disrupt the watering system for the Cape sundews. As a result, you start with a canary deployment and increase the ratio of traffic to the green network by 10% each day. The process takes about two weeks, but you feel confident that you updated the network correctly! If you find a problem, you reduce traffic to the green network and debug.

Figure 9.8 shows your gradual process of increasing traffic and testing the green environment over time. You gradually decrease traffic to the blue environment until it reaches 0%. You inversely increase traffic to the green environment until 100%. You run the green environment for a week or two before disabling the blue network just in case the change broke the system in the green environment.

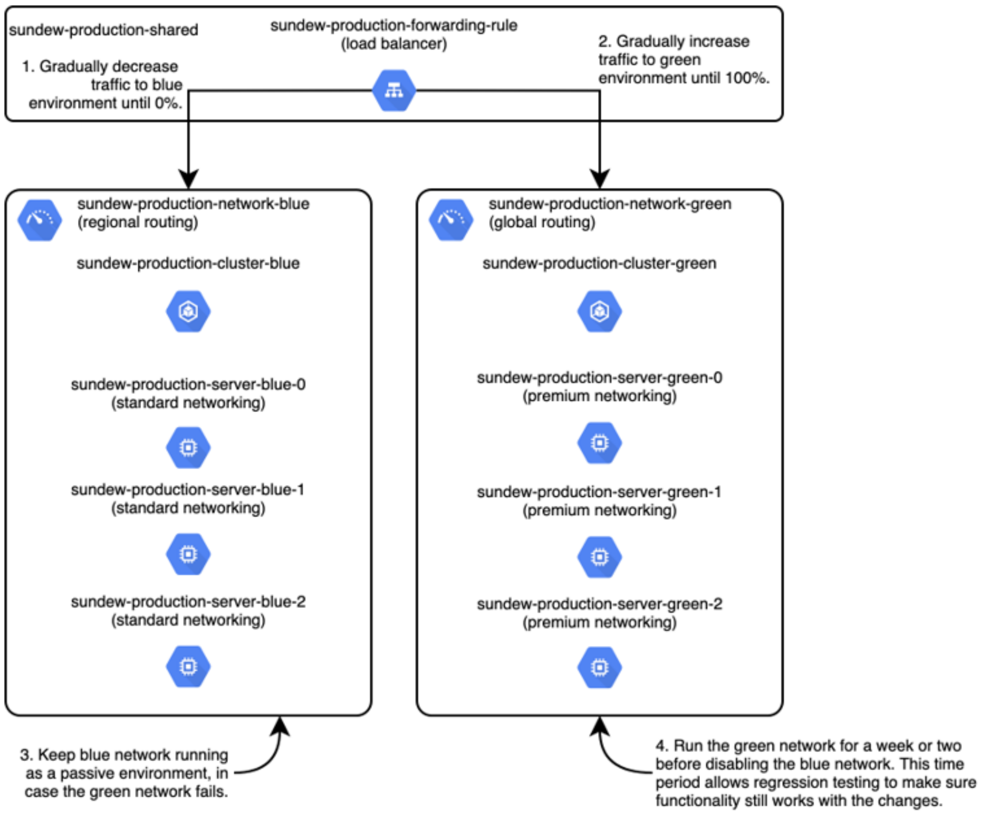

sundew-production-shared
sundew-production-forwarding-rule (load balancer)
1. Gradually decrease traffic to blue environment until 0%.
2. Gradually increase traffic to green environment until 100%.
sundew-production-network-blue (regional routing)
sundew-production-cluster-blue
sundew-production-server-blue-0 (standard networking)
sundew-production-server-blue-1 (standard networking)
sundew-production-server-blue-2 (standard networking)
sundew-production-network-green (global routing)
sundew-production-cluster-green
sundew-production-server-green-0 (premium networking)
sundew-production-server-green-1 (premium networking)
sundew-production-server-green-2 (premium networking)
3. Keep blue network running as a passive environment, in case the green network fails.
4. Run the green network for a week or two before disabling the blue network. This time period allows regression testing to make sure functionality still works with the changes.

Figure 9.8. Allow a week for a regression test before cutting all traffic to the new network removing the old one, which allows for time to verify functionality.

The process of gradually increasing traffic and waiting a week before proceeding seems so painful! However, you need to allow the system to run enough traffic through the green environment to determine if you can proceed. Some failures only appear with enough traffic through the system, while others take time to detect.

The window of time you spend testing, observing, and monitoring the system for errors becomes part of a regression test for the system. ***Regression testing*** checks if changes to the system affect existing or new functionality. Gradually increasing the traffic over time allows you to assess the system's functionality while mitigating the potential failure impact.

DEFINITION Regression testing checks if changes to the system affect existing or new functionality.

How much should you increase traffic to the green environment? Increasing traffic by 1% each day does not provide much information unless your system serves millions of requests.

“Gradual” doesn’t offer clear criteria on which increments you should use. I recommend assessing how many requests your system serves daily and the cost of failure (such as errors on user requests).

I usually start with increments of 10% and check how many requests that means for the system. If I do not get a sufficient sample size of requests to identify failures, I increase the increment. You want to insert a regression testing window between each percentage increase to identify system failures.

Even after increasing the load balancer to send all the traffic to the green network, you still want to keep running tests and monitor system functionality for a week or two. Why run regression tests for a few weeks? Sometimes, you might encounter edge cases from application requests that break functionality. By allowing a period for regression tests, you can observe whether or not the system can handle unexpected or uncommon payloads or requests.

9.2.5 Delete blue infrastructure

You observe the sundew system for two weeks and resolve any errors. You know that the blue network has not processed any requests or data for about two weeks, which means you can remove it without additional migration steps. You confirm the inactivity with a peer or change advisory board review.

Figure 9.9 updates the default service to the green environment before deleting the blue environment from infrastructure as code.

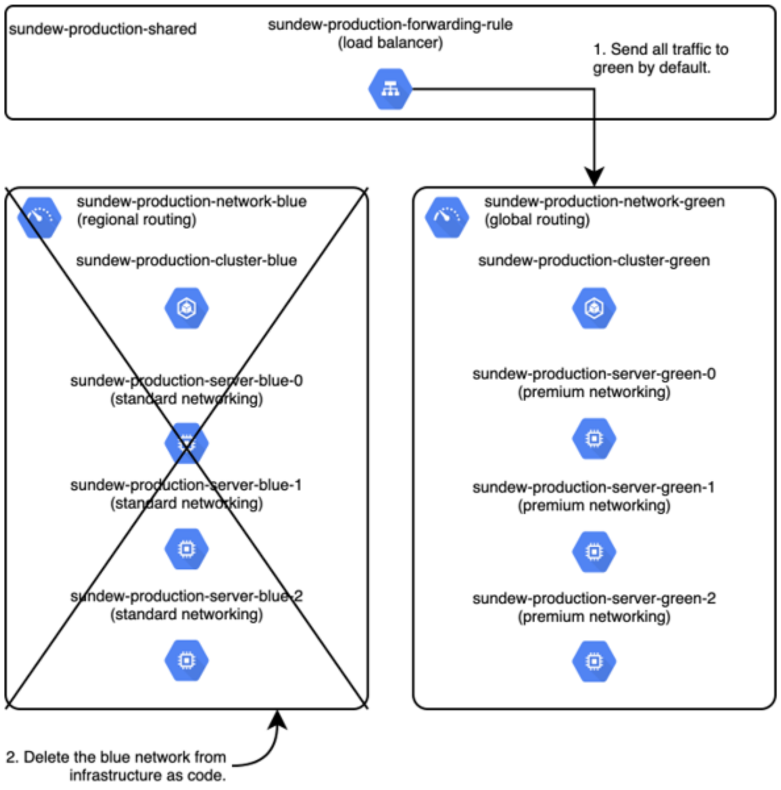

sundew-production-shared
sundew-production-forwarding-rule
(load balancer)
1. Send all traffic to
green by default.
sundew-production-network-blue
(regional routing)
sundew-production-cluster-blue
sundew-production-server-blue-0
(standard networking)
sundew-production-server-blue-1
(standard networking)
sundew-production-server-blue-2
(standard networking)
sundew-production-network-green
(global routing)
sundew-production-cluster-green
sundew-production-server-green-0
(premium networking)
sundew-production-server-green-1
(premium networking)
sundew-production-server-green-2
(premium networking)
2. Delete the blue network from
infrastructure as code.

Figure 9.9. Decommission the old network by deleting it from infrastructure as code and removing all references.

I consider deleting the blue environment with the network a major change that requires additional peer review. You may not know who uses the network. Other resources that you do not share with other teams, like servers, may not need additional peer review or change approval. Assess the potential impact of deleting an environment and categorize the change based on patterns in chapter 7.

Let's adjust the load balancer in `shared.py` by setting the default service to the green network and removing the blue network from its backend services.

Listing 9.9 Remove blue environment load balancer

```
import blue    #A
import green    #A

TEAM = 'sundew'
ENVIRONMENT = 'production'
PORT = 8080

shared_name = f'{TEAM}-{ENVIRONMENT}-shared'

default_version = 'green'    #B

services_list = [    #C
   {
       'version': 'green',
       'zone': green.zone,
       'name': f'{shared_name}-green',
       'weight': 100    #D
   }
]
```

#A Import infrastructure as code for both blue and green environments.
#B Change the default version of the network for the load balancer to green.
#C Remove the blue network and instances from the list of backend services to generate.
#D Send all traffic to the green network.

You apply the changes to the load balancer. However, you do *not* delete the blue resources immediately because you must ensure the load balancer does not reference any blue resources. After testing the changes, you remove the code to build the blue environment from `main.py` and leave the green environment.

Listing 9.10 Remove blue environment from main.py

```
import green
import json

if __name__ == "__main__":
   resources = {
       'resource':
       shared.build() +    #C
       green.build()    #A
   }

   with open('main.tf.json', 'w') as outfile:    #B
       json.dump(resources, outfile,    #B
                 sort_keys=True, indent=4)    #B
```

#A Use the green module to create the JSON configuration for the network with global routing, servers with premium networking, and the cluster.
#B Write it out to a JSON file to be executed by Terraform later.
#C Use the shared module to create the JSON configuration for the global load balancer.

You apply the changes, and your infrastructure as code tool deletes all blue resources. You decide to delete the `blue.py` file to prevent anyone from creating new blue resources. I recommend removing any files you do not use to reduce confusion for your teammates in the future. Otherwise, you may have a system with more resources than you need.

Exercise 9.1

Given:

```
if __name__ == "__main__":
  network.build()
  queue.build(network)
  server.build(network, queue)
  load_balancer.build(server)
  dns.build(load_balancer)
```

The queue depends on the network. The server depends on the network and queue. How would you run a blue-green deployment to upgrade the queue with SSL?

Find answers and explanations at the end of the chapter.

9.2.6 Additional considerations

Imagine the sundew team needs to change the network again. Instead of creating a new green network, they can create a new blue network and repeat the deploy, regression test, and delete process! Since the old "blue" network no longer exists, their update does not conflict with an existing environment.

What you name your versions or iterations of changes does not matter, as long as you differentiate between old and new resources. For networking specifically, you consider allocating two sets of IP address ranges. You should permanently reserve one for the blue network and the other for the green network. The allocation will allow you the flexibility to make changes using blue-green deployment without the need to search for open network space.

In general, I decide to use a blue-green deployment strategy when I encounter the following:

- It takes a long time to revert changes to the resource.
- I am not confident that I can revert changes to the resource after deployment.
- The resource has many high-level dependencies that I cannot easily identify.
- The resource change affects critical applications that cannot have downtime.

Not all infrastructure resources should use blue-green deployment. For example, you can update identity and access management policies in place and quickly revert them if you identify a problem. You'll learn more about reverting changes in chapter 11.

A blue-green deployment strategy costs less in time and money than maintaining multiple environments. However, they will cost *more* when you have to deploy low-level infrastructure resources like networks, projects, or accounts! I usually consider the pattern worth the cost. It isolates the change to specific resources and provides a lower risk methodology for deploying changes and minimizing disruption to systems.

9.3 Stateful infrastructure

Throughout this chapter, the example omitted an essential class of infrastructure resources. However, the sundew system includes a number of resources that process, manage, and store data. For example, the sundew system includes a Google SQL database running on a network with regional routing.

9.3.1 Blue-green deployment

The sundew application team reminds you that they need to update their database to use the new network with global routing. You update the private network ID in infrastructure as code and push the changes to your repository. Your deployment pipeline fails on a compliance test (something we learned in chapter 8).

You notice that the test checks the dry run (plan) for whether or not the database deletes have failed.

```
$ pytest . -q

F                                                                                  [100%]
==========================================================================================
       FAILURES
       ==================================================================================
       ======
_________________________________________________________________________________
       test_if_plan_deletes_database
       __________________________________________________________________________________

database = {'address': 'google_sql_database_instance.blue', 'change': {'actions':
       ['delete'], 'after': None, 'after_sensitive': False, 'after_unknown': {}, ...},
       'mode': 'managed', 'name': 'blue', ...}

    def test_if_plan_deletes_database(database):
>       assert database['change']['actions'][0] != 'delete'
E       AssertionError: assert 'delete' != 'delete'

test/test_database_plan.py:35: AssertionError
======= short test summary info =======
FAILED test/test_database_plan.py::test_if_plan_deletes_database - AssertionError: assert
       'delete' != 'delete'
1 failed in 0.04s
```

The compliance test prevents you from deleting a critical database! If you applied the change without the test, you would delete *all* of the sundew data! The sundew team raises concerns about updating the database's network in place, so you need to do a blue-green deployment.

In figure 9.10, you manually verify if you migrate the database to the green network. However, you discover you cannot migrate the database. The sundew system can accommodate for missing data, so you copy the infrastructure as code for the blue database and create a new green database instance in the premium network. After migrating the data from the blue to green database, you switch applications to use the new database and remove the old one.

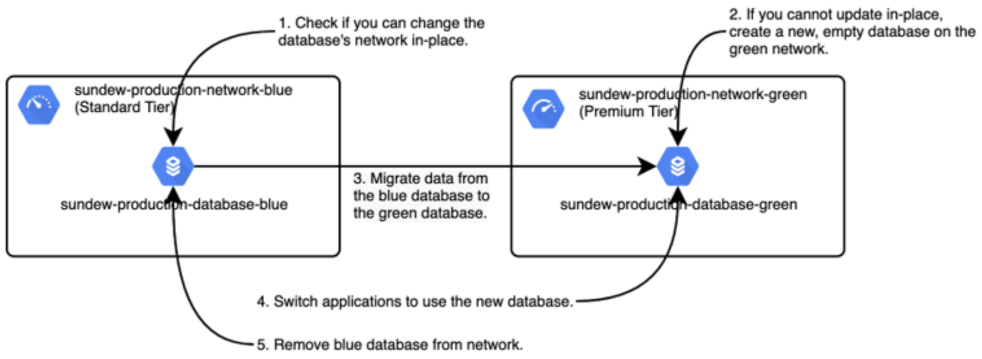

1. Check if you can change the database's network in-place.
2. If you cannot update in-place, create a new, empty database on the green network.
sundew-production-network-blue (Standard Tier)
sundew-production-database-blue
3. Migrate data from the blue database to the green database.
sundew-production-network-green (Premium Tier)
sundew-production-database-green
4. Switch applications to use the new database.
5. Remove blue database from network.

Figure 9.10. If you can't update the database in place, you must deploy a new green database, migrate and reconcile the data, and change the database endpoint from blue to green.

The blue-green deployment strategy applies to the database, a class of infrastructure resource that focuses on state. ***Stateful*** infrastructure resources like the database store and manage data. In reality, all applications process and store some amount of data. However, stateful infrastructure requires additional care because changes can directly affect data. This type of infrastructure includes databases, queues, caches, or stream processing tools.

DEFINITION Stateful infrastructure describes infrastructure resources that store and manage data.

Why use a blue-green deployment for infrastructure resources with data? Sometimes, you cannot update the resource in-place with infrastructure as code. Replacing the resource may corrupt or lose data, which affects your applications. A blue-green deployment helps you test the functionality of your new database before your applications use it.

9.3.2 Update delivery pipeline

Let's return to the sundew team. You must fix the delivery pipeline to automate the update for the database. In figure 9.11, you update your delivery pipeline with a step that automatically migrates data from the blue to green database. When you add the green database and deploy it, your pipeline deploys the new database, runs integration tests, automatically migrates data from the blue to green database, and completes the pipeline with end-to-end tests.

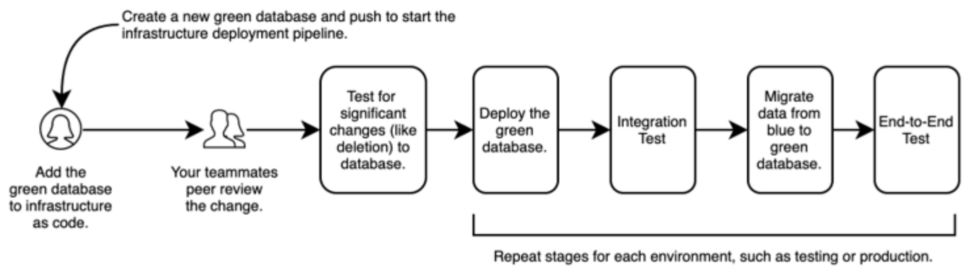

Create a new green database and push to start the infrastructure deployment pipeline.
Add the green database to infrastructure as code.
Your teammates peer review the change.
Test for significant changes (like deletion) to database.
Deploy the green database.
Integration Test
Migrate data from blue to green database.
End-to-End Test
Repeat stages for each environment, such as testing or production.

Figure 9.11. The infrastructure deployment pipeline should add the green database and copy data from blue to green.

Preserve *idempotency* with the automation for your migration step. Your script or automation for migration should result in the same database state each time. It should avoid duplicating data each time you run the automation. The data migration process differs depending on the kind of stateful infrastructure you have (database, queue, cache, or stream processing tools).

For more on managing stateful infrastructure

You can dedicate entire books to migrating and managing stateful infrastructure with minimal (or zero) downtime. I recommend *Database Reliability Engineering* by Charity Majors and Laine Campbell for other patterns and practices on managing databases. You can refer to their specific documentation on migration, upgrading, and availability for other stateful infrastructure resources.

Depending on how often you update your stateful infrastructure, you should capture the automated data migration to your deployment pipelines and *not* within your infrastructure as code. Separating data migration allows you to change any steps and debug problems with the migration independent of creating and removing the stateful resources.

9.3.3 Canary deployment

To complete the database update for the sundew team, you snake changes to application configuration to use the green database. As high-level resources, the applications depend on the database like a load balancer sends traffic to servers. You can use a modified form of canary deployment to cut over to the new database.

Figure 9.12 shows the pattern of regression testing and configuring the application to use the database. After a period of regression testing (to ensure that functionality still works), you update the application to *write* data exclusively to the green database. After another period of regression testing, you update the application to *read* data from the green database.

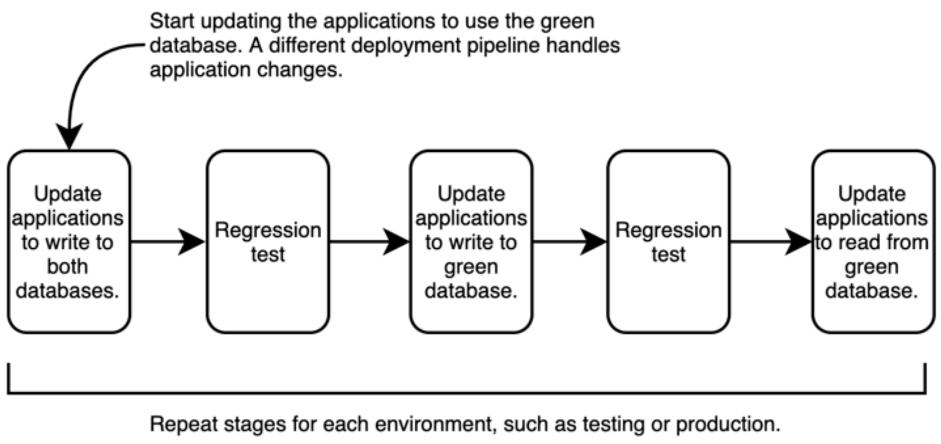

Start updating the applications to use the green database. A different deployment pipeline handles application changes.
Update applications to write to both databases.
Regression test
Update applications to write to green database.
Regression test
Update applications to read from green database.
Repeat stages for each environment, such as testing or production.

Figure 9.12. You incrementally update the application deployment pipeline to write to both databases, write to the green database, and then read from the green database.

You incrementally roll out the changes to the application to revert to the blue database if you encounter an issue. Since the application writes to two databases, you may have to write additional automation to reconcile data. However, writing to new stateful infrastructure ensures that you test any critical functionality related to storing and updating data. Only then can applications properly read and process data.

Now that the sundew applications use the green database, Figure 9.13 removes the blue database by deleting it from infrastructure as code. Note that the compliance tests will fail when you remove the database because you deleted a database! You update the test to fail if you plan on deleting a green database and not a blue one. The manual override allows you to remove the blue database since applications no longer use it.

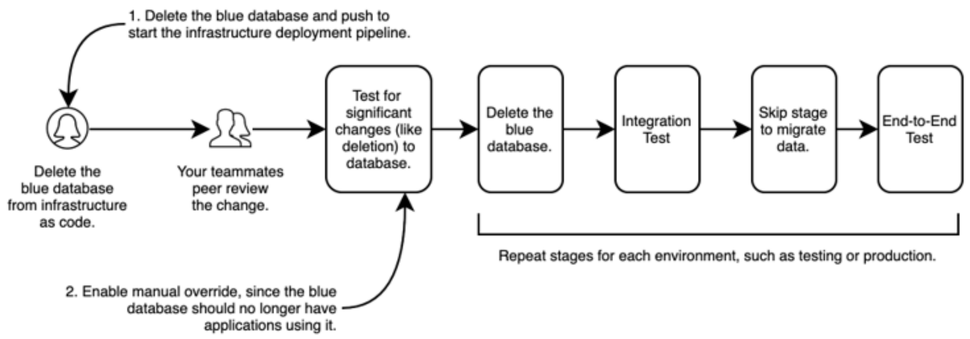

1. Delete the blue database and push to start the infrastructure deployment pipeline.
Delete the blue database from infrastructure as code.
Your teammates peer review the change.
Test for significant changes (like deletion) to database.
Delete the blue database.
Integration Test
Skip stage to migrate data.
End-to-End Test
Repeat stages for each environment, such as testing or production.
2. Enable manual override, since the blue database should no longer have applications using it.

Figure 9.13. Delete the blue database by removing it from infrastructure as code and pushing the changes into your deployment pipeline.

Techniques like canary deployment provide rapid feedback to mitigate the impact of failure, especially in a situation involving data processing. It can mean the difference between fixing a few wrong entries in a database versus restoring it entirely from backup! I get a sense of comfort knowing that I am making changes to stateful infrastructure in an isolated “green” environment instead of the live production system.

Strategies using immutability like blue-green deployment offer a structured process to make changes and minimize the blast radius of potential failures. Thanks to the principle of reproducibility, you can typically use an immutable approach for changing infrastructure as code by duplicating and editing configuration. The principle also allows you to improve the redundancy of your infrastructure system with a similar process of duplication.

9.4 Exercises and Solutions

Exercise 9.1

Given:

```
if __name__ == "__main__":
 network.build()
 queue.build(network)
 server.build(network, queue)
 load_balancer.build(server)
 dns.build(load_balancer)
```

The queue depends on the network. The server depends on the network and queue. How would you run a blue-green deployment to upgrade the queue with SSL?

Answer:

You create a "green" queue with SSL enabled. Then, you create a new "green" server that depends on the queue. Test that the application on the green server can access the queue with the SSL configuration. If it passes, you can add the green server to the load balancer. Gradually send traffic through the green server using a canary deployment until you confirm all requests succeed. Then, you remove the original server and queue.

Optionally, you can omit the step creating the "green" server and send traffic from the original server to SSL. However, this approach may not allow you to run a canary deployment. You update the server's configuration to directly communicate with the new queue.

9.5 Summary

- Ensure your system has testing, monitoring, and observability for infrastructure and applications before starting any infrastructure updates.
- Redundancy in infrastructure as code means adding extra, idle resources to the configuration so you can failover in case of component failure.
- Adding redundant configuration to infrastructure as code can improve system reliability, measuring how long a system performs correctly over some time.
- An active-passive configuration includes an active environment serving requests and a duplicate idle environment for when the active environment fails.
- Failover switches traffic from a failing active environment to an idle passive environment.
- An active-active configuration sets two active environments serving requests, both of which you can duplicate and manage with infrastructure as code.
- Blue-green deployment creates a new subset of infrastructure resources that stages the changes you want to make and gradually switches requests to the new subset. You can then remove the old set of resources.
- In a blue-green deployment, deploy the resource you want to change and the high-level resources that depend on it.
- Canary deployment sends a small percentage of traffic to the new infrastructure resources to verify whether or not the system works correctly. Over time, you increase the percentage of traffic.
- Allow a few weeks for regression testing to check if changes to the system affect existing or new functionality.
- Stateful infrastructure resources, like databases, caches, queues, and stream processing tools, store and manage data.
- Add a migration step to the blue-green deployment of stateful infrastructure resources. You must copy data between blue and green stateful infrastructure.

10 Refactoring

This chapter covers

- **Determining when to refactor infrastructure as code to avoid impacting systems**
- **Applying feature flagging to change infrastructure attributes mutably**
- **Explaining rolling updates to complete in-place updates**

Over time, you might outgrow the patterns and practices you use to collaborate on infrastructure as code. Even change techniques like blue-green deployment cannot solve conflicts in configuration or changes as your team works on some infrastructure as code. You must deliver a series of major changes to your infrastructure as code and address problems with scaling the practice.

For example, the sundew team for Datacenter for Carnivorous Plants expresses that they can no longer comfortably and confidently roll out new changes to their system. They put all infrastructure resources in one repository (as per the singleton pattern) to quickly deliver the system and just kept adding new updates on top of it.

The sundew team outlines a few problems with their system. First, the sundew team finds their updates to infrastructure configuration constantly overlapping. One teammate works on updating servers, only to find another teammate has updated the network and will affect their changes.

Second, it takes more than thirty minutes to run a single change. One change makes hundreds of calls to your infrastructure API to retrieve the state of resources, which slows down the feedback cycle.

Finally, the security team expresses concern that the sundew infrastructure may have an insecure configuration. The current configuration does not use standardized, hardened company infrastructure modules.

You realize you need to change the sundew team's configuration. The configuration should use an existing server module approved by the security team. You also need to break down the configuration into separate resources to minimize the blast radius of changes.

This chapter will discuss some infrastructure as code patterns and techniques to break down large singleton repositories with hundreds of resources. As the infrastructure as code helper on the sundew team, you'll refactor their system's singleton configuration into separate repositories and structure their server configurations to use modules to avoid conflicts and comply with security standards.

Code examples

Demonstrating refactoring requires a sufficiently large (and complex) example. If you run the complete example, you will incur a cost that exceeds the free tier in Google Cloud Platform (GCP). I only include the relevant lines of code in the text and omit the rest for readability. For complete listings, refer to the book's code repository at https://github.com/joatmon08/manning-book/tree/main/ch10.

10.1 Minimize refactoring impact

The sundew team needs help breaking down their infrastructure configuration. You decide to refactor the infrastructure as code to isolate conflicts better, reduce the amount of time to apply changes to production, and secure them according to company standards. ***Refactoring*** infrastructure as code involves restructuring configuration or code without impacting existing infrastructure resources.

> **DEFINITION** Refactoring infrastructure as code is the practice of restructuring configuration or code without impacting existing infrastructure resources.

You communicate to the sundew team that their configuration needs to undergo a refactor to fix their problems. While they support your effort, they challenge you to minimize the impact of your refactor. Challenge accepted you apply a few techniques to reduce the potential blast radius as you refactor infrastructure as code.

Technical debt

Refactoring often resolves technical debt. *Technical debt* began as a metaphor to describe the cost of any code or approach that makes the overall system challenging to change or extend. To understand technical debt applied to infrastructure as code, recall that the sundew team put all their infrastructure resources into one repository. The sundew team accumulates debt in time and effort. A change to a server that *should* take a day takes four days because they need to resolve conflicts with another change and wait for hundreds of requests to the infrastructure API. Note that you'll always have some technical debt in complex systems, but you need continuous efforts to minimize it.

A management team dreads hearing that you need to address technical debt because you don't work on features. I argue that the technical debt you accumulate in infrastructure will always come back to haunt you. The gremlin of technical debt comes in the form of someone changing infrastructure and causing application downtime, or worse, a security breach that exposes personal information and incurs a monetary cost. Assessing the impact of *not* fixing technical debt helps justify the effort.

10.1.1 Reduce blast radius with rolling updates

The Datacenter for Carnivorous Plants platform and security team offer a server module with secure configurations, which you can use for the sundew system. The sundew team's infrastructure configuration has three server configurations but no usage of the secure module. How do you change the sundew infrastructure as code to use the module?

Imagine you create three new servers together and immediately send traffic to them. If the applications do not run correctly on the server, you could disrupt the sundew system entirely, and the poor plants do not get watered! Instead, you might reduce the blast radius of your server module refactor by gradually changing the servers one-by-one.

In figure 10.1, you create one server configuration using the module, deploy the application to the new server, validate the application works, and delete the old server. You repeat the process two more times for each server. You gradually roll out the change to one server before updating the next one.

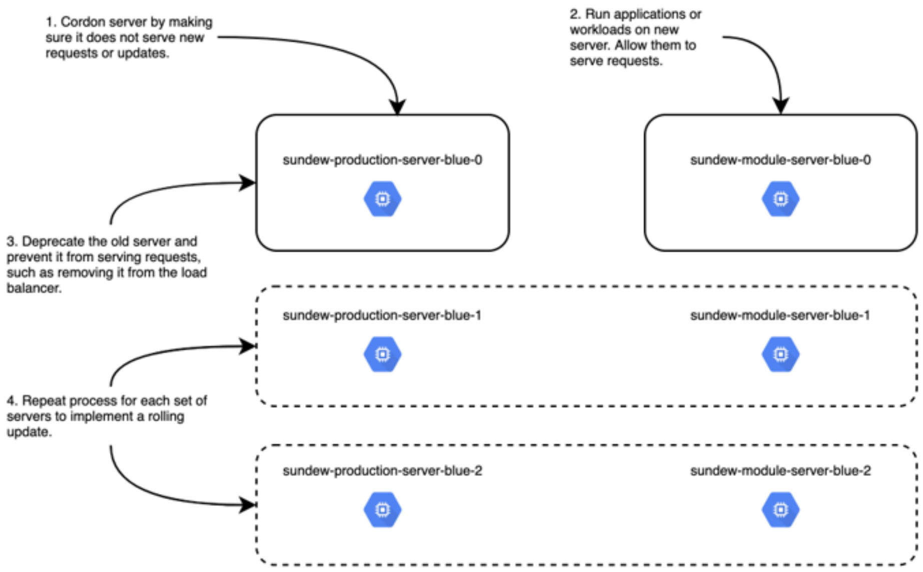

1. Cordon server by making sure it does not serve new requests or updates.
2. Run applications or workloads on new server. Allow them to serve requests.
sundew-production-server-blue-0
sundew-module-server-blue-0
3. Deprecate the old server and prevent it from serving requests, such as removing it from the load balancer.
sundew-production-server-blue-1
sundew-module-server-blue-1
4. Repeat process for each set of servers to implement a rolling update.
sundew-production-server-blue-2
sundew-module-server-blue-2

Figure 10.1. Use rolling updates to create each new server, deploy the application, and test its functionality while minimizing disruption to other resources.

A ***rolling update*** gradually changes similar resources one by one and tests *each* one before continuing the update.

> **DEFINITION** A rolling update is a practice of changing a group of similar resources one by one and testing them before implementing the change to the next one.

Applying a rolling update to the sundew team's configuration isolates failures to individual servers each time you make the update and allows you to test the server's functionality before proceeding to the next one.

The practice of rolling updates can save you the pain of detangling a large set of failed changes or incorrectly configured infrastructure as code. For example, if the Datacenter for Carnivorous Plants module doesn't work on one server, you have not yet rolled it out and affected the remaining servers. A rolling update lets you check that you have the proper infrastructure as code for each server before continuing to the next one. A gradual approach also mitigates any downtime in the applications or failures in updating the servers.

Rolling updates for workload orchestration

I borrowed "rolling updates" for refactoring from workload orchestrators, like Kubernetes. When you need to update new nodes (virtual machines) for workload orchestrators, you may find it uses an automated rolling update mechanism. The orchestrator cordons the old node, prevents new workloads from running on it, starts all of the running processes on a new node, drains all of the processes on the old node, and sends workloads and requests to the new node. You should mimic the workflow when you refactor!

Thanks to the rolling update and incremental testing, you know that the servers can run with the secure module. You tell the team that you finished refactoring the servers and confirmed they work with internal services. The sundew team can now send all customer traffic to the newly secured servers. However, the sundew team tells you that they need to update the customer-facing load balancer first!

10.1.2 Stage refactoring with feature flags

You need a way to hide the new servers from the customer-facing load balancer for a few days and attach them when the team approves. However, you have all the configurations ready! You want to hide the server attachments with a single variable to simplify the sundew team. When they complete their load balancer update, they only need to update one variable to add the new servers to the load balancer.

Figure 10.2 outlines how you set up the variable to add new servers created by the module. You create a boolean to enable or disable the new server module using true or false. Then, you add an if-statement to infrastructure as code that references the boolean value. A true variable adds the new servers to the load balancer. A false variable removes the servers from the load balancer.

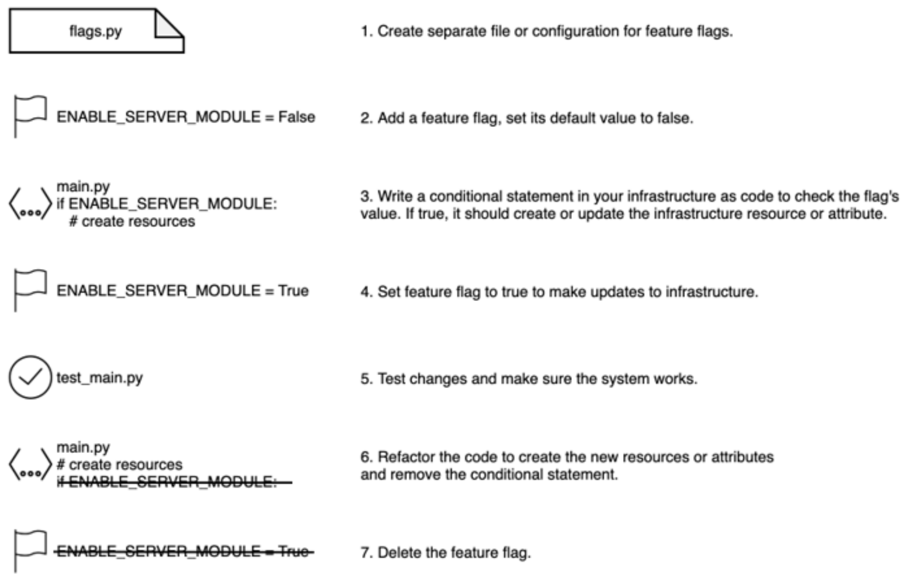

flags.py
1. Create separate file or configuration for feature flags.
ENABLE_SERVER_MODULE = False
2. Add a feature flag, set its default value to false.
main.py
if ENABLE_SERVER_MODULE:
# create resources
3. Write a conditional statement in your infrastructure as code to check the flag's value. If true, it should create or update the infrastructure resource or attribute.
ENABLE_SERVER_MODULE = True
4. Set feature flag to true to make updates to infrastructure.
test_main.py
5. Test changes and make sure the system works.
main.py
# create resources
~~if ENABLE_SERVER_MODULE:~~
6. Refactor the code to create the new resources or attributes and remove the conditional statement.
~~ENABLE_SERVER_MODULE = True~~
7. Delete the feature flag.

Figure 10.2. Feature flags in infrastructure as code involve a process of creation, management, and removal.

The boolean variable helps with the composability or evolvability of infrastructure as code. A single change to the variable adds, removes, or updates a configuration. The variable, called a ***feature flag*** (or feature toggle), to enable or disable infrastructure resources, dependencies, and attributes. You often find feature flags in software development with a trunk-based development model (only work on the main branch).

> **DEFINITION** Feature flags (also known as feature toggles) enable or disable infrastructure resources, dependences, or attributes using a boolean.

Flags hide certain features or code and prevent them from impacting the rest of the team on the main branch. For the sundew team, you hide the new servers from the load balancer until they complete the load balancer change. Similarly, you can use feature flags in infrastructure as code to stage configuration and push the update with a single variable.

SET UP THE FLAG

To start implementing a feature flag and stage changes for the new servers, you add a flag and set it to false. In figure 10.3, you default a feature flag to false to preserve the original infrastructure state. The sundew configuration disables the server module by default so that nothing happens to the original servers.

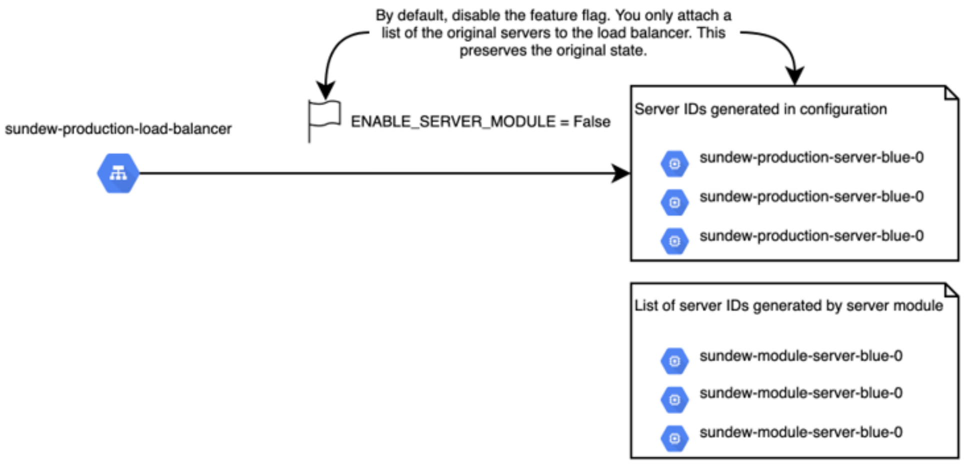

By default, disable the feature flag. You only attach a list of the original servers to the load balancer. This preserves the original state.
sundew-production-load-balancer
ENABLE_SERVER_MODULE = False
Server IDs generated in configuration
sundew-production-server-blue-0
sundew-production-server-blue-0
sundew-production-server-blue-0
List of server IDs generated by server module
sundew-module-server-blue-0
sundew-module-server-blue-0
sundew-module-server-blue-0

Figure 10.3. Set the feature flag to false by default to preserve the infrastructure resources' original state and dependencies.

Let's implement the feature flag in Python. You set the server module flag's default to false in a separate file called `flags.py`. The file defines the flag, `ENABLE_SERVER_MODULE`, and sets it to false.

```
ENABLE_SERVER_MODULE = False
```

You could also embed feature flags as variables in other files, but you might lose track of them! You decide to put them in a separate Python file.

> **NOTE** I always define feature flags in a file to identify and change them in one place.

Listing 10.1 imports the feature flag in `main.py` and adds the logic to generate a list of servers to add to the load balancer.

Listing 10.1 Include the feature flag to add servers to the load balancer

```
import flags    #A

def _generate_servers(version):
   instances = [    #D
       f'${{google_compute_instance.{version}_0.id}}',    #D
       f'${{google_compute_instance.{version}_1.id}}',    #D
       f'${{google_compute_instance.{version}_2.id}}'    #D
   ]    #D
   if flags.ENABLE_SERVER_MODULE:    #B
       instances = [    #C
           f'${{google_compute_instance.module_{version}_0.id}}',    #C
           f'${{google_compute_instance.module_{version}_1.id}}',    #C
           f'${{google_compute_instance.module_{version}_2.id}}',    #C
       ]    #C
   return instances
```

#A Import the file that defines all of the feature flags.
#B Use a conditional statement to evaluate the feature flag and add the server module's resources to the load balancer.
#C A feature flag set to true will attach the servers created by the module to the load balancer. Otherwise, it will keep the original servers.
#D Define a list of existing Google compute instances (servers) using a Terraform resource in the system.

Run Python with the feature flag toggled off to generate a JSON configuration. The resulting JSON configuration only adds the original servers to the load balancer, which preserves the existing state of infrastructure resources.

Listing 10.2 JSON configuration with feature flag disabled

```
{
   "resource": [
       {
           "google_compute_instance_group": {    #A
               "blue": [
                   {
                       "instances": [
                           "${google_compute_instance.blue_0.id}",    #B
                           "${google_compute_instance.blue_1.id}",    #B
                           "${google_compute_instance.blue_2.id}"    #B
                       ]
                   }
               ]
           }
       }
   ]
}
```

#A Create a Google compute instance group using a Terraform resource to the load balancer.
#B Configuration includes a list of the original Google compute instances, preserving the current state of infrastructure resources.

The feature flag set to false by default uses the principle of idempotency. When you run the infrastructure as code, your infrastructure state should not change. Setting the flag ensures you do not accidentally change existing infrastructure. Preserving the original state of the existing servers minimizes disruption to dependent applications.

Enable the flag

The sundew team made their changes and provided approval to add the new servers created by the module to the load balancer. In figure 10.4, you set the feature flag to true. When you deploy the change, you attach servers from a module to the load balancer and remove the old servers.

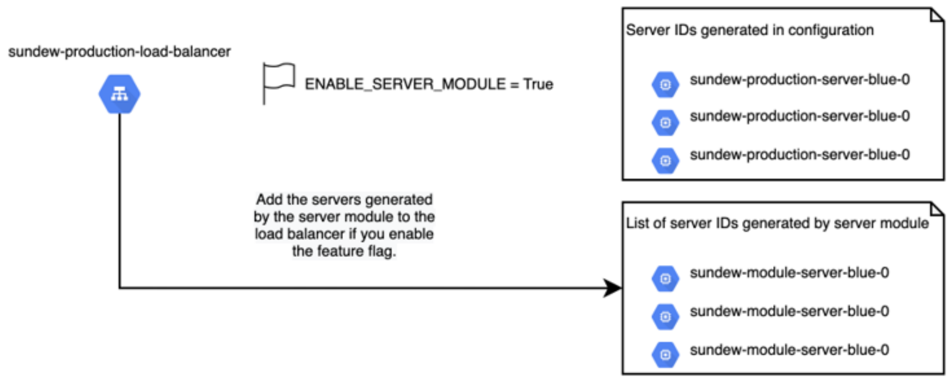

sundew-production-load-balancer
ENABLE_SERVER_MODULE = True
Server IDs generated in configuration
sundew-production-server-blue-0
sundew-production-server-blue-0
sundew-production-server-blue-0
Add the servers generated by the server module to the load balancer if you enable the feature flag.
List of server IDs generated by server module
sundew-module-server-blue-0
sundew-module-server-blue-0
sundew-module-server-blue-0

Figure 10.4. Set the feature flag to true, attach the three new servers created by the module to the load balancer, and detach the old servers.

Let’s examine the updated feature flag in action. You start by setting the feature flag for the servers to true.

```
ENABLE_SERVER_MODULE = True
```

Run Python to generate a new JSON configuration. The configuration now includes the servers created by the module to the load balancer.

Listing 10.3 JSON configuration with feature flag enabled

```
{
   "resource": [
       {
           "google_compute_instance_group": {
               "blue": [
                   {
                       "instances": [
                           "${google_compute_instance.module_blue_0.id}",    #A
                           "${google_compute_instance.module_blue_1.id}",    #A
                           "${google_compute_instance.module_blue_2.id}"    #A
                       ]
                   }
               ]
           }
       }
   ]
}
```

#A The new servers created by the module replace the old servers because you enabled the feature flag.

The feature flag allows you to stage the module's low-level server resources without affecting the load balancer's high-level dependency. You can re-run the code with the feature toggle off to re-attach the old servers.

Why use the feature flag to switch to the server module? A feature flag hides functionality from production until you feel ready to deploy resources associated with it. You offer one variable to add, remove, or update a set of resources. You can also use the same variable to revert changes.

REMOVE THE FLAG

After running the servers for some time, the sundew team reports that the new server module works. You can now remove the old servers. You no longer need the feature flag, and you don't want to confuse another team member when they read the code.

You refactor the Python code for the load balancer to remove the old servers and delete the feature flag.

Listing 10.4 Remove the feature flag once change completes

```
import blue    #A


def _generate_servers(version):
   instances = [    #B
       f'${{google_compute_instance.module_{version}_0.id}}',    #B
       f'${{google_compute_instance.module_{version}_1.id}}',    #B
       f'${{google_compute_instance.module_{version}_2.id}}',    #B
   ]    #B
   return instances
```

#A You can remove the import for the feature flags because you no longer need it for your servers.
#B Permanently attach the servers created by the module to the load balancer and remove the feature flag.

Feature flags in domain-specific languages (DSLs)

The example shows a feature flag in a programming language. You can also use feature flags in domain-specific languages, although you must adapt them based on your tool's syntax. In HashiCorp Terraform, you can mimic a feature flag using a variable and the `count` meta-argument (https://www.terraform.io/docs/language/meta-arguments/count.html).

```
variable "enable_server_module" {
 type       = bool
 default    = false
 description = "Choose true to build servers with a module."
}

module "server" {
 count   = var.enable_server_module ? 1 : 0
 ## omitted for clarity
}
```

In AWS CloudFormation, you can pass a parameter and set a condition ([https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/conditions-section-structure.html)](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/conditions-section-structure.html) to enable or disable resource creation.

```
AWSTemplateFormatVersion: 2010-09-09
Description: Truncated example for CloudFormation feature flag
Parameters:
 EnableServerModule:
   AllowedValues:
     - 'true'
     - 'false'
   Default: 'false'
   Description: Choose true to build servers with a module.
   Type: String
Conditions:
 EnableServerModule: !Equals
   - !Ref EnableServerModule
   - true
Resources:
 ServerModule:
   Type: AWS::CloudFormation::Stack
   Condition: EnableServerModule
   ## omitted for clarity
```

Besides feature flags to enable and disable entire resources, you can use conditional statements to enable or disable specific attributes for a resource.

As a general rule, *remove* the feature flag after you finish the change. Too many feature flags can clutter your infrastructure as code with complicated logic, making it hard to troubleshoot infrastructure configuration.

USE CASES

The example uses feature flags to *refactor singleton configurations into infrastructure modules*. I often apply feature flags to this use case to simplify the creation and removal of infrastructure resources. Other use cases for feature flags include:

- Collaborating and avoiding change conflicts on the *same infrastructure* resources or dependencies
- Staging a *group of changes* and rapidly deploying them with a single update to the flag
- *Testing* a change and quickly disabling it on failure

A feature flag offers a technique to hide or isolate infrastructure resource, attribute, and dependency changes during the refactoring of infrastructure configuration. However, changing the toggle can still disrupt a system. In the example of the sundew team's servers, we cannot simply toggle the feature flag to true and expect the servers to run the application. Instead, we combine the feature flag with other techniques like rolling updates to minimize disruption to the system.

10.2 Breaking down monoliths

The sundew team expresses that they still have a problem with their system. You identify the singleton configuration with hundreds of resources and attributes as the root cause. Whenever someone makes a change, the teammate must resolve conflicts with another person. They also have to wait thirty minutes to make a change.

A ***monolithic architecture*** for infrastructure as code means defining all infrastructure resources in one place. You need to break down the monolith of infrastructure as code into smaller, modular components to minimize working conflicts between teammates and speed up the deployment of changes.

> **DEFINITION** A monolithic architecture for infrastructure as code defines all infrastructure resources in a single configuration and the same state.

In this section, we'll walk through a refactor of the sundew team's monolith. The most crucial step begins with identifying and grouping high-level infrastructure resources and dependencies. We complete the refactor with the low-level infrastructure resources.

Monolith versus mono-repository

Recall that you can put your infrastructure configuration into a single repository (Chapter 5). Does a single repository mean you have a monolithic architecture? Not necessarily. You can subdivide a single repository into separate subdirectories. Each subdirectory contains a separate infrastructure as code.

A monolithic architecture means you manage many resources together and tightly couple them, making it difficult to change a subset in isolation. The monolith usually results from an initial singleton pattern (all configurations in one place) that expands over time.

You might have noticed that I started immediately with patterns for modularizing infrastructure resources and dependencies in chapters 3 and 4. Why not move this chapter on refactoring earlier? If you can identify and apply some of the patterns early in infrastructure as code development, you can avoid the monolithic architecture. However, you sometimes inherit a monolith and often need to refactor it.

10.2.1 Refactor high-level resources

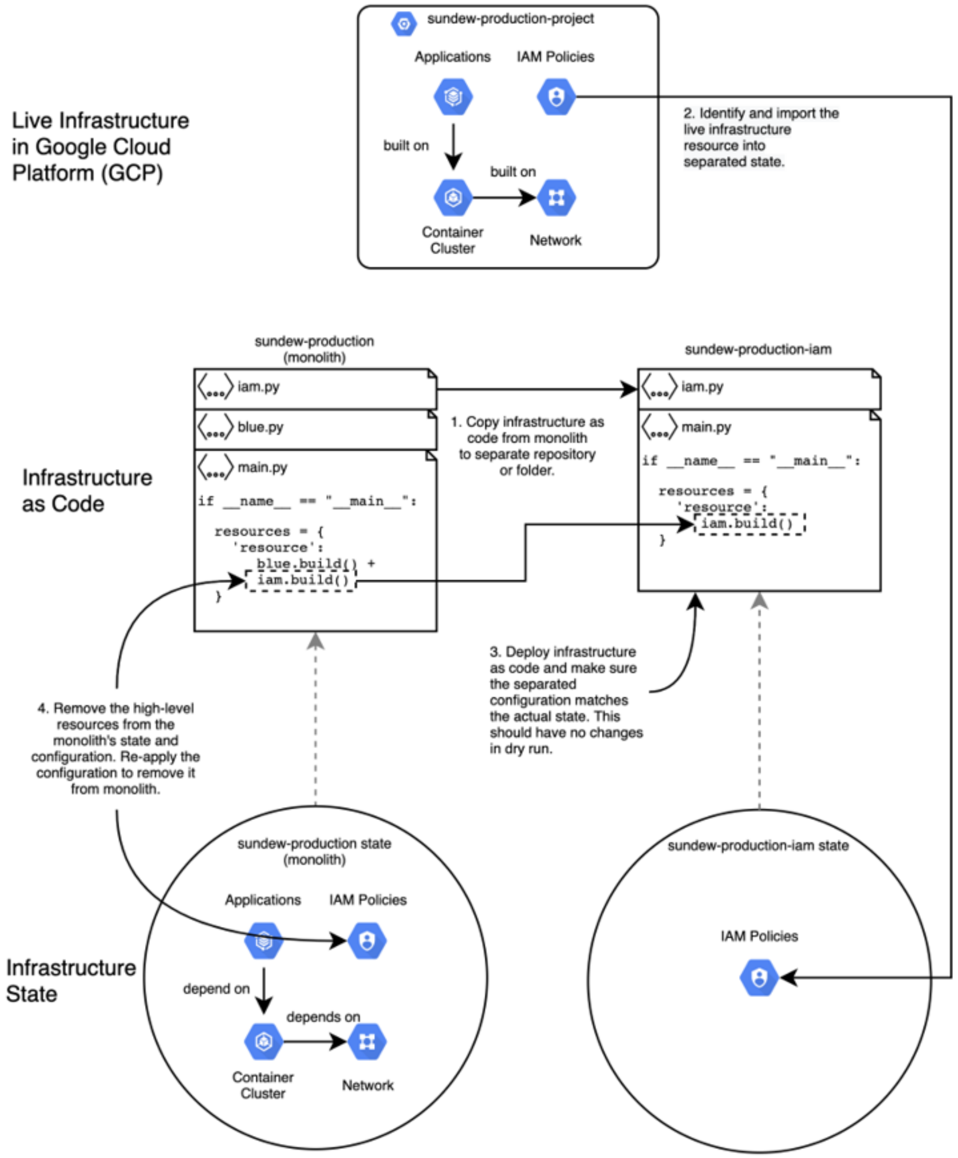
The sundew team manages hundreds of resources in one set of configuration files. Where should you start breaking down the infrastructure as code? You decide to look for *high-level infrastructure resources* that do not depend on other resources.

The sundew team has one set of high-level infrastructure in Google Cloud Platform (GCP) project-level identity and access management (IAM) service accounts and roles. The IAM service accounts and roles don’t need to create a network or server before setting user and service account rules on the project. None of the other resources depend on the IAM roles and service accounts. You can group and extract them first.

You cannot use a blue-green deployment approach because GCP does not allow duplicate policies. However, you cannot simply delete roles and accounts from the monolithic configuration and copy them to a new repository. Deleting them prevents everyone from logging into the project! How can you extract them?

You can copy and paste the configuration into its separate repository or directory, initialize the state for the separated configuration, and import the resources into the infrastructure state associated with the new configuration. Then, you delete the identity and access management configuration in the monolithic configuration. Like the rolling update, you gradually change each set of infrastructure resources, test the changes, and proceed to the next one.

Figure 10.5 outlines the solution to refactoring a monolith for high-level resources. You copy the code from the monolith to a new folder and import the live infrastructure resource into the state of the code in the new folder. You re-deploy the code to make sure it does not change existing infrastructure. Finally, remove the high-level resources from the monolith.


Live Infrastructure in Google Cloud Platform (GCP)
sundew-production-project
Applications
IAM Policies
built on
built on
Container Cluster
Network
2. Identify and import the live infrastructure resource into separated state.
Infrastructure as Code
sundew-production (monolith)
iam.py
blue.py
main.py
1. Copy infrastructure as code from monolith to separate repository or folder.
sundew-production-iam
iam.py
main.py
3. Deploy infrastructure as code and make sure the separated configuration matches the actual state. This should have no changes in dry run.
4. Remove the high-level resources from the monolith's state and configuration. Re-apply the configuration to remove it from monolith.
Infrastructure State
sundew-production state (monolith)
Applications
IAM Policies
depend on
depends on
Container Cluster
Network
sundew-production-iam state
IAM Policies

```
if __name__ == "__main__":

  resources = {
    'resource':
      blue.build() +
      iam.build()
  }
```

```
if __name__ == "__main__":

  resources = {
    'resource':
      iam.build()
  }
```

Figure 10.5. The sundew system’s identity and access management (IAM) policies for the Google Cloud Platform (GCP) project have no dependencies, and you can refactor them easily without disrupting other infrastructure.

Like the case with feature flags, we use the principle of idempotency to run infrastructure as code and verify that we do not affect the active infrastructure state. Any time you refactor, make sure you deploy the changes and check the dry run. You do not want to accidentally change an existing resource and affect its dependencies.

We will refactor the example in the following few sections. Stay with it! I know refactoring tends to feel tedious, but a gradual approach ensures you do not introduce widespread failures into your system.

COPY FROM MONOLITH TO SEPARATE STATE

Your initial refactor begins by copying the code to create the IAM roles and service accounts to a new directory. The sundew team wants to keep a single repository structure that stores all the infrastructure as code in one source control repository but separates the configurations into folders.

In figure 10.6, you identify the IAM roles and service accounts to copy their code to a new folder. The active IAM policies and their infrastructure state in Google Cloud Platform (GCP) do not change.

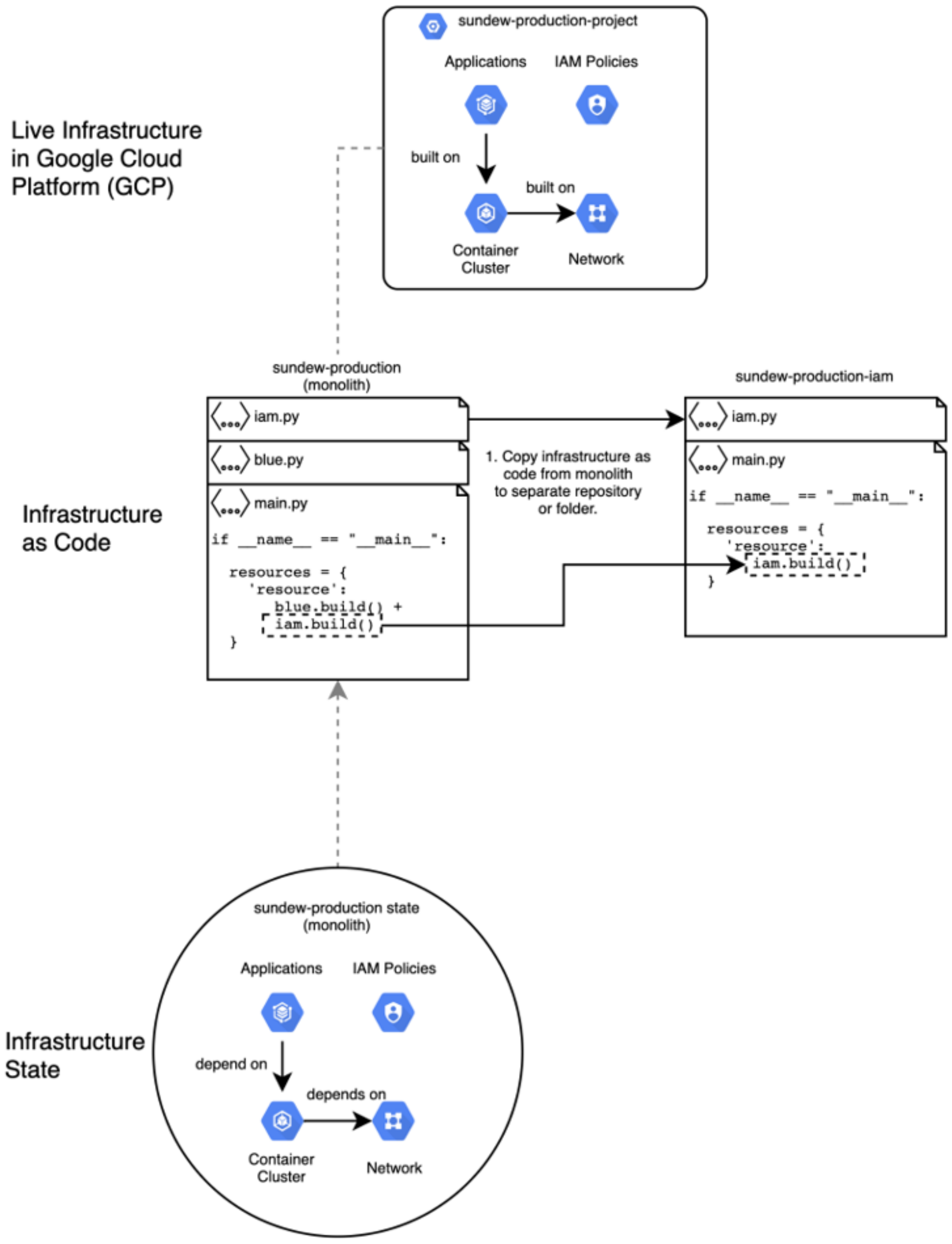


Figure 10.6. Copy the files for the IAM policies into a new directory for the sundew-production-iam configuration, and avoid changing the live infrastructure resources in GCP.

Why reproduce the infrastructure as code for the IAM policies in a separate folder? You want to split up your monolithic infrastructure as code without affecting any of the active resources. The most important practice when refactoring involves preserving idempotency. Your active state should never change when you move your infrastructure as code.

Let's start refactoring the IAM policies out of the monolith. Create a new directory that only manages the IAM policies for the GCP project.

```
$ mkdir -p sundew_production_iam
```

Copy the IAM configuration from the monolith into the new directory.

```
$ cp iam.py sundew_production_iam/
```

You don't need to change anything since the IAM policies do not depend on other infrastructure. The file `iam.py` separates the creation and role assignment for a set of users.

Listing 10.5 IAM configuration separated from the monolith

```
import json

TEAM = 'sundew'
TERRAFORM_GCP_SERVICE_ACCOUNT_TYPE = 'google_service_account'    #D
TERRAFORM_GCP_ROLE_ASSIGNMENT_TYPE = 'google_project_iam_member'    #D

users = {    #A
   'audit-team': 'roles/viewer',    #A
   'automation-watering': 'roles/editor',    #A
   'user-02': 'roles/owner'    #A
}    #A

def get_user_id(user):
   return user.replace('-', '_')

def build():
   return iam()

def iam(users=users):    #E
   iam_members = []
   for user, role in users.items():
       user_id = get_user_id(user)
       iam_members.append({
           TERRAFORM_GCP_SERVICE_ACCOUNT_TYPE: [{     #B
               user_id: [{     #B
                   'account_id': user,     #B
                   'display_name': user     #B
               }]     #B
           }]     #B
       })
       iam_members.append({
           TERRAFORM_GCP_ROLE_ASSIGNMENT_TYPE: [{     #C
               user_id: [{     #C
                   'role': role,     #C
                   'member': 'serviceAccount:${google_service_account.'     #C
                   + f'{user_id}' + '.email}'     #C
               }]     #C
           }]     #C
       })
   return iam_members
```

#A Keep all of the users you added to the project as part of the monolith.
#B Create a GCP service account for the project for each user in the sundew production project.
#C Assign the specific role defined for each service account, such as viewer, editor, or owner.
#D Set the resource types that Terraform uses as constants so you can reference them later if needed.
#E Use the module to create the JSON configuration for the IAM policies outside of the monolith.

AWS equivalents

The example code creates all users and groups as service accounts in GCP so you can run the example to completion. You typically use service accounts for automation. A service account in GCP is similar to an AWS IAM user dedicated to service automation.

Set constants and create methods that output resource types and identifiers when separating configuration. You can always use them for other automation and continued system maintenance, especially as you continue to refactor the monolith!

Create a `main.py` file in the `sundew_production_iam` folder that references the IAM configuration and outputs the Terraform JSON for it.

Listing 10.6 Entrypoint to build the separate JSON configuration for IAM

```
import iam    #A
import json

if __name__ == "__main__":
   resources = {
       'resource': iam.build()    #A
   }

   with open('main.tf.json', 'w') as outfile:    #B
       json.dump(resources, outfile,    #B
                 sort_keys=True, indent=4)    #B
```

#A Import the IAM configuration code and build the IAM policies.
#B Write it out to a JSON file to be executed by Terraform later.

Do not run Python yet to create the Terraform JSON or deploy the IAM policies! You already have IAM policies defined as part of Google Cloud Platform. If you run `python main.py` and apply the Terraform JSON with the separated IAM configuration, GCP throws an error that the user account and assignment already exists.

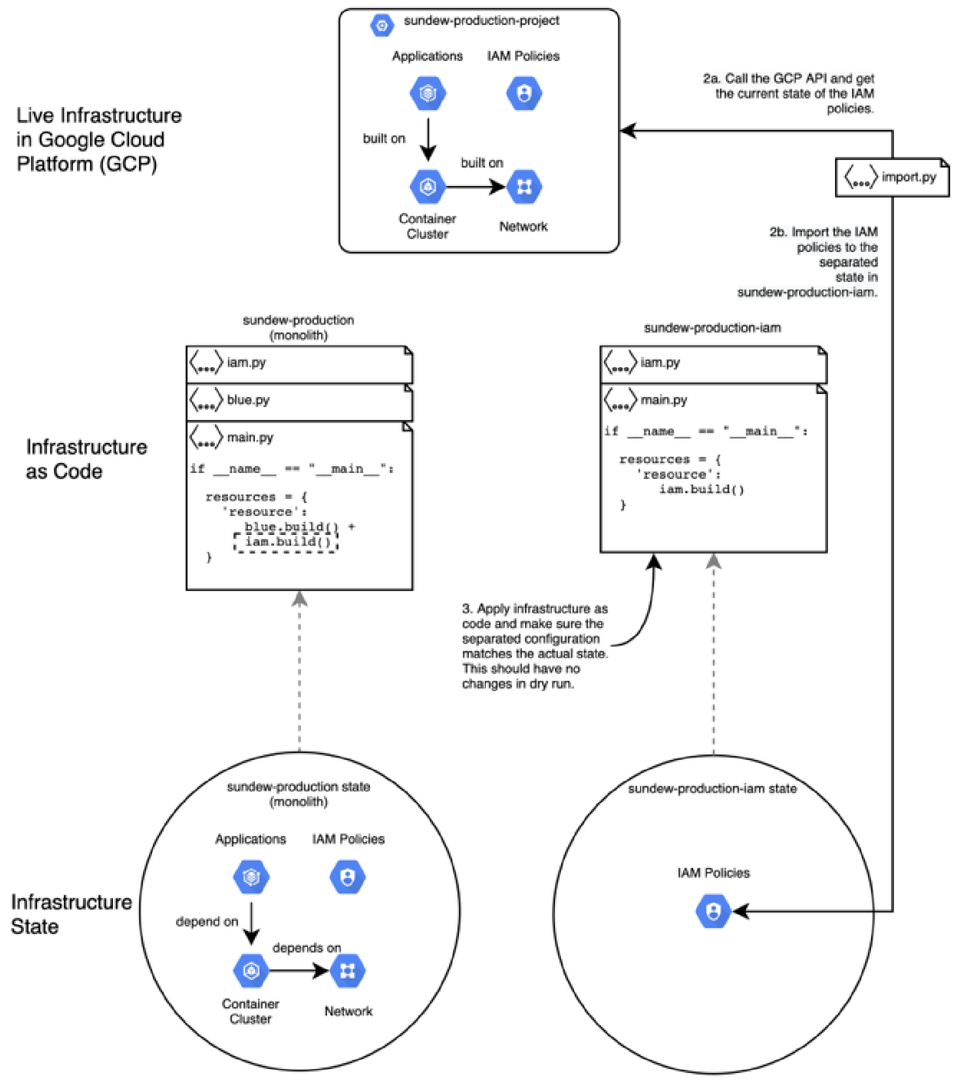
```
$ python main.py

$ terraform apply -auto-approve
## output omitted for clarity
│ Error: Error creating service account: googleapi: Error 409: Service account audit-team
     already exists within project projects/infrastructure-as-code-book., alreadyExists
## output omitted for clarity
```

The sundew team does not want you to remove and create new accounts and roles. If you delete and create new accounts, they cannot log into their GCP project. You need a way to migrate the existing resources defined in the monolith and link them to code defined in its own folder.

IMPORT THE RESOURCES TO THE NEW STATE

Sometimes, creating new resources with your refactored infrastructure as code will disrupt development teams and business-critical systems. You cannot use the principle of immutability to delete the old resources and create new ones. Instead, you must migrate active resources from one infrastructure as code definition to another.

In the case of the sundew team, you extract the identifiers for each service account from the monolithic configuration and "move" them to the new state. Figure 10.7 demonstrates how to detach each service account and their role assignments from the monolith and attach it to the infrastructure as code in the `sundew_production_iam` directory. You call the GCP API for the current state of the IAM policies and *import* the live infrastructure resources into the separated configuration and state. Running the infrastructure as code should reveal no changes to its dry run.


Live Infrastructure in Google Cloud Platform (GCP)
sundew-production-project
Applications
IAM Policies
built on
built on
Container Cluster
Network
2a. Call the GCP API and get the current state of the IAM policies.
import.py
2b. Import the IAM policies to the separated state in sundew-production-iam.
Infrastructure as Code
sundew-production (monolith)
iam.py
blue.py
main.py
sundew-production-iam
iam.py
main.py
3. Apply infrastructure as code and make sure the separated configuration matches the actual state. This should have no changes in dry run.
Infrastructure State
sundew-production state (monolith)
Applications
IAM Policies
depend on
depends on
Container Cluster
Network
sundew-production-iam state
IAM Policies

```
if __name__ == "__main__":

  resources = {
    'resource':
      blue.build() +
      iam.build()
  }
```

```
if __name__ == "__main__":

  resources = {
    'resource':
      iam.build()
  }
```

Figure 10.7. Get the current state of the separated resources from the infrastructure provider and import the identifiers before re-applying infrastructure as code.

Why import the IAM policy information with the GCP API? You want to import the updated, active state of the resource. A cloud provider's API offers the most up-to-date configuration for resources. You can call the GCP API to retrieve the user emails, roles, and identifiers for the sundew team.

Rather than write your own import capability and save the identifiers in a file, you decide to use Terraform's import capability to add existing resources to the state. You write some Python code that wraps around Terraform to automate a batch import of IAM resources so that the sundew team can reuse it.

Listing 10.7 File import.py separately imports sundew IAM resources

```
import iam    #A
import os
import googleapiclient.discovery    #B
import subprocess

PROJECT = os.environ['CLOUDSDK_CORE_PROJECT']    #H

def _get_members_from_gcp(project, roles):    #B
   roles_and_members = {}    #B
   service = googleapiclient.discovery.build(    #B
       'cloudresourcemanager', 'v1')    #B
   result = service.projects().getIamPolicy(    #B
       resource=project, body={}).execute()    #B
   bindings = result['bindings']    #B
   for binding in bindings:    #B
       if binding['role'] in roles:    #B
           roles_and_members[binding['role']] = binding['members']    #B
   return roles_and_members    #B

def _set_emails_and_roles(users, all_members):    #C
   members = []    #C
   for username, role in users.items():    #C
       members += [(iam.get_user_id(username), m, role)    #C
                   for m in all_members[role] if username in m]    #C
   return members    #C

def check_import_status(ret, err):
   return ret != 0 and \
       'Resource already managed by Terraform' not in str(err)

def import_service_account(project_id, user_id, user_email):    #D
   email = user_email.replace('serviceAccount:', '')    #D
   command = ['terraform', 'import', '-no-color',    #D
              f'{iam.TERRAFORM_GCP_SERVICE_ACCOUNT_TYPE}.{user_id}',     #D
              f'projects/{project_id}/serviceAccounts/{email}']    #D
   return _terraform(command)    #D

def import_project_iam_member(project_id, role,    #E
                              user_id, user_email):    #E
   command = ['terraform', 'import', '-no-color',    #E
              f'{iam.TERRAFORM_GCP_ROLE_ASSIGNMENT_TYPE}.{user_id}',    #E
              f'{project_id} {role} {user_email}']    #E
   return _terraform(command)    #E

def _terraform(command):    #F
   process = subprocess.Popen(    #F
       command,    #F
       stdout=subprocess.PIPE,    #F
       stderr=subprocess.PIPE)    #F
   stdout, stderr = process.communicate()    #F
   return process.returncode, stdout, stderr    #F
```

```
if __name__ == "__main__":
   sundew_iam = iam.users    #A
   all_members_for_roles = _get_members_from_gcp(    #B
       PROJECT, set(sundew_iam.values()))    #B
   import_members = _set_emails_and_roles(    #C
       sundew_iam, all_members_for_roles)    #C
   for user_id, email, role in import_members:
       ret, _, err = import_service_account(PROJECT,    #D
                                            user_id, email)    #D
       if check_import_status(ret, err):    #G
           print(f'import service account failed: {err}')    #G
       ret, _, err = import_project_iam_member(PROJECT, role,    #E
                                               user_id, email)    #E
       if check_import_status(ret, err):    #G
           print(f'import iam member failed: {err}')    #G
```

#A Retrieve the list of sundew users from iam.py in sundew_production_iam.
#B Use the Google Cloud Client Libraries for Python to get a list of members assigned to a role in the GCP project.
#C Get the email and user ids for the sundew IAM members only.
#D Import the service account to sundew_production_iam state based on project and user email, using the resource type constant you set in iam.py.
#E Import a role assignment to sundew_production_iam state based on project, role, and user email, using the resource type constant you set in iam.py.
#F Both import methods wrap around the Terraform CLI command and return any errors and output.
#G If the import fails and it did not already import the resource, output the error.
#H Retrieve the GCP project ID from the CLOUDSDK_CORE_PROJECT environment variable.

Like defining dependencies, you want to *dynamically retrieve identifiers* from the infrastructure provider API for your resources to import. You never know when someone changes the resource, and the identifier you thought you needed no longer exists! Use your tags and naming conventions to search the API response for the resources you need.

Why not use Apache Libcloud?

The examples in this chapter need to use the Google Cloud Client Library for Python instead of Apache Libcloud. While Apache Libcloud works for retrieving information about virtual machines, it does not work for other resources in GCP. For more information about the Google Cloud Client Library for Python, review https://cloud.google.com/compute/docs/tutorials/python-guide.

When you run `python import.py` and dry run the Terraform JSON with the separated IAM configuration, you get a message that you do not have to make any changes. You successfully imported the existing IAM resources into their separate configuration and state!

```
$ python main.py

$ terraform plan
No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no
       differences, so no changes are needed.

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

Sometimes, your dry run indicates drift between the active resource state and the separated configuration. Your copied configuration does not match the active state of the resource. The differences often come from someone changing the active state of an infrastructure resource during a manual change or updates to the default value for an attribute. Update your separated infrastructure as code to *match the attributes of the active infrastructure resource*.

Import with and without a provisioning tool

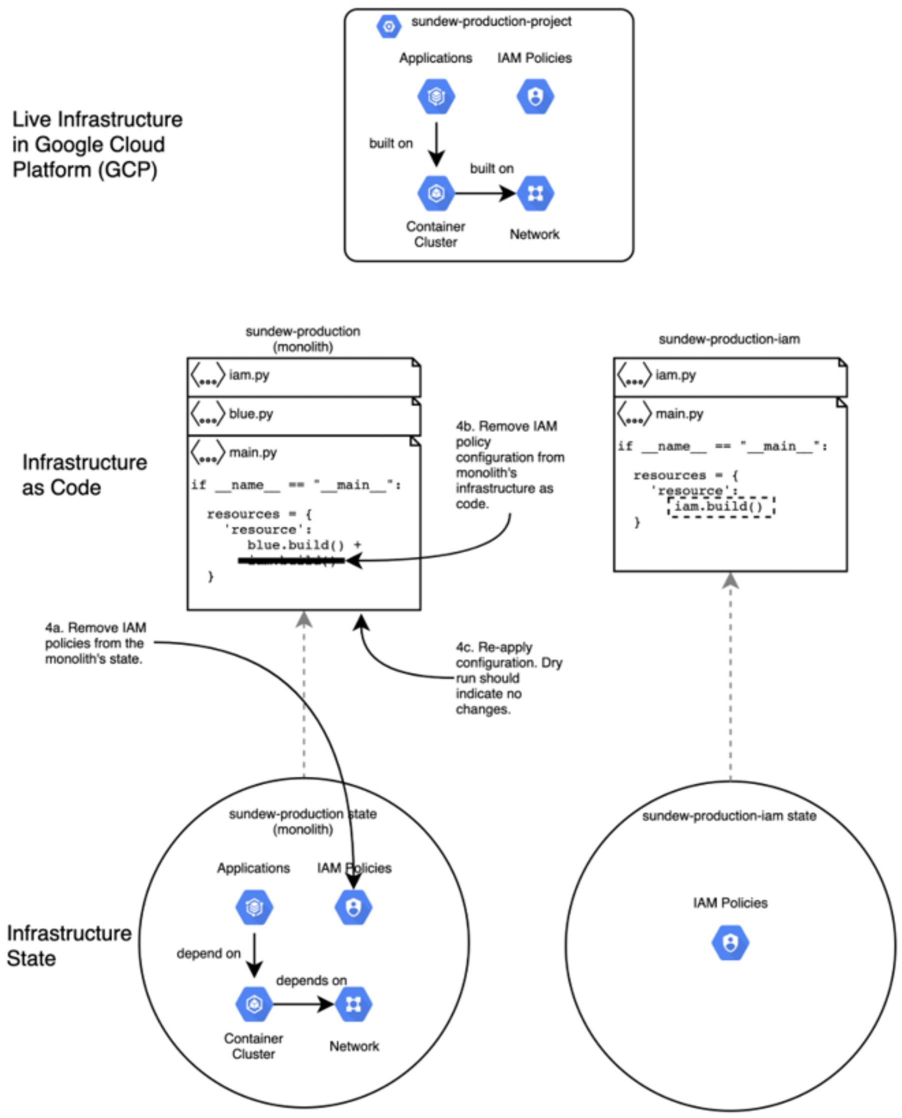
Many provisioning tools have a function for importing resources. For example, AWS CloudFormation uses the `resource import` command. The example uses Python wrapped around `terraform import` to move service accounts. Breaking down monolithic configuration will become tedious without it.

If you write infrastructure as code without a tool, you do not need a direct import capability. Instead, you need logic to check if the resources exist. The sundew service accounts and role assignments can work without Terraform or infrastructure as code import capability:

1. **Call the GCP API to check if the sundew team's service accounts and role attachments exist.**
2. **If they do, check if the API response for the service account attributes matches your desired configuration. Update the service account as needed.**
3. **If they do not, create the service accounts and role attachments.**

REMOVE THE REFACTORED RESOURCES FROM THE MONOLITH

You managed to extract and move the sundew team's service accounts and role assignments to separate infrastructure as code. However, you don't want the resources to stay in the monolith. In figure 10.8, you remove the resources from the monolith's *state and configuration* before re-applying and updating your tool.


Live Infrastructure in Google Cloud Platform (GCP)
sundew-production-project
Applications
IAM Policies
built on
built on
Container Cluster
Network
Infrastructure as Code
sundew-production (monolith)
iam.py
blue.py
main.py
4b. Remove IAM policy configuration from monolith's infrastructure as code.
sundew-production-iam
iam.py
main.py
4a. Remove IAM policies from the monolith's state.
4c. Re-apply configuration. Dry run should indicate no changes.
Infrastructure State
sundew-production state (monolith)
Applications
IAM Policies
depend on
depends on
Container Cluster
Network
sundew-production-iam state
IAM Policies

```
if __name__ == "__main__":

  resources = {
    'resource':
      blue.build() +
      ~~iam.build()~~
  }
```

```
if __name__ == "__main__":

  resources = {
    'resource':
      iam.build()
  }
```

Figure 10.8. Remove the policies from the monolith’s state and configuration before applying the updates and completing the refactor.

This step helps maintain infrastructure as code hygiene. Remember from chapter 2 that our infrastructure as code should serve as the *source of truth*. You do not want to manage one resource with two sets of infrastructure as code. If they conflict, the two infrastructure as code definitions for the resource may affect dependencies and the configuration of the system.

You want the IAM policy directory to serve as the source of truth. Going forward, the sundew team needs to declare changes to their IAM policy in the separate directory and not in the monolith. To avoid confusion, let's remove the IAM resources from the infrastrastructure as code monolith.

To start, you must remove the sundew IAM resources from Terraform state, represented in a JSON file. Terraform includes a state removal command that you can use to take out portions of the JSON based on the resource identifier. Listing 10.8 uses Python code to wrap around the Terraform command. The code allows you to pass any resource type and identifier you want to remove from the infrastructure state.

Listing 10.8 File remove.py removes resources from the monolith's state

```
from sundew_production_iam import iam    #A
import subprocess

def check_state_remove_status(ret, err):    #D
   return ret != 0 \    #D
       and 'No matching objects found' not in str(err)    #D

def state_remove(resource_type, resource_identifier):    #C
   command = ['terraform', 'state', 'rm', '-no-color',    #C
              f'{resource_type}.{resource_identifier}']    #C
   return _terraform(command)    #C

def _terraform(command):    #E
   process = subprocess.Popen(
       command,
       stdout=subprocess.PIPE,
       stderr=subprocess.PIPE)
   stdout, stderr = process.communicate()
   return process.returncode, stdout, stderr

if __name__ == "__main__":
   sundew_iam = iam.users    #A
   for user in iam.users:    #B
       ret, _, err = state_remove(    #F
           iam.TERRAFORM_GCP_SERVICE_ACCOUNT_TYPE,    #F
           iam.get_user_id(user))    #F
       if check_state_remove_status(ret, err):    #G
           print(f'remove service account from state failed: {err}')    #G
       ret, _, err = state_remove(    #H
           iam.TERRAFORM_GCP_ROLE_ASSIGNMENT_TYPE,    #H
           iam.get_user_id(user))    #H
       if check_state_remove_status(ret, err):    #G
           print(f'remove role assignment from state failed: {err}')    #G
```

#A Retrieve the list of sundew users from iam.py in sundew_production_iam. Referencing the variable from the separated infrastructure as code allows you to run the removal automation for future refactoring efforts.
#B For each user in sundew_production_iam, remove their service account and role assignment from the monolith's state.
#C Create a method that wraps around Terraform's state removal command. The command passes the resource type, such as service account and identifier to remove.

#D If the removal failed and did not already remove the resource, output the error.
#E Open a subprocess that runs the Terraform command to remove the resource from state.
#F Remove the GCP service account from the monolith's Terraform state based on its user identifier.
#G Check that the subprocess's Terraform command removed the resource from the monolith's state successfully.
#H Remove the GCP role assignment from the monolith's Terraform state based on its user identifier.

Do not run `python remove.py` yet! Your monolith still contains a definition of the IAM policies. Open your monolithic infrastructure as code's `main.py`. Remove the code that builds the IAM service accounts and role assignments for the sundew team.

Listing 10.9 Remove the IAM policies from the monolith's code

```
import blue    #B
import json    #B

if __name__ == "__main__":
   resources = {
       'resource': blue.build()    #A
   }

   with open('main.tf.json', 'w') as outfile:    #C
       json.dump(resources, outfile,    #C
                 sort_keys=True, indent=4)    #C
```

#A Remove the code to build the IAM policies within the monolith and leave the other resources.
#B Remove the import for the IAM policies.
#C Write the configuration out to a JSON file to be executed by Terraform later. The configuration does not include the IAM policies.

You can now update your monolith. First, `python remove.py` to delete the IAM resources from the monolith's state.

```
$ python remove.py
```

This step signals that your monolith no longer serves as the source of truth for the IAM policies and service accounts. You *do not* delete the IAM resources! You can imagine this as handing over ownership of the IAM resources to the new infrastructure as code in a separate folder.

In your terminal, you can finally update the monolith. Generate a new Terraform JSON without the IAM policies, and apply the updates, you should not have any changes!

```
$ python main.py

$ terraform apply
google_service_account.blue: Refreshing state...
google_compute_network.blue: Refreshing state...
google_compute_subnetwork.blue: Refreshing state...
google_container_cluster.blue: Refreshing state...
google_container_node_pool.blue: Refreshing state…

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no
       differences, so no changes are needed.

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

If your dry run *includes* a resource you refactored, you know that you did not remove it from the monolith's state or configuration. You need to examine the resources and identify whether or not to remove them manually.

10.2.2 Refactor resources with dependencies

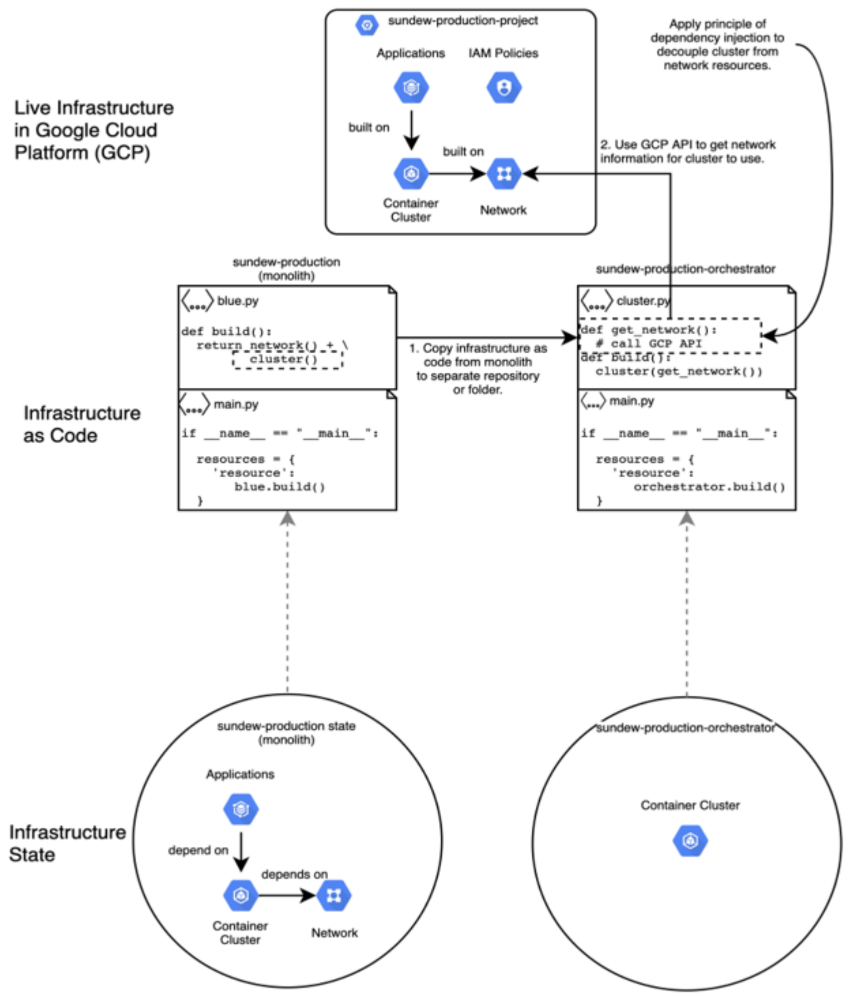
You can now work on the lower-level infrastructure resources with dependencies, such as the sundew team's container orchestrator. The sundew team asks you to avoid creating a new orchestrator and destroying the old one since they do not want to disrupt applications. You need to refactor and extract the low-level container orchestrator in place.

Start copying the container configuration out of the monolith, repeating the same process you used for refactoring the IAM service accounts and roles. You create a separate folder labeled `sundew_production_orchestrator`.

```
$ mkdir -p sundew_production_orchestrator
```

You select and copy the method to create the cluster into `sundew_production_orchestrator/cluster.py`. However, you have a problem. The container orchestrator *needs the network and subnet names*! How do you get the name of the network and subnet when the container orchestrator cannot reference the monolith?

Figure 10.9 implements dependency injection with an existing monolith using the infrastructure provider's API as the abstraction layer. The infrastructure as code to create the cluster calls the GCP API to get network information. You pass the network ID to the cluster to use.


sundew-production-project
Applications
IAM Policies
built on
built on
Container Cluster
Network
Apply principle of dependency injection to decouple cluster from network resources.
Live Infrastructure in Google Cloud Platform (GCP)
2. Use GCP API to get network information for cluster to use.
sundew-production (monolith)
sundew-production-orchestrator
blue.py
cluster.py
1. Copy infrastructure as code from monolith to separate repository or folder.
main.py
main.py
Infrastructure as Code
sundew-production state (monolith)
Applications
depend on
depends on
Container Cluster
Network
Infrastructure State
sundew-production-orchestrator
Container Cluster

```
def build():
  return network() + \
      cluster()
```

```
if __name__ == "__main__":

  resources = {
    'resource':
      blue.build()
  }
```

```
def get_network():
  # call GCP API
def build():
  cluster(get_network())
```

```
if __name__ == "__main__":

  resources = {
    'resource':
       orchestrator.build()
  }
```

Figure 10.9. Copy the infrastructure and add new methods to call the GCP API and get the network ID for the cluster.

A monolith passes the dependency explicitly between resources. When you create a new folder, your separated resources need information about its low-level dependencies. Recall that you can decouple infrastructure modules with the *dependency injection* (previously in Chapter 4). A high-level module calls an abstraction layer to get identifiers for low-level dependencies.

When you start refactoring resources with dependencies, you must implement an interface for dependency injection. In the sundew team’s code, update sundew_production_orchestrator/cluster.py to use Google Cloud Client Library and retrieve the subnet and network names for the cluster configuration.

> **NOTE** A number of dependencies, variables, and imports have been removed from this code listing for additional clarity. Refer to the book's code repository at https://github.com/joatmon08/manning-book/tree/main/ch10/s03/s02 for the full example.

Listing 10.10 Use dependency inversion for network name in the cluster .

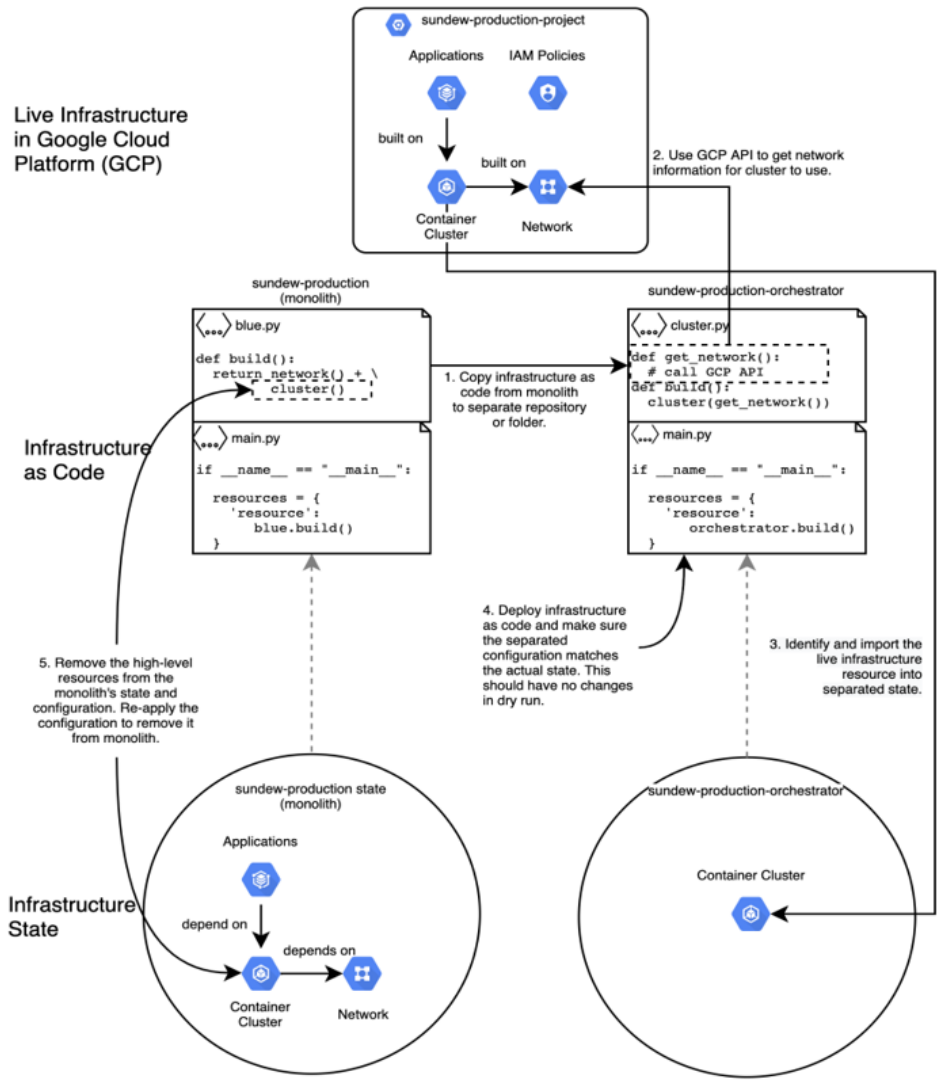
```
import googleapiclient.discovery    #A
    #G

def _get_network_from_gcp():    #H
   service = googleapiclient.discovery.build(    #A
       'compute', 'v1')    #A
   result = service.subnetworks().list(    #B
       project=PROJECT,    #B
       region=REGION,    #B
       filter=f'name:"{TEAM}-{ENVIRONMENT}-*"').execute()    #B
   subnetworks = result['items'] if 'items' in result else None
   if len(subnetworks) != 1:    #C
       print("Network not found")    #C
       exit(1)    #C
   return subnetworks[0]['network'].split('/')[-1], \    #D
       subnetworks[0]['name']    #D

def cluster(name=cluster_name,    #G
           node_name=cluster_nodes,
           service_account=cluster_service_account,
           region=REGION):
   network, subnet = _get_network_from_gcp()    #E
   return [
       {
           'google_container_cluster': {    #I
               VERSION: [
                   {
                       'name': name,
                       'network': network,    #F
                       'subnetwork': subnet    #F
                   }
               ]
           },
           'google_container_node_pool': {    #I
               VERSION: [
                   {
                       'cluster':
                       '${google_container_cluster.' +
                           f'{VERSION}' + '.name}'
                   }
               ]
           },
           'google_service_account': {    #I
               VERSION: [
                   {
                       'account_id': service_account,
                       'display_name': service_account
                   }
```

```
                ]
            }
        }
    ]
```

#A Set up access to the GCP API using Google Cloud Client Library for Python.
#B Query the GCP API for a list of subnetworks with names that start with “sundew-production.”
#C Throw an error if the GCP API did not find the subnetwork.
#D Return the network name and subnetwork name.
#E Apply the dependency inversion principle and call the GCP API to retrieve the network and subnet names.
#F Use the network and subnet names to update the container cluster.
#G A number of dependencies, variables, and imports have been removed from the code listing for additional clarity. Refer to the book’s code repository for the full example.
#H Create a method that retrieves the network information from GCP and implements dependency injection.
#I Create the Google container cluster, node pool, and service account using a Terraform resource.

When refactoring an infrastructure resource with dependencies, you must implement dependency injection to retrieve the low-level resource attributes. The example uses an infrastructure provider’s API, but you can use any abstraction layer you choose. An infrastructure provider’s API often provides the most straightforward abstraction. You can use it to avoid implementing your own.

After copying and updating the container cluster to reference network and subnet names from the GCP API, you repeat the refactoring workflow in Figure 10.10. You import the live infrastructure resource into `sundew_production_orchestrator`, apply the separate configuration, check for any drift between the active state and the infrastructure as code, and remove the resource’s configuration and reference in the monolith’s state.


Live Infrastructure in Google Cloud Platform (GCP)
sundew-production-project
Applications
IAM Policies
built on
built on
Container Cluster
Network
2. Use GCP API to get network information for cluster to use.
Infrastructure as Code
sundew-production (monolith)
blue.py
main.py
1. Copy infrastructure as code from monolith to separate repository or folder.
sundew-production-orchestrator
cluster.py
main.py
4. Deploy infrastructure as code and make sure the separated configuration matches the actual state. This should have no changes in dry run.
3. Identify and import the live infrastructure resource into separated state.
5. Remove the high-level resources from the monolith's state and configuration. Re-apply the configuration to remove it from monolith.
Infrastructure State
sundew-production state (monolith)
Applications
depend on
depends on
Container Cluster
Network
sundew-production-orchestrator
Container Cluster

```
def build():
  return network() + \
    cluster()
```

```
if __name__ == "__main__":

  resources = {
    'resource':
      blue.build()
  }
```

```
def get_network():
  # call GCP API
def build():
  cluster(get_network())
```

```
if __name__ == "__main__":

  resources = {
    'resource':
      orchestrator.build()
  }
```

Figure 10.10. Refactor higher-level resources to get low-level identifiers with the GCP API before continuing to refactor low-level resources.

The main difference between refactoring a high-level resource versus a lower-level resource out of a monolith involves the implementation of dependency injection. You can choose the type of dependency injection you want to use, such as the infrastructure provider's API, module outputs, or infrastructure state. Note that you might need to change the monolithic infrastructure as code to output the attributes if you do not use the infrastructure provider's API.

Otherwise, ensure you apply idempotency by re-running your infrastructure as code after refactor. You want to avoid affecting the active resources and isolate all changes to infrastructure as code. If your dry run reflects changes, you must fix the drift between your refactored code before moving forward with other resources.

10.2.3 Repeat refactoring workflow

After you extract the IAM service accounts and roles and the container orchestrator, you can continue to break down the sundew system's monolithic infrastructure as code configuration. The workflow in Figure 10.11 summarizes the general pattern for breaking down monolithic infrastructure as code. You identify which resources depend on each other, extract their configuration, and update their dependencies to use dependency injection.

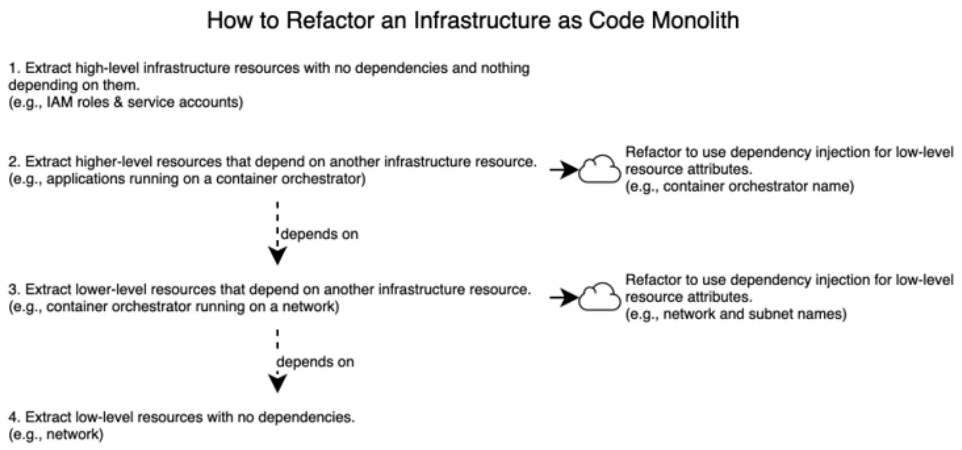

How to Refactor an Infrastructure as Code Monolith
1. Extract high-level infrastructure resources with no dependencies and nothing depending on them. (e.g., IAM roles & service accounts)
2. Extract higher-level resources that depend on another infrastructure resource. (e.g., applications running on a container orchestrator)
Refactor to use dependency injection for low-level resource attributes. (e.g., container orchestrator name)
depends on
3. Extract lower-level resources that depend on another infrastructure resource. (e.g., container orchestrator running on a network)
Refactor to use dependency injection for low-level resource attributes. (e.g., network and subnet names)
depends on
4. Extract low-level resources with no dependencies. (e.g., network)

Figure 10.11. The workflow for refactoring an infrastructure as code monolith starts by identifying high-level resources with no dependencies

Identify the high-level infrastructure resources that do not depend on anything or have anything depending on them. I use the high-level resources to test the workflow of copying, separating, importing, and deleting them from the monolith. Next, I identify the higher-level resources that depend on other resources. During the copying, I refactor them to reference attributes through dependency injection. I identify and repeat the process through the system, eventually concluding with the lowest-level resources that do not have any dependencies.

Configuration management

While this chapter focused primarily on infrastructure as code provisioning tools, configuration management can also turn into a monolith of automation and result in the same challenges, like taking a long time to run or conflicting changes to parts of the configuration. You can apply a similar refactoring workflow to monolithic configuration management!

1. **Extract the most independent parts of the automation with no dependencies and separate them into a module.**
2. **Run the configuration manager and make sure you did not change your resource state.**
3. **Identify configuration that depends on the outputs or existence of lower-level automation. Extract them and apply dependency injection to retrieve any required values for the configuration.**
4. **Run the configuration manager and make sure you did not change your resource state.**
5. **Repeat the process until you effectively reach your configuration manager's "first step."**

As you refactor infrastructure as code monoliths, identify ways to decouple the resources from each other. I find refactoring a challenge and rarely without some failures and mistakes. Isolating individual components and carefully testing them will help identify problems and minimize disruption to the system. If I do encounter failures, I use the techniques in Chapter 11 to fix them.

Exercise 10.1

Given:

```
if __name__ == "__main__":
 zones = ['us-west1-a', 'us-west1-b', 'us-west1-c']
 project.build()
 network.build(project)
 for zone in zones:
  subnet.build(project, network, zone)
 database.build(project, network)
 for zone in zones:
  server.build(project, network, zone)
 load_balancer.build(project, network)
 dns.build()
```

What order and grouping of resources would you use to refactor and break down the monolith?

A. **DNS, load balancer, servers, database, network + subnets, project**
B. **Load balancer + DNS, database, servers, network + subnets, project**
C. **Project, network + subnets, servers, database, load balancer + DNS**
D. **Database, load balancer + DNS, servers, network + subnets, project**

Find answers and explanations at the end of the chapter

10.3 Exercises and Solutions

Exercise 10.1

Given:

```
if __name__ == "__main__":
 zones = ['us-west1-a', 'us-west1-b', 'us-west1-c']
 project.build()
 network.build(project)
 for zone in zones:
  subnet.build(project, network, zone)
 database.build(project, network)
 for zone in zones:
  server.build(project, network, zone)
 load_balancer.build(project, network)
 dns.build()
```

What order and grouping of resources would you use to refactor and break down the monolith?

A. DNS, load balancer, servers, database, network + subnets, project
B. Load balancer + DNS, database, servers, network + subnets, project
C. Project, network + subnets, servers, database, load balancer + DNS
D. Database, load balancer + DNS, servers, network + subnets, project

Answer:

The order and grouping that mitigates the risk of refactor starts with DNS and proceeds with the load balancer, queue, database, servers, network and subnets, and project (A). You want to start with DNS as the highest-level dependency and evolve it separately from the load balancer, since the load balancer requires project and network attributes. Refactor the next resources as follows: the servers, database, network, and project.

You refactor the servers before the database because servers do not manage data directly but depend on the database. If you do not feel confident that you can refactor the database, at least you've moved the servers out of the monolith! You can always leave the database in the monolith with the network and project. Refactoring the bulk of resources out of the monolith sufficiently decouples the system to scale

10.4 Summary

- Refactoring infrastructure as code involves restructuring configuration or code without impacting existing infrastructure resources.
- Refactoring resolves technical debt, a metaphor to describe the cost of changing code.
- A rolling update changes similar infrastructure resources one by one and tests each resource before moving to the next one.
- Rolling updates allow you to implement and troubleshoot changes incrementally.
- Feature flags (also known as feature toggles) enable or disable infrastructure resources, dependencies, or attributes.
- Apply feature flags to test, stage, and hide changes before applying them to production.
- Define feature flags in one place (such as a file or configuration manager) to identify their values at a glance.
- Remove feature flags when you do not need them anymore.
- Monolithic infrastructure as code happens when you define all of your infrastructure resources in one place and removing one resource causes the entire configuration to fail.
- Refactoring a resource out of a monolith involves separating and copying the configuration into a new directory or repository, importing them into a new separate state, and removing the resources from the monolithic configuration and state.
- If your resource depends on another resource, update your separated resource configuration to use dependency injection and retrieve identifiers from an infrastructure provider API.
- Breaking down a monolith starts by refactoring high-level resources or configurations with no dependencies, then resources or configurations with dependencies, and concluding with low-level resources or configurations with no dependencies.

11 Fixing failures

This chapter covers

- **Determining how to roll forward failed changes to restore functionality**
- **Organizing an approach for infrastructure as code troubleshooting**
- **Categorizing repairs for failed changes**

We took many chapters to discuss writing and collaborating on infrastructure as code. All of the practices and principles you learned for infrastructure as code accumulates to the crucial moment when you push a change, it causes your system to fail, and you need to roll it back! However, infrastructure as code doesn't support "rollback." You do not fully revert infrastructure as code changes! What does it mean to fix failures if you don't roll them back?

This chapter focuses on fixing failed changes from infrastructure as code. First, we'll discuss what it means to "revert" infrastructure as code changes by rolling forward. Then, you'll learn workflows for troubleshooting and fixing the failed change. While the techniques in this chapter might not apply to every scenario you'll encounter in your system, they establish a broad set of practices you can use to start repairing infrastructure as code failures.

Troubleshooting and site reliability engineering

I do not dive too deeply into the process and principles of troubleshooting failed systems in this book. Most of the discussion around troubleshooting will center around how to manage it in the context of infrastructure as code. For more information on troubleshooting and building reliable systems, I recommend *Site Reliability Engineering* by the team at Google (https://sre.google/sre-book/effective-troubleshooting/).

As a general rule, *prioritize stabilization and restoration of functionality* for services and customers before you debug further for a root cause (the issue that led to the problem). A temporary bandage provides you the opportunity to troubleshoot and implement a longer-term fix for the system.

11.1 Restoring functionality

Imagine you work for a company called Cool Caps for Keys. They create custom keyboard caps and connect customers with artists to design the caps. As the security engineer, you need to narrow down the access control for applications and users across Google Cloud Platform (GCP) projects.

You copy the Google Cloud SQL database configuration and update the access control to implement least-privilege access for team members and applications. You choose the policies required for different applications to use infrastructure and verify the applications still work.

Next, you talk to the promotions team. Their application access the database directly using a database username and password. Direct access to the database means that you can remove the policy for "roles/cloudsql.admin" from the promotion application's service account. In figure 11.1, you remove the policy, test the changes, confirm with the promotions team that the change did not affect their application in their testing environment, and push it to production.

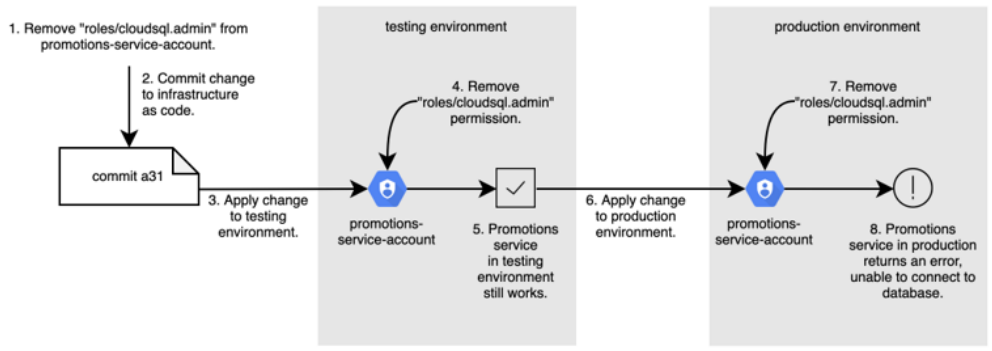

1. Remove "roles/cloudsql.admin" from promotions-service-account.
2. Commit change to infrastructure as code.
commit a31
3. Apply change to testing environment.
testing environment
4. Remove "roles/cloudsql.admin" permission.
promotions-service-account
5. Promotions service in testing environment still works.
6. Apply change to production environment.
production environment
7. Remove "roles/cloudsql.admin" permission.
promotions-service-account
8. Promotions service in production returns an error, unable to connect to database.

Figure 11.1. After removing the administrative database access for the promotion service, you discover that the change broke the service's ability to access the database.

An hour later, the promotions team tells you that their application keeps throwing an error that it can't access the database! You suspect that your change might have introduced a problem. While you could start digging around for the problem, you prioritize fixing the promotion service so it can access the database before investigating further.

11.1.1 Rolling forward to revert changes

We need to fix the service so users can make requests to our system. However, we cannot simply return the system to a previously working state. Infrastructure as code prioritizes immutability, which means any changes to the system like *reverted* changes must create new resources!

For example, let's fix the promotion service for Cool Caps for Keys by reverting the change and adding the role to the service account. In figure 11.2, you revert the commit and add "roles/cloudsql.admin" back to the service account. Then, you push the changes to your testing and production environments.

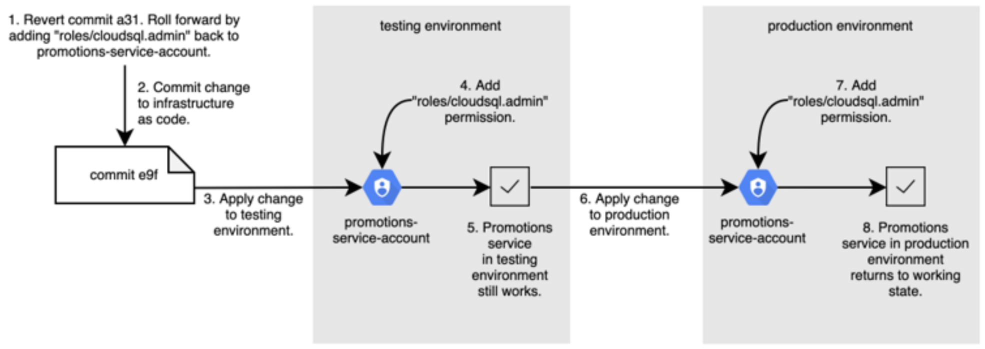

1. Revert commit a31. Roll forward by adding "roles/cloudsql.admin" back to promotions-service-account.
2. Commit change to infrastructure as code.
commit e9f
3. Apply change to testing environment.
testing environment
4. Add "roles/cloudsql.admin" permission.
promotions-service-account
5. Promotions service in testing environment still works.
6. Apply change to production environment.
production environment
7. Add "roles/cloudsql.admin" permission.
promotions-service-account
8. Promotions service in production environment returns to working state.

Figure 11.2. You add the administrative database role for the promotion service to roll-forward the system to a working state.

You revert the commit and push *forward* the changes to the testing and production environments. You ***roll forward*** infrastructure as code because it uses immutability to return the system to a working state.

> **DEFINITION** The practice of rolling forward infrastructure as code reverts changes to a system and uses immutability to return the system to a working state.

Rollback implies that we return infrastructure to a previous state. In reality, the immutability of infrastructure as code means that you create a *new* state any time you make changes. You cannot fully restore the state of infrastructure back to its previous state. Sometimes, you can't actually restore the infrastructure to a previous state because your change has a large blast radius.

Let’s revert your changes to the service account and roll forward changes to add back the permission. First, check your commit history because version control keeps track of all the changes you make. The commit prefixed with “a31” included your removal of “roles/cloudsql.admin”.

```
$ git log --oneline -2
a3119fc (HEAD -> main) Remove database admin access from promotions
6f84f5d Add database admin to promotions service
```

Applying the GitOps practices from Chapter 7, you want to avoid making manual, break-glass changes. Instead, you favor operational changes through infrastructure as code! You revert the commit to push updates to restore the promotion service to a working state.

```
$ git revert a3119fc
```

You push the commit, and the pipeline adds the role back to the service account. After you roll forward, the application works again. You successfully returned the infrastructure state to a working state. However, you never achieve a full “restore” of state. Instead, you rolled out a new state that matched the previous working state.

Rolling back infrastructure as code often means rolling forward changes to the infrastructure state. You use `git revert` as a forward-moving undo to preserve immutability and roll-forward undo updates to infrastructure.

Configuration management

Configuration management does not prioritize immutability but still rolls forward reverted changes to a server or resource. For example, imagine you install a package with version 3.0.0 and need to revert to 2.0.0. Your configuration management tool may choose to uninstall the new version and reinstall the old version. You do not restore the package and its configuration to its previous state. You just restore the server to a new working state with an older package.

11.1.2 Rolling forward for new changes

The benefit of taking a roll-forward mentality means that you expand your troubleshooting approach. In the example, we reverted a broken commit and restored functionality to the promotion service by matching the new state to a prior working one. However, you will find some circumstances in which reverting a commit would not fix your system or make everything worse! Instead, you can roll forward *new changes* and restore functionality.

Rather than try to fix the application, you create a new environment with the change and a new promotions service. In figure 11.3, you start a canary deployment technique from Chapter 9 to gradually increase the traffic to fully restore the application. You disable the failed environment once all requests go to the new service instance for debugging.

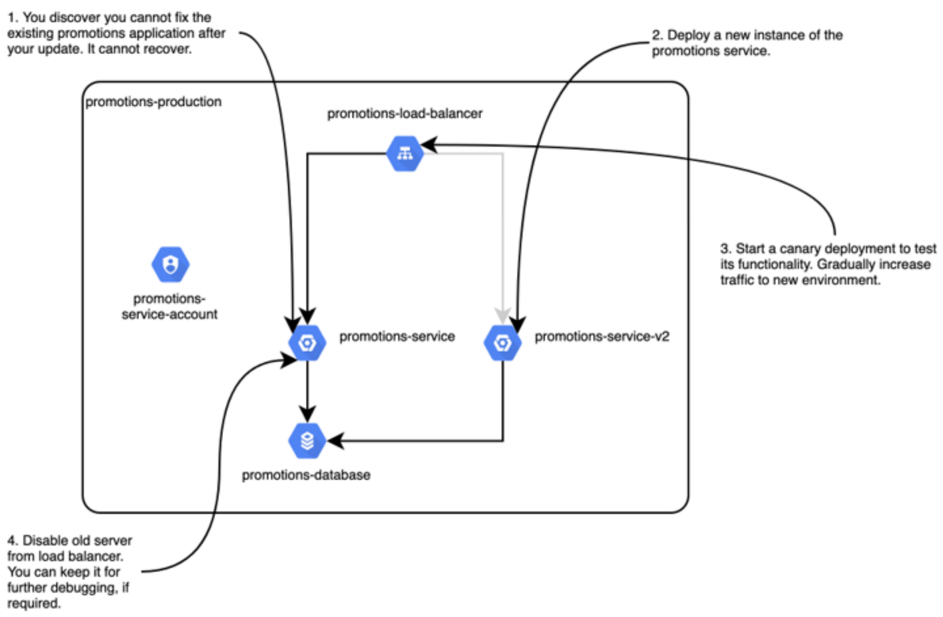

1. You discover you cannot fix the existing promotions application after your update. It cannot recover.
2. Deploy a new instance of the promotions service.
promotions-production
promotions-load-balancer
3. Start a canary deployment to test its functionality. Gradually increase traffic to new environment.
promotions-service-account
promotions-service
promotions-service-v2
promotions-database
4. Disable old server from load balancer. You can keep it for further debugging, if required.

Figure 11.3. When you cannot recover the promotions application, you can use a canary deployment to cut over traffic to a new instance and restore the system.

Infrastructure as code allows you to reproduce environments with less effort. Furthermore, conforming to immutability means that you already have a pattern of creating new environments for changes. The combination of the two principles helps mitigate higher-risk changes with a larger blast radius.

Use cases that involve data or completely irrecoverable resources cannot roll forward to a prior state. You could corrupt application data or affect other infrastructure while detangling a cascading failure. Rather than roll forward to revert, you can roll forward and implement new changes by applying the change techniques from Chapter 9.

You may also restore functionality with a combination of reverts and completely new changes. Expanding your roll-forward mentality to include new changes outside of undoing old ones offers a helpful alternative to restore functionality quickly and minimize disruption to other parts of your system.

11.2 Troubleshooting

You put a bandage on your system so the promotion team can still send out promotional offers for Cool Caps for Keys. However, you still need to secure the identity and access management of the application! Where do you start to find out why the promotion service failed when you removed administrative permission they shouldn't need?

Troubleshooting your infrastructure as code also follows specific patterns. Even in the most complex infrastructure systems, many failed changes from infrastructure as code usually come from three causes: drift, dependencies, or differences. Examining your configuration for any of these causes helps you identify the problem and a potential fix.

11.2.1 Check for drift

Many broken infrastructure changes stem from configuration drift between configuration and resource state. In figure 11.4, you start by checking for drift. Make sure the infrastructure as code for the service account matches the state of the service account in GCP.

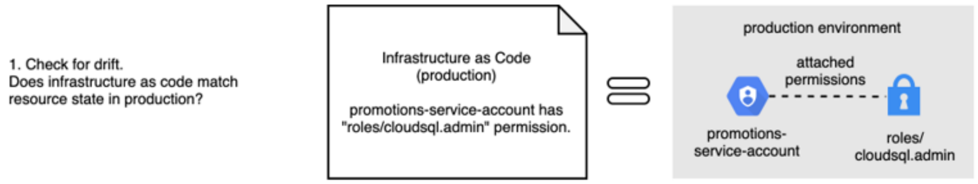

1. Check for drift.
Does infrastructure as code match resource state in production?
Infrastructure as Code (production)
promotions-service-account has "roles/cloudsql.admin" permission.
production environment
attached permissions
promotions-service-account
roles/cloudsql.admin

Figure 11.4. Start by checking for drift between infrastructure as code and state.

Checking for drift between code and state ensures that you eliminate any failures due to differences between the two. Differences between code and state can introduce unexpected problems. Removing those differences ensures your change behavior works as expected.

In the case of Cool Caps for Keys, you review the permissions for the promotions service account that you define in infrastructure as code. Listing 11.1 outlines the infrastructure as code that defines the service and roles.

Listing 11.1 Promotion service account with database admin permission

```
from os import environ
import database    #C
import iam    #B
import network    #D
import server    #E
import json
import os

SERVICE = 'promotions'
ENVIRONMENT = 'prod'
REGION = 'us-central1'
ZONE = 'us-central1-a'
PROJECT = os.environ['CLOUDSDK_CORE_PROJECT']
role = 'roles/cloudsql.admin'    #A

if __name__ == "__main__":
   resources = {    #F
       'resource':
       network.Module(SERVICE, ENVIRONMENT, REGION).build() +    #D
       iam.Module(SERVICE, ENVIRONMENT, REGION, PROJECT,    #B
                  role).build() +    #A
       database.Module(SERVICE, ENVIRONMENT, REGION).build() +    #C
       server.Module(SERVICE, ENVIRONMENT, ZONE).build()    #E
   }

   with open('main.tf.json', 'w') as outfile:    #G
       json.dump(resources, outfile,    #G
                 sort_keys=True, indent=4)    #G
```

#A Promotion service account should have permission for the "cloudsql.admin" role to access the database.
#B Import the Google service account module and create the configuration with the permissions.
#C Import the database module to build the Google Cloud SQL database.
#D Import the network module to build the Google network and subnetwork.
#E Import the server module to build the Google compute instance.
#F Use the module to create the JSON configuration for the database, network, service account, and server.
#G Write it out to a JSON file to be executed by Terraform later.

AWS equivalent

The AWS equivalent of the GCP Cloud SQL administrator permission is similar to "AmazonRDSFullAccess."

Then, match the code to the promotion application's service account permissions in Google Cloud Platform (GCP). The service account only has "roles/cloudsql.admin" permissions consistent with your infrastructure as code.

```
$ gcloud projects get-iam-policy $CLOUDSDK_CORE_PROJECT
bindings:
- members:
  - serviceAccount:promotions-prod@infrastructure-as-code-book.iam.gserviceaccount.com
  role: roles/cloudsql.admin
version: 1
```

If you find configuration drift between infrastructure as code and the active resource state, you can further investigate whether or not it affects system functionality. You may choose to eliminate some of the drift to ensure it doesn't contribute to the root cause. However, just because you detect some drift does not mean that it breaks your system! Some drift may have nothing to do with the failure.

11.2.2 Check for dependencies

If you determine drift does not contribute to the failure, you can check for resources that depend on your updated one. In figure 11.5, you start graphing which resources depend on the service account. In both the infrastructure as code and production environment, the server depends on the service account.

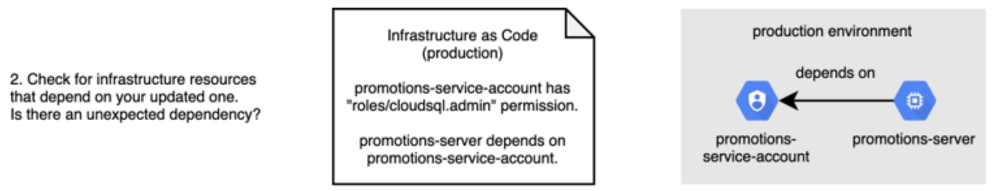

2. Check for infrastructure resources that depend on your updated one. Is there an unexpected dependency?
Infrastructure as Code (production)
promotions-service-account has "roles/cloudsql.admin" permission.
promotions-server depends on promotions-service-account.
production environment
depends on
promotions-service-account
promotions-server

Figure 11.5. Troubleshoot any resources that depend on the one you want to update.

You want to check that the expected dependencies match the actual. Unexpected dependencies disrupt change behavior. When you review the code, you verify that the service account's email gets passed to the server.

Listing 11.2 Promotion server depends on the promotion service account

```
class Module():    #C
   def __init__(self, service, environment,
                zone, machine_type='e2-micro'):
       self._name = f'{service}-{environment}'
       self._environment = environment
       self._zone = zone
       self._machine_type = machine_type

   def build(self):    #C
       return [
           {
               'google_compute_instance': {    #B
                   self._environment: {
                       'allow_stopping_for_update': True,
                       'boot_disk': [{
                           'initialize_params': [{
                               'image': 'ubuntu-1804-lts'
                           }]
                       }],
                       'machine_type': self._machine_type,
                       'name': self._name,
                       'zone': self._zone,
                       'network_interface': [{
                           'subnetwork':
                           '${google_compute_subnetwork.' +
                           f'{self._environment}' + '.name}',
                           'access_config': {
                               'network_tier': 'STANDARD'
                           }
                       }],
                       'service_account': [{    #A
                           'email': '${google_service_account.' +    #A
                           f'{self._environment}' + '.email}',    #A
                           'scopes': ['cloud-platform']    #A
                       }]    #A
                   }
               }
           }
       ]
```

#A The factory for the promotion application's server uses a service account to access GCP services.
#B Create the Google compute instance using a Terraform resource based on the name, address, region, and network.
#C Use the module to create the JSON configuration for the server.

However, the promotion team mentions that their application *directly* accesses the database using its IP address, username, and password. Why would the server need the service account if the application reads the database connection string from a file?

You realize this highlights a discrepancy. You ask the promotion team to show you the application code. The application configuration does not use the database IP address, username, or password!

After additional debugging with the promotion team, you discover that the promotion application connects to the database on `localhost`. The configuration uses Cloud SQL auth proxy (https://cloud.google.com/sql/docs/mysql/sql-proxy), which handles the connection and logs into the database! Therefore, the service account connected to the server needs database access.

Figure 11.6 shows that the promotions application accesses the database through a proxy. The proxy uses the service account to authenticate and access the database. The service account needs access to the database with a policy.

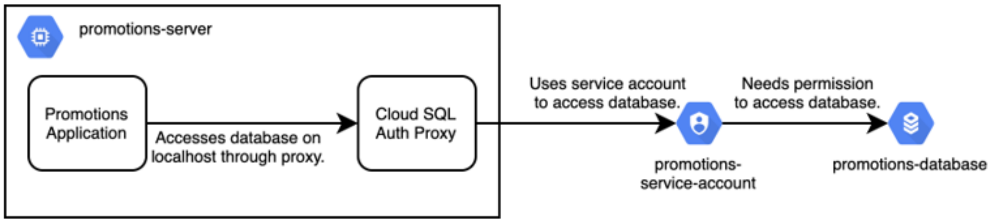

promotions-server
Promotions Application
Accesses database on localhost through proxy.
Cloud SQL Auth Proxy
Uses service account to access database.
promotions-service-account
Needs permission to access database.
promotions-database

Figure 11.6. The promotion application accesses the database through a proxy, which needs a service account with database permissions.

AWS equivalent

The AWS equivalent of the GCP Cloud SQL auth proxy is Amazon RDS Proxy. The proxy helps enforce database connections and avoids the need for database usernames and passwords in application code.

Congratulations, you discovered why the promotion application broke when you removed the service account! However, you get a little suspicious. Shouldn't you have found the same problem in the testing environment? After all, you tested the change in a testing environment, and the application did not break.

11.2.3 Check for differences in environments

Why did the change work in testing but not in production? You examine the promotion application in testing. The application *does not* connect to the database on `localhost`. Instead, it uses the database IP address, username, and password.

In figure 11.7, you explain to the application team that the production infrastructure as code uses Cloud SQL auth proxy, while the testing infrastructure as code directly calls the database. Both configurations use "roles/cloudsql.admin" permission.

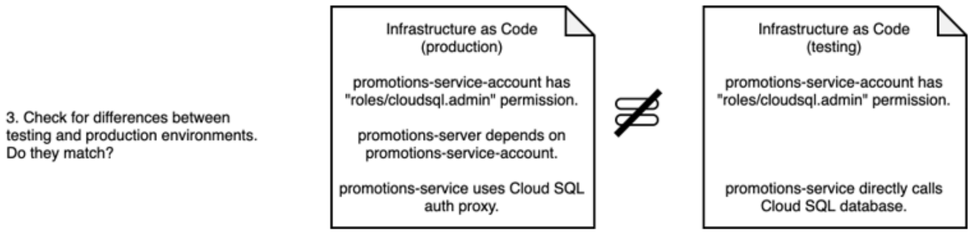

3. Check for differences between testing and production environments. Do they match?
Infrastructure as Code (production)
promotions-service-account has "roles/cloudsql.admin" permission.
promotions-server depends on promotions-service-account.
promotions-service uses Cloud SQL auth proxy.
≢
Infrastructure as Code (testing)
promotions-service-account has "roles/cloudsql.admin" permission.
promotions-service directly calls Cloud SQL database.

Figure 11.7. Check for differences between testing and production to reconcile any tested changes that fail.

After further discussion with the promotion team, you discover the team implemented an emergency change to secure production with Cloud SQL auth proxy. However, they did not have a chance to update the testing environment to match! The mismatch allowed your updates to succeed in the testing environment but fail in the production environment.

You want to keep testing and production as similar as possible. However, you cannot always reproduce production in testing environments. As a result, you will encounter failed changes due to discrepancies in both. Systematically identifying differences between testing and production environments helps highlight gaps in testing and change delivery.

While infrastructure as code should document all changes and configuration to your system, you might still discover a few surprises between infrastructure as code and environments. Figure 11.8 summarizes your structured approach to debugging failed changes in the promotion application's infrastructure as code. You check for drift, dependencies, and finally, differences between environments.

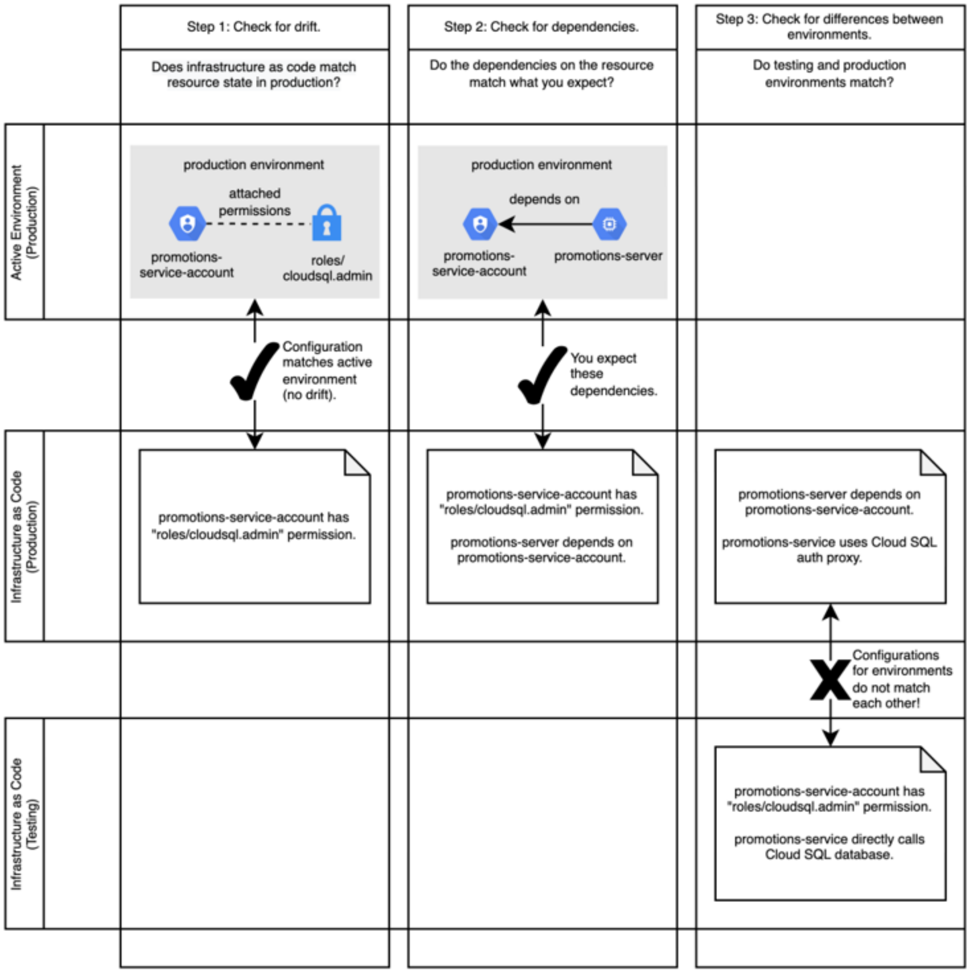

Step 1: Check for drift.
Does infrastructure as code match resource state in production?
Step 2: Check for dependencies.
Do the dependencies on the resource match what you expect?
Step 3: Check for differences between environments.
Do testing and production environments match?
Active Environment (Production)
Infrastructure as Code (Production)
Infrastructure as Code (Testing)
production environment
attached permissions
promotions-service-account
roles/cloudsql.admin
production environment
depends on
promotions-service-account
promotions-server
Configuration matches active environment (no drift).
You expect these dependencies.
promotions-service-account has "roles/cloudsql.admin" permission.
promotions-service-account has "roles/cloudsql.admin" permission.
promotions-server depends on promotions-service-account.
promotions-server depends on promotions-service-account.
promotions-service uses Cloud SQL auth proxy.
Configurations for environments do not match each other!
promotions-service-account has "roles/cloudsql.admin" permission.
promotions-service directly calls Cloud SQL database.

Figure 11.8. You use infrastructure as code to troubleshoot your broken change by checking for drift, unexpected dependencies, and differences between testing and production.

After determining a root cause, you can finally implement a long-term fix. You must now reconcile the difference between testing and production environment and revisit least-privilege access to secure the promotion application's service account.

Exercise 11.1

A team reports that that their application can no longer connect to another application. It worked last week but requests have failed since Monday. They have made no changes to their application and suspect it may be a problem with a firewall rule. Which steps can you take to troubleshoot the problem? (Choose all that apply.)

A. Log into the cloud provider and check the firewall rules for the application.
B. Deploy new infrastructure and applications to a "green" environment for testing.
C. Examine the changes in infrastructure as code for the application.
D. Compare the firewall rules in the cloud provider with infrastructure as code.
E. Edit firewall rules and allow all traffic between the applications.

Find answers and explanations at the end of the chapter.

11.3 Fixing

Your original task for Cool Caps for Keys involved updating the service account permissions for each application to ensure least-privilege access to services. You tried to remove database administrative access from the promotion application's service account but failed. After troubleshooting the issue, you can now fix the problem.

You might find yourself a bit impatient at this point! After all, you have not finished updating the access for other applications in Cool Caps for Keys. However, don't just go in and change everything at once! Pushing a batch of changes can make it difficult to debug the source of failure (previously referenced in Chapter 7). Your testing environment still doesn't match production, and you can still affect the promotion application if you make too many changes at once.

Throughout the book, I mention the process of making minor changes to minimize the blast radius of potential failure. Similarly, ***incremental fixes*** break down the changes you need to make to a system to prevent future failure.

DEFINITION Incremental fixes break down changes into smaller parts to gradually improve a system and prevent future failure.

Making minor configuration changes and gradually deploying them helps you recognize the first sign of trouble and stage your infrastructure as code for future success.

11.3.1 Reconcile drift

As I mentioned in Chapter 2, you need to reconcile any manual changes to the infrastructure state with infrastructure as code. In the situation that you find some drift, you need to address it first! Your system should not have too many break-glass changes if you prioritize the use of infrastructure as code.

Recall that the promotion application for Cool Caps for Keys implemented a break-glass change that resulted in a difference between testing and production environments. The production application uses Cloud SQL auth proxy to connect to the database, while the testing application directly connects to the database through IP address and password. You need to build a Cloud SQL auth proxy in the testing environment.

To start fixing drift, you need to reconstruct the current state of infrastructure in configuration. Figure 11.9 reconstructs the installation commands for Cloud SQL auth proxy based on the production server. Then, you add the commands to infrastructure as code and apply them to the testing environment.

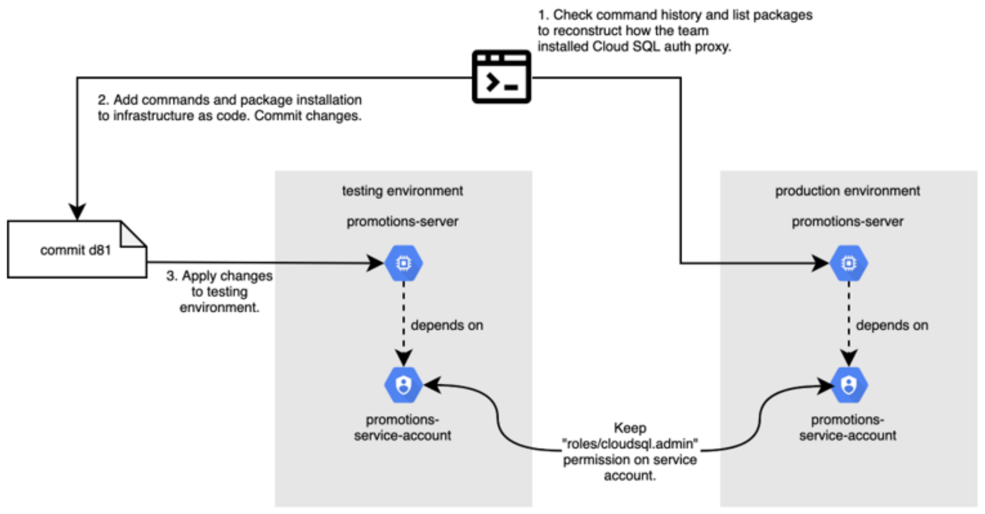

1. Check command history and list packages to reconstruct how the team installed Cloud SQL auth proxy.
2. Add commands and package installation to infrastructure as code. Commit changes.
commit d81
3. Apply changes to testing environment.
testing environment
promotions-server
depends on
promotions-service-account
production environment
promotions-server
depends on
promotions-service-account
Keep "roles/cloudsql.admin" permission on service account.

Figure 11.9. In testing, you need to install the Cloud SQL auth proxy package onto the promotion application's server to reconcile drift from the break-glass change.

In this example, the team did not add infrastructure as code for the manual change. As a result, you spend additional time rebuilding the installation of the Cloud SQL auth proxy. An out-of-band change such as the proxy caused a failed change, which takes even more time and effort to fix.

To help minimize some of these problems, use the process of migrating to infrastructure as code in chapter 2. Capturing the manual change as infrastructure as code helps minimize differences between environments and drift between infrastructure as code and actual state. If you need to reconstruct the state of infrastructure, remember that chapter 2 includes a high-level example of migrating existing infrastructure into infrastructure as code. However, you typically have to find or write a tool to transform the state to infrastructure as code.

Let's write the infrastructure as code to install the proxy. You checked the command history on the promotion application's server in production and reconstructed the installation of Cloud SQL auth proxy. Listing 11.3 automates the commands and installation process in a startup script for the promotion application's server.

Listing 11.3 Install Cloud SQL auth proxy in server startup script

```
class Module():
   def _startup_script(self):    #A
       proxy_download = 'https://dl.google.com/cloudsql/' + \    #D
           'cloud_sql_proxy.linux.amd64'    #D
       exec_start = '/usr/local/bin/cloud_sql_proxy ' + \    #E
           '-instances=${google_sql_database_instance.' + \    #E
           f'{self._environment}.connection_name}}=tcp:3306'    #E

       return f"""    #F
       #!/bin/bash
       wget {proxy_download} -O /usr/local/bin/cloud_sql_proxy
       chmod +x /usr/local/bin/cloud_sql_proxy

       cat << EOF > /usr/lib/systemd/system/cloudsqlproxy.service    #B
       [Install]
       WantedBy=multi-user.target

       [Unit]
       Description=Google Cloud Compute Engine SQL Proxy
       Requires=networking.service
       After=networking.service

       [Service]
       Type=simple
       WorkingDirectory=/usr/local/bin
       ExecStart={exec_start}
       Restart=always
       StandardOutput=journal
       User=root
       EOF

       systemctl daemon-reload    #B
       systemctl start cloudsqlproxy    #B
       """

   def build(self):    #G
       return [
           {
               'google_compute_instance': {    #C
                   self._environment: {    #C
                       'metadata_startup_script': self._startup_script()    #C
                   }    #C
               }    #C
           }
       ]
```

#A Create a startup script that reconstructs the manual installation commands for Cloud SQL auth proxy.
#B Configure systemd daemon to start and stop Cloud SQL auth proxy.
#C Add the startup script to the server. I omit other attributes for clarity.
#D Set a variable as the proxy download URL.
#E Set a variable that runs the Cloud SQL auth proxy binary on port 3306.
#F Return a shell script that installs the proxy and starts it up with the server.
#G Use the module to create the JSON configuration for the Google compute instance and include a startup script to install the proxy.

You *do not* update the service account with new permissions! In the spirit of incremental fixes, you want to avoid adding more changes to track as you push to production. You add the startup script to the promotion application's server and change the testing environment without more updates.

Startup script, configuration manager, or image builder?

I use the startup script field in this example to avoid introducing more syntax. Instead, you should implement the configuration of any new packages or processes with a *configuration manager* or *image builder*.

For example, the configuration manager would push the Cloud SQL auth proxy installation process to any server with the promotions application. Similarly, the image builder configures the proxy for each image you bake! Whenever you reference the image for the promotion application, you always have the proxy built into the server.

11.3.2 Reconcile differences in environments

While you updated your infrastructure as code to account for drift, you also need to make sure testing and production environments use your new infrastructure as code. For Cool Caps for Keys, you make sure the database connection works in the testing environment. Then, you ask the promotions team to update their application configuration to connect to the database through the proxy on `localhost`.

In figure 11.10, the promotions team pushes their application configuration to use Cloud SQL auth proxy into the testing environment, run their tests, and update production. You keep the "roles/cloudsql.admin" permission on the service account because the proxy needs it.

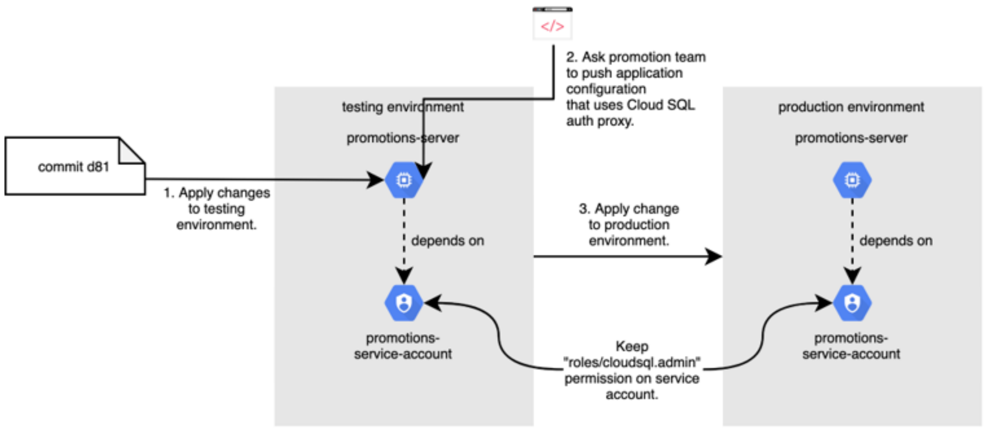

2. Ask promotion team to push application configuration that uses Cloud SQL auth proxy.
testing environment
promotions-server
commit d81
1. Apply changes to testing environment.
depends on
promotions-service-account
3. Apply change to production environment.
production environment
promotions-server
depends on
promotions-service-account
Keep "roles/cloudsql.admin" permission on service account.

Figure 11.10. You need to push your infrastructure as code changes onto the promotion application's server in testing and production environments.

The push recreates the production server with the new startup script. After additional end-to-end testing for the promotion application, you confirm that you successfully updated both testing and production environments.

Why start with reconciling drift before differences between production and testing environments? In this example, you opt to reconcile drift first because you will spend more time manually installing the packages in the testing environment! If you update your infrastructure as code and automate the package installation, you can ensure that the change works in the testing environment before pushing it to the production environment.

You might choose to reconcile testing and production environments first because you have a large amount of drift. In that case, match testing and production environments before fixing drift. You want an accurate testing environment before you implement your reconciliation changes.

Reconcile drift and differences in environments to help the next person make updates to the system. They do not have to worry about knowing the difference in configuration or manually configuring the proxy. The extra time you spend updating your infrastructure as code helps you avoid additional time debugging!

11.3.3 Implement the original change

Now that you've minimized the blast radius of potential failure by reconciling drift and updating your environments, you can finally push forward the original change! Your debugging and incremental fixes change your infrastructure. When you return to implementing the original change, you may need to adjust your code.

Let's finish the original change for the Cool Caps for Keys promotion application. Recall the security team asked you to remove administrative permissions from the service account. This process ensures least-privilege access and accounts for using Cloud SQL auth proxy.

You know that the service account must have database access because the application uses Cloud SQL auth proxy. Now, you try to figure out what kind of minimal access the application *should* use. The "roles/cloudsql.client" permission offers enough access for a service account to get a list of instances and connect to them.

In figure 11.11, you change the service account's permissions from administrative access to "roles/cloudsql.client". You push this change to the testing environment, verify that promotions still work, and deploy the "roles/cloudsql.client" permission to production.

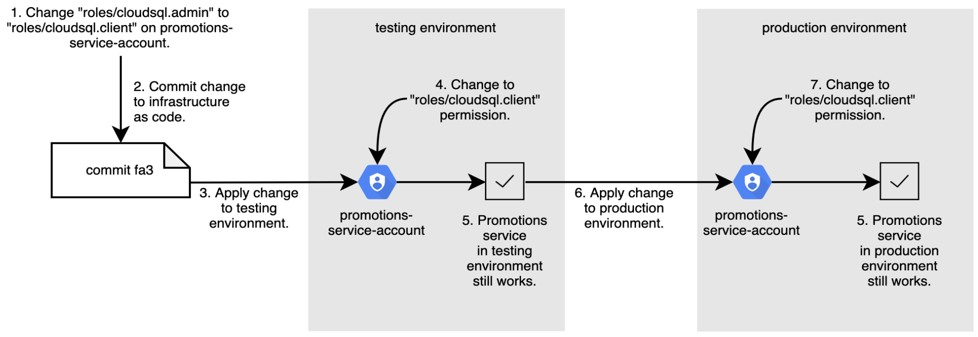

1. Change "roles/cloudsql.admin" to "roles/cloudsql.client" on promotions-service-account.
2. Commit change to infrastructure as code.
commit fa3
3. Apply change to testing environment.
testing environment
4. Change to "roles/cloudsql.client" permission.
promotions-service-account
5. Promotions service in testing environment still works.
6. Apply change to production environment.
production environment
7. Change to "roles/cloudsql.client" permission.
promotions-service-account
5. Promotions service in production environment still works.

Figure 11.11. You need to push your infrastructure as code changes onto the promotion application's server in testing and production environments.

You reconciled the difference between testing and production environments for the proxy. In theory, the testing environment should now catch any issues with your change. Any failed changes should now appear in the testing environment.

Let's change the permission for the service account in your infrastructure as code from "roles/cloudsql.admin" to "roles/cloudsql.client".

Listing 11.4 Change service account role to database client

```
from os import environ
import database
import iam
import network
import server
import json
import os

SERVICE = 'promotions'
ENVIRONMENT = 'prod'
REGION = 'us-central1'
ZONE = 'us-central1-a'
PROJECT = os.environ['CLOUDSDK_CORE_PROJECT']
role = 'roles/cloudsql.client'    #A

if __name__ == "__main__":
   resources = {    #B
       'resource':    #B
       network.Module(SERVICE, ENVIRONMENT, REGION).build() +    #B
       iam.Module(SERVICE, ENVIRONMENT, REGION, PROJECT,    #C
                  role).build() +    #B
       database.Module(SERVICE, ENVIRONMENT, REGION).build() +    #B
       server.Module(SERVICE, ENVIRONMENT, ZONE).build()    #B
   }    #B

   with open('main.tf.json', 'w') as outfile:
       json.dump(resources, outfile, sort_keys=True, indent=4)
```

#A Change the promotions service account role to client access, which allows connecting to the database instance.

#B Import the network, database, and server modules without changing the resources.
#C Import the service account and attach the "roles/cloudsql.client" role to its permissions.

You commit and apply the change. The testing environment applies the change and validates that the application still works! You confirm with the promotion team, who approves the production change.

The team promotes the new permissions change to production, runs end-to-end tests, and confirms the promotion application can access the database! After a few weeks of debugging and making changes, you can *finally* fix the other applications in Cool Caps for Keys.

Why dedicate an entire chapter with a single example of fixing failed changes? It represents the reality of fixing infrastructure as code. You want to resolve the failure as quickly as possible without making it worse.

Rolling *forward* helps restore the system's working state and minimize disruption to infrastructure resources. Then, you can work on troubleshooting the root cause. Many infrastructure failures come from drift, dependencies, or differences between testing and production environments. After you address these discrepancies, you implement your original change.

Learning the art of rolling forward infrastructure as code takes time and experience. While you could just log into the cloud provider console and make manual changes to get the system working, remember that the bandage will fall off very quickly and won't promote long-term system healing! Using infrastructure as code to track and fix the system incrementally minimizes the impact of the repairs and provides context for anyone else updating the system.

11.4 Exercises and Solutions

Exercise 11.1

A team reports that that their application can no longer connect to another application. It worked last week but requests have failed since Monday. They have made no changes to their application and suspect it may be a problem with a firewall rule. Which steps can you take to troubleshoot the problem? (Choose all that apply.)

A. Log into the cloud provider and check the firewall rules for the application.
B. Deploy new infrastructure and applications to a "green" environment for testing.
C. Examine the changes in infrastructure as code for the application.
D. Compare the firewall rules in the cloud provider with infrastructure as code.
E. Edit firewall rules and allow all traffic between the applications.

Answer:

The steps are A, C, and D. You can troubleshoot by checking for any drift in firewall rules. If you do not find drift, you can search the infrastructure as code running the application for other discrepancies. From a troubleshooting perspective, this problem may not need a new environment for testing or allowing all traffic between applications.

11.5 Summary

- Repairing failures for infrastructure as code involves rolling forward fixes instead of rolling back.
- Rolling forward infrastructure as code uses immutability to return the system to a working state.
- Before debugging and implementing long-term fixes, prioritize stabilizing and restoring the system to a working state.
- When troubleshooting infrastructure as code, check for drift, unexpected dependencies, and differences between environments as part of the root cause.
- Work on incremental fixes to quickly recognize and reduce the blast radius of potential failure.
- Before you re-implement the original change that failed, make sure you reconcile drift and differences between environments for accurate testing and future system updates.
- Reconstructing state in infrastructure as code to reconcile drift involves aggregating manual commands for server configuration or transforming infrastructure metadata into infrastructure as code.

12 Cost of cloud computing

This chapter covers

- **Investigating cost drivers for cloud cost**
- **Comparing cost optimization practices**
- **Implementing tests for cost-compliant infrastructure as code**
- **Calculating an estimate for infrastructure cost**

When you use a cloud provider, you get very excited at the ease of provisioning. After all, you can create a resource at the click of the mouse or a single command. However, the cost of cloud computing becomes a concern as your organization scales and grows. Your updates to infrastructure as code can affect the overall cost of the cloud!

We must build cost considerations into our infrastructure just as we build security into it. If you make your system and find out you overran your cost, you could break the system trying to remove resources and reduce your cloud computing bill. In Chapter 8, I recommended baking security into infrastructure as code like a cake. The cost of ingredients affects how many cakes we could bake, essential to know *before* I start.

This chapter covers the practices you can combine with infrastructure as code to manage the cost of cloud computing and reduce unused resources. You will find some high-level, general cost control practices and patterns that I describe in the context of infrastructure as code. However, regularly apply these practices to re-optimize costs as your system evolves based on customer demand, organizational scale, and cloud provider billing.

What about the cost of my datacenter or managed services?

I focus primarily on cloud computing because of its flexibility and on-demand billing. You often account for the cost of datacenter computing with your organization's chargeback system. Each business unit establishes a budget for their datacenter resources, and the technology department issues ***chargeback*** based on resource usage, taking into account the operating cost of the datacenter.

You can always apply the practices for managing your cost drivers with cost control and estimation. However, the techniques I outline for cost reduction and optimization may or may not work for every situation (no matter cloud, datacenter, or managed service). Depending on your scale, geographic architecture, business domain, or datacenter usage, your use cases and systems may require specialized assessment or re-platforming.

12.1 Manage cost drivers

You work as a consultant for a company that needs to migrate its platform that supports conferences and events to the public cloud. They ask you to "lift and shift" their configuration in their datacenter to the public cloud. You help their teams build out infrastructure in Google Cloud Platform (GCP), applying all of the principles and practices you learned from this book. Eventually, your team rolls out the platform on GCP and successfully supports its first customer - a small three-hour community conference.

A few weeks after the event, your client schedules a cryptic meeting. When the meeting starts, they show you their cloud bill. It totals over $10,000 for the development and support of a single three-hour conference! The finance team does not seem happy with the cost, especially since they lost money running the conference. You get your next task: *reduce the cost per conference as much as possible*.

Are you using an actual cloud bill?

The example uses a fictitious, *very* simplified cloud bill that approximates the cost of a conference platform service based on Google Cloud Platform (GCP)'s pricing calculator (https://cloud.google.com/products/calculator) as of 2021. The estimates may not include all of the offerings you need, updated pricing for the platform, differences between environments, or the sizes you might use in a comparable system! I rounded the subtotals to streamline the example.

If you run the example, you may reach a GCP quota for the N2D machine type instances. The servers will exceed the platform's free tier! You can change the machine type to free tier instances to run the examples without a charge.

Thanks to the tagging practices you borrowed from Chapter 8, the bill uses the tag to identify which resources belong to the community conference and their environment. In table 12.1, you manage to break down your cloud computing bill and identify the costs by the type and size of the infrastructure resource.

Table 12.1. Your cloud bill by offering and the environment

Offering	Subtotal, Testing Environment	Subtotal, Production Environment	Subtotal
Compute (Servers)	$400	$3,600	$4,000
Database (Cloud SQL)	$250	$2,250	$2,500
Messaging (Pub/Sub)	$100	$900	$1,000
Object Storage (Cloud Storage)	$100	$900	$1,000
Data Transfer (Networking Egress)	$100	$900	$1,000
Other (Cloud CDN, Support)	$50	$450	$500
Total	$1,000	$9,000	$10,000

Separating cost by offerings and environments helps you identify what factors contribute to the cost and where you should investigate further. To start reducing the bill, you must determine the ***cost drivers***, the factors or activities that affect the total cost.

DEFINITION Cost drivers are the factors or activities that affect your total cloud computing cost.

When you assess cost drivers, calculate the percentage cost of cloud offerings. Some offerings will always cost more than others. You can still use the breakdown to help you identify services to optimize. Breaking down cost by environment helps you identify the footprint of testing versus production environments. Comparing the two will give you a better picture of which environment you can reduce inefficiencies.

Based on your breakdown, you calculate the percentage for each offering and environment. In figure 12.1, you chart out that compute resources take 40% of the bill. You also discover that the team spent 10% of the total on the testing environment and 90% on the production environment.

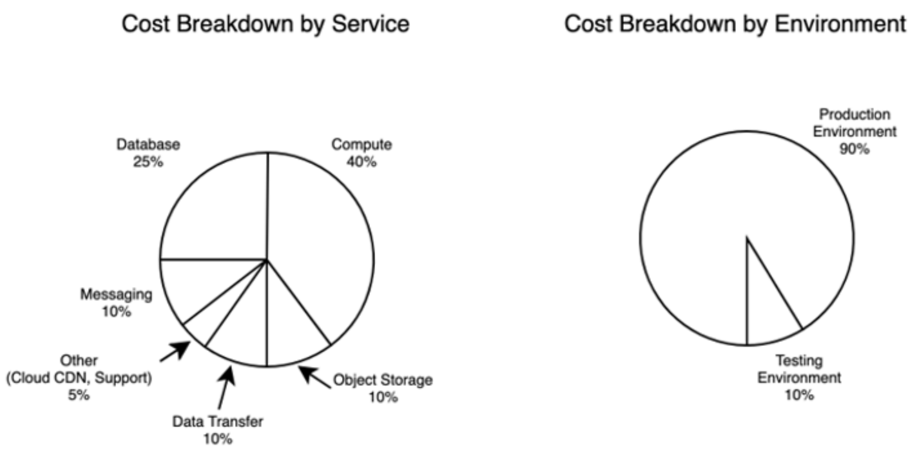


Figure 12.1. The resource tags break down the cost by service and environment.

A large part of the bill goes to compute resources, specifically servers. If your team needs to support an even larger conference, they need to control the type of resources they create and optimize the resource size based on usage. You decide to investigate methods of controlling the size and types of servers the team can use.

12.1.1 Implement tests to control cost

You examine the metrics for the conference and the resource usage for each server. None of the servers exceeded their virtual CPU (vCPU) or memory usage. For the most part, you determine that you need at most 32 vCPUs for your production environment. Your client's infrastructure team confirms that the maximum usage does not exceed 32 vCPUs.

> **NOTE** GCP uses the term "machine type" to refer to a predefined virtual machine shape with specific vCPU and memory ratios to fit your workload requirements. Similarly, Amazon Web Services (AWS) uses the term "instance type."

However, the public cloud makes it easy for anyone to adjust a server to use 48 vCPU. Your bill increases by 50% because of the additional CPU, and you don't even use all of it. To more proactively control the cost using infrastructure as code, you combine unit testing from Chapter 6 and policy enforcement from Chapter 8.

Figure 12.2 adds a new policy test to the server’s delivery pipeline. The test checks every server defined in infrastructure as code for its number of vCPUs and compares it to the value returned by the GCP API. If the API’s information exceeds your maximum vCPU limit of 32, your test fails.

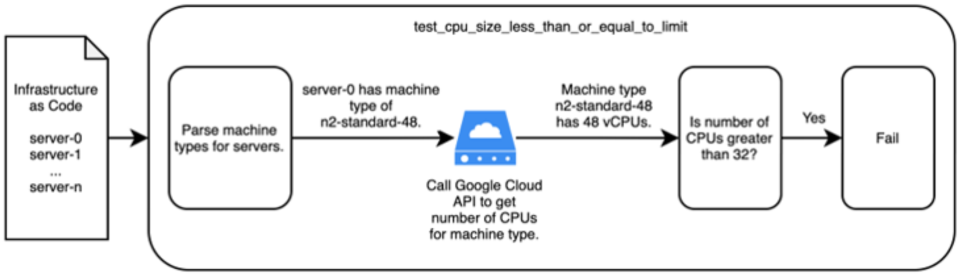

test_cpu_size_less_than_or_equal_to_limit
Infrastructure as Code
server-0
server-1
...
server-n
Parse machine types for servers.
server-0 has machine type of n2-standard-48.
Call Google Cloud API to get number of CPUs for machine type.
Machine type n2-standard-48 has 48 vCPUs.
Is number of CPUs greater than 32?
Yes
Fail

Figure 12.2. Your test should parse machine types from server configurations, check the GCP API for the number of CPUs, and verify they do not exceed the limit.

Why use call the infrastructure provider’s API for vCPU information? Many infrastructure providers offer an API or client library to retrieve information about a given machine type from their catalogs. You can use this to dynamically get information about the number of CPUs and memory.

Infrastructure providers change offerings frequently. Furthermore, you cannot account for every possible server type. Writing your test to call the infrastructure provider API for the most updated information helps improve the overall evolvability of the test.

Let’s implement the policy test to check for the maximum vCPU limit. First, you build a method to call the GCP API for the number of vCPUs for a given machine type.

Listing 12.1 Retrieve vCPU count for machine type from GCP API

```
import googleapiclient.discovery

class MachineType():    #A
   def __init__(self, gcp_json):
       self.name = gcp_json['name']
       self.cpus = gcp_json['guestCpus']    #A
       self.ram = self._convert_mb_to_gb(    #B
           gcp_json['memoryMb'])    #B
       self.maxPersistentDisks = gcp_json[
           'maximumPersistentDisks']
       self.maxPersistentDiskSizeGb = gcp_json[
           'maximumPersistentDisksSizeGb']
       self.isSharedCpu = gcp_json['isSharedCpu']

   def _convert_mb_to_gb(self, mb):    #B
       GIGABYTE = 1.0/1024    #B
       return GIGABYTE * mb    #B

def get_machine_type(project, zone, type):
   service = googleapiclient.discovery.build(
       'compute', 'v1')
   result = service.machineTypes().list(    #C
       project=project,    #C
       zone=zone,    #C
       filter=f'name:"{type}"').execute()    #C
   types = result['items'] if 'items' in result else None
   if len(types) != 1:
       return None
   return MachineType(types[0])    #D
```

#A Define a machine type object to store any attributes you might need to check, including number of vCPUs.
#B Convert megabytes of memory to gigabytes for consistent unit measure.
#C Call the GCP API to retrieve the number of vCPUs for a given machine type.
#D Return a machine type object with vCPU and disk attributes.

Any time you use a new machine type, you can use the same function to retrieve vCPUs and memory. Next, you write a test to parse every server defined in your configuration for their machine type. In listing 122., you retrieve the number of vCPUs for the machine type for a list of servers and verify that the vCPUs do not exceed the limit of 32.

Listing 12.2 Write policy test to check servers do not exceed 32 vCPU

```
import pytest
import os
import compute
import json

ENVIRONMENTS = ['testing', 'prod']    #A
CONFIGURATION_FILE = 'main.tf.json'     #A

PROJECT = os.environ['CLOUDSDK_CORE_PROJECT']

@pytest.fixture(scope="module")     #A
def configuration():
   merged = []
   for environment in ENVIRONMENTS:
       with open(f'{environment}/{CONFIGURATION_FILE}', 'r') as f:
           environment_configuration = json.load(f)
           merged += environment_configuration['resource']
   return merged

def resources(configuration, resource_type):     #A
   resource_list = []
   for resource in configuration:
       if resource_type in resource.keys():
           resource_name = list(
               resource[resource_type].keys())[0]
           resource_list.append(
               resource[resource_type]
               [resource_name])
   return resource_list

@pytest.fixture
def servers(configuration):     #A
   return resources(configuration,
                    'google_compute_instance')

def test_cpu_size_less_than_or_equal_to_limit(servers):
   CPU_LIMIT = 32    #B
   non_compliant_servers = []    #F
   for server in servers:
       type = compute.get_machine_type(    #C
           PROJECT, server['zone'],    #C
           server['machine_type'])    #C
       if type.cpus > CPU_LIMIT:    #D
           non_compliant_servers.append(server['name'])    #D
   assert len(non_compliant_servers) == 0, \     #E
       f'Servers found using over {CPU_LIMIT}' + \   #E
       f' vCPUs: {non_compliant_servers}'   #E
```

#A Parse and extract any server JSON configurations across testing and production environments.
#B Set the CPU limit to 32, the maximum required for the application.
#C For each server configuration, retrieve the machine type attribute and call GCP API for more information.
#D If the server configuration includes a machine type that exceeds 32 vCPUs, add it to a list of non-compliant servers.
#E Check that all servers comply with the CPU limit. If not, fail the test and throw an error for the servers that exceed 32 CPUs.

#F Initialize a list of non-compliant servers which exceed the 32 vCPU limit.

You configure the test with a soft mandatory enforcement policy. Soft mandatory enforcement means your team reviews and approves the more expensive resource type before you create it. You can override the machine type to a larger size if you have a business justification.

Outside of checking machine types for vCPU and memory limits, you may also need to add overrides for unique architectures or machine types that apply to certain use cases, like machine learning. However, they cost more than a general-purpose resource type.

You can test that infrastructure as code uses general-purpose resources by default. General-purpose machine or resource types offer lower-cost options. If someone needs a specialized, more expensive resource, you can enable them with soft mandatory enforcement.

Other tests might include checks for specific configurations like scheduled reboots, automatic scaling, or private networking. Each of these configurations contributes to optimizing the cost of your resources. Expressing them in infrastructure as code lets you verify configuration conforms to best practices to reduce cost early in the development process.

12.1.2 Automate cost estimation

You react to too large or expensive resource changes with policy tests to control costs. What if you want a proactive way to check how you change your budget by changing cost drivers? Imagine you want to know how adjusting your production server size to the machine type of "n2d-standard-16" (16 vCPU) might affect the future cost of a different three-hour conference.

Figure 12.3 outlines the workflow to *estimate* the cost of five servers with the machine type "n2d-standard-16". Once you calculate the price, you can add a policy test to verify the total does not exceed your monthly budget.

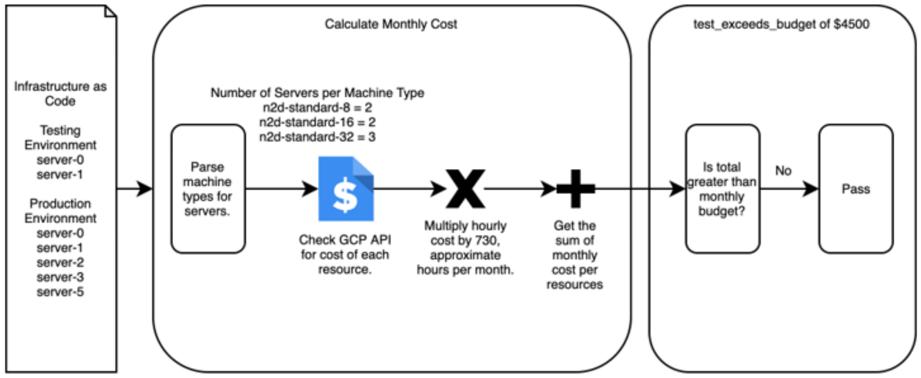

Infrastructure as Code
Testing Environment
server-0
server-1
Production Environment
server-0
server-1
server-2
server-3
server-5
Calculate Monthly Cost
Number of Servers per Machine Type
n2d-standard-8 = 2
n2d-standard-16 = 2
n2d-standard-32 = 3
Parse machine types for servers.
Check GCP API for cost of each resource.
Multiply hourly cost by 730, approximate hours per month.
Get the sum of monthly cost per resources
test_exceeds_budget of $4500
Is total greater than monthly budget?
No
Pass

Figure 12.3. Cost estimation parses infrastructure as code for machine types, calculates the monthly cost of the resource, and generates a value to compare with your expected budget.

Cost estimation parses your infrastructure as code for resource attributes and generates an estimate of their cost. You can use cost estimation to check that your changes stay within your budget or assess adjustments to cost drivers.

> **DEFINITION** Cost estimation extracts infrastructure resource attributes and generates an estimate of their total cost.

How does cost estimation help you evolve your infrastructure? Cost estimation offers additional transparency into cost drivers that may affect your architecture. As you change your system, you can use these tests to help budget and communicate chargeback across teams.

Cost estimation example and tools

I wrote minimal code to demonstrate the general workflow of cost estimation. For clarity, I omitted some of the code from the text. You can find all of the code organized at https://github.com/joatmon08/manning-book/tree/main/ch12.

The example uses the Google Cloud Billing Catalog API, which offers a service catalog with pricing. I also use a specialized Python client library for the Cloud Catalog to access the billing API (https://googleapis.dev/python/cloudbilling/latest/billing_v1/cloud_catalog.html). The example does not account for specialized pricing, such as sustained use discounts or preemptible (equivalent to spot instances in AWS).

You will find a few tools that offer cost estimation. Each cloud provider offers its own cost estimator user interface where you can enter your resources. Other tools implement a more scalable workflow than my example code to parse configuration, call a cloud provider's API and calculate a cost estimate. I will not list them in this chapter, as they change frequently and depend on the cloud provider and your infrastructure as code tooling.

GET PRICE PER UNIT

I recommend dynamically requesting information from a cloud provider’s service catalog API. Price per unit can change, and hard-coding the prices often results in the wrong cost estimation. To start implementing cost estimation in the example, you need some logic to call the cloud provider's catalog and retrieve the price per unit based on your machine type.

The Google Cloud Catalog API offers a list of services and stock-keeping units (SKUs) based on price per unit of CPU or memory (RAM). In your code, you get the service identifier for the Google Compute Engine service. The Google Cloud Catalog API categorizes prices based on service identifiers, which you must retrieve dynamically.

Listing 12.3 Get Google Compute Engine service from the catalog

```
from google.cloud import billing_v1

class ComputeService:
   def __init__(self):
       self.billing = \    #A
           billing_v1.services.cloud_catalog.CloudCatalogClient()    #A
       for result in self.billing.list_services():
           if result.display_name == 'Compute Engine':    #B
               self.name = result.name
```

#A Create a client with the Python library for Google Cloud Billing API.
#B Get the service identifier for Google Compute Engine in the catalog.

You can call the Google Cloud Catalog API a second time for the price of a machine type. Using the service identifier from the last step, you get a list of SKUs for the Google Compute Engine service. You write some code to parse its response list of SKUs to match the machine type and purpose, and retrieve the unit price per CPU or gigabyte of memory.

Listing 12.4 Get CPU and RAM price for Compute Engine SKUs

```
from google.cloud import billing_v1

class ComputeSKU:
   def __init__(self, machine_type, service_name):
       self.billing = \
           billing_v1.services.cloud_catalog.CloudCatalogClient()     #A
       self.service_name = service_name
       type_name = machine_type.split('-')    #B
       self.family = type_name[0]    #B
       self.exclude = [    #C
           'custom',    #C
           'preemptible',    #C
           'sole tenancy',    #C
           'commitment'    #C
       ] if type_name[1] == 'standard' else []    #C

   def _filter(self, description):    #C
       return not any(    #C
           type in description for type in self.exclude    #C
       )    #C

   def _get_unit_price(self, result):     #D
       expression = result.pricing_info[0]
       unit_price = expression. \
           pricing_expression.tiered_rates[0].unit_price.nanos \
           if expression else 0
       category = result.category.resource_group
       if category == 'CPU':
           self.cpu_pricing = unit_price
       if category == 'RAM':
           self.ram_pricing = unit_price

   def get_pricing(self, region):    #E
       for result in self.billing.list_skus(parent=self.service_name):
           description = result.description.lower()    #F
```

```
            if region in result.service_regions and \    #F
                    self.family in description and \    #F
                    self._filter(description):    #F
                self._get_unit_price(result)
        return self.cpu_pricing, self.ram_pricing
```

#A Create a client with the Python library for Google Cloud Billing API.
#B For a machine type like "n2d-standard-16", extract the machine family (n2d) and purpose (standard) to identify the SKU.
#C If you use a standard machine type, do not search for any specialized Compute Service SKUs in the catalog.
#D Retrieve the unit price per CPU or RAM in nano-dollars (10^-9).
#E Call the Google Cloud Catalog and retrieve a list of SKUs for the Compute Service.
#F Find SKUs that match the region, machine family, and purpose of the machine type based on its description.

The Google Cloud Catalog sets a unit price based on the number of CPUs and gigabytes of memory. As a result, you cannot search based on the name of the machine type. Instead, you need to correlate the general-purpose machine type with the catalog's description.

CALCULATE THE MONTHLY COST FOR A SINGLE RESOURCE

Once you retrieve the CPU and RAM unit price for a given machine type, you can use it to calculate a monthly cost for a single instance of the machine. Some cloud catalogs set up a unit price by a factor. For example, GCP uses nano units, which means I need to also multiply by the factor. Listing 12.5 implements the code to calculate the monthly cost for a single server. I multiply the unit price by the average number of hours per month, 730, and the nano units.

Listing 12.5 Calculate the monthly cost for a single server

```
HOURS_IN_MONTH = 730    #A
NANO_UNITS = 10**-9    #A

def calculate_monthly_compute(machine_type, region):
   service_name = ComputeService().name     #B
   sku = ComputeSKU(machine_type.name, service_name)    #C
   cpu_price, ram_price = sku.get_pricing(region)    #C

   cpu_cost = machine_type.cpus * cpu_price * \     #D
       HOURS_IN_MONTH if cpu_price else 0     #D
   ram_cost = machine_type.ram * ram_price * \     #E
       HOURS_IN_MONTH if ram_price else 0     #E
   return (cpu_cost + ram_cost) * NANO_UNITS     #F
```

#A Set a constant for the average of 730 hours in a month and convert nano-dollars to dollars (10^-9).
#B Get the Compute Engine service identifier from the Google Cloud Catalog API.
#C Set up the SKU for the machine type and get its CPU and RAM unit pricing.
#D Multiply a machine type's number of CPUs by unit price and hours in the month.
#E Multiply a machine type's gigabytes of memory (RAM) by unit price and hours in the month.
#F Sum up the CPU and RAM cost and convert it to dollar units.

You now have a minimal form of cost estimation that calculates the cost of a single server. With the initial cost calculation for a single server, you can parse your infrastructure as code for all servers, retrieve their machine types and regions, and calculate a total cost. In the future, you can add more logic to retrieve SKUs for other services, like databases or messaging.

CHECK YOUR COST DOES NOT EXCEED THE BUDGET.

You decide that you can do more with your cost estimation. You write a test with a soft mandatory enforcement approach to check if your estimated cost exceeds your monthly budget.

For example, your client tells you that a conference should not exceed a monthly budget of $4,500. You can compare your cost estimation to the budget and proactively identify any cost drivers.

Let’s write a test to estimate the new cost of servers and compare it to the budget. In the code, you parse your infrastructure as code for all servers and count the number of servers with specific machine types and regions.

Listing 12.6 Parse infrastructure as code for all servers

```
from compute import get_machine_type
import pytest
import os
import json

ENVIRONMENTS = ['testing', 'prod']
CONFIGURATION_FILE = 'main.tf.json'

@pytest.fixture(scope="module")
def configuration():     #A
   merged = []
   for environment in ENVIRONMENTS:
       with open(f'{environment}/{CONFIGURATION_FILE}', 'r') as f:
           environment_configuration = json.load(f)
           merged += environment_configuration['resource']
   return merged

@pytest.fixture
def servers(configuration):     #B
   servers = dict()
   server_configs = resources(configuration,
                              'google_compute_instance')
   for server in server_configs:
       region = server['zone'].rsplit('-', 1)[0]
       machine_type = server['machine_type']
       key = f'{region},{machine_type}'
       if key not in servers:
           type = get_machine_type(     #C
               PROJECT, server['zone'],     #C
               machine_type)     #C
           servers[key] = {     #D
               'type': type,     #D
               'num_servers': 1     #D
           }     #D
       else:
           servers[key]['num_servers'] += 1     #D
   return servers
```

#A Read every configuration file that defines each environment, such as testing and production.
#B For each server in the configuration file, create a list of their regions and machine types.
#C Call the Google Compute API and get details on the machine type, such as its number of CPUs and memory.

#D Track the number of servers with the specific machine type and region to streamline the SKUs you need to retrieve.

You can call these methods in your test to retrieve cost information for each machine type in a specific region and sum the total cost. The test checks if the total cost exceeds the monthly budget of $4,500.

Listing 12.7 Get CPU and RAM price for Compute Engine SKUs

```
from estimation import calculate_monthly_compute

PROJECT = os.environ['CLOUDSDK_CORE_PROJECT']
MONTHTLY_COMPUTE_BUDGET = 4500     #A

def test_monthly_compute_budget_not_exceeded(servers):     #B
   total = 0
   for key, value in servers.items():
       region, _ = key.split(',')
       total += calculate_monthly_compute(value['type'], region) * \     #C
           value['num_servers']     #C
   assert total < MONTHTLY_COMPUTE_BUDGET     #D
```

#A Set a constant to communicate the expected monthly compute budget.
#B Test that the cost of your servers does not exceed the monthly compute budget.
#C Calculate the monthly total per server based on machine type and region, multiply by servers for that machine type, and sum up the total.
#D Confirm that the estimated total cost does not exceed the monthly compute budget.

You now have a test to estimate the total monthly cost of computing resources and compare it to your budget! Each time someone changes the infrastructure, the test recalculates the new cost of the system.

A cost estimation gives you a general view of your infrastructure cost but may not accurately reflect your actual bill. You’ll have to account for some margin of error. If your estimation exceeds the monthly budget, it may indicate that you need to reassess size and resource usage. You’ll also refine your monthly budget over time depending on the growth of your systems.

CONTINUOUS DELIVERY WITH COST ESTIMATION

How do you check that infrastructure changes do not exceed your budget? Each time you change your infrastructure as code and push it to a repository, your cost estimation and test for budget runs. Budget tests in your pipeline help you identify costly infrastructure changes and refine the resource in a testing environment. The process prevents chargeback in the production environment.

For example, let's say you want to add another server to the testing environment for a different conference. In figure 12.4, you create a configuration to add another server with the machine type of "n2d-standard-8". The pipeline runs a test to calculate the monthly cost with the new server and check it against the monthly budget.

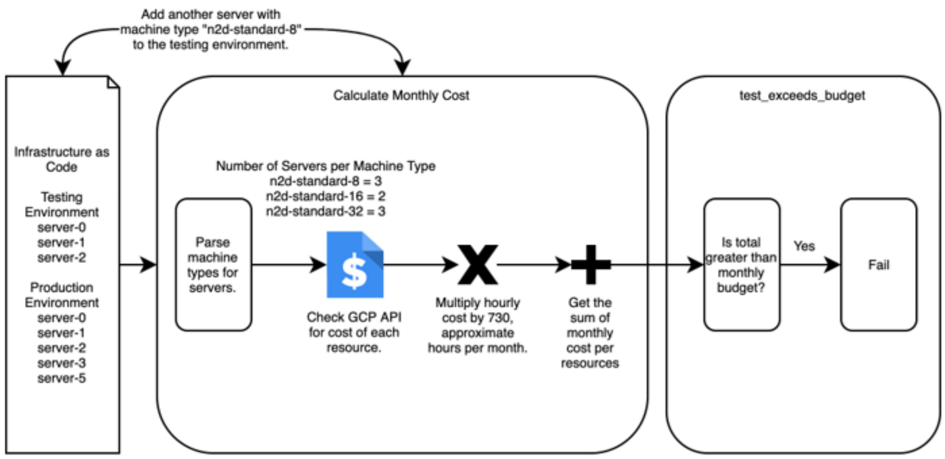

Add another server with machine type "n2d-standard-8" to the testing environment.
Infrastructure as Code
Testing Environment
server-0
server-1
server-2
Production Environment
server-0
server-1
server-2
server-3
server-5
Calculate Monthly Cost
Number of Servers per Machine Type
n2d-standard-8 = 3
n2d-standard-16 = 2
n2d-standard-32 = 3
Parse machine types for servers.
Check GCP API for cost of each resource.
Multiply hourly cost by 730, approximate hours per month.
Get the sum of monthly cost per resources
test_exceeds_budget
Is total greater than monthly budget?
Yes
Fail

Figure 12.4. Adding another server exceeds your budget of $4,500, causing the test to fail.

You push the configuration to the repository, and your delivery pipeline runs the tests to check for budget compliance. The pipeline fails! You check the logs and discover that your cost estimation exceeds the expected monthly budget.

```
$ pytest test_budget.py
FAILED test_budget.py::test_monthly_compute_budget_not_exceeded - assert 4687.6161600000005
       < 4500
```

You speak with your finance team. The finance analyst confirms that the budget can increase to accommodate the new testing instance. You update the monthly budget within the test to $4,700 for future changes!

Whether you write a cost estimation mechanism or use a tool, you should consider adding it to your delivery pipelines as another test for policy. Estimation helps guide instance sizing and usage. It should *never* stop a change before production. Instead, it should provide an *opportunity* for you to reassess the need for the resource.

Do not account for every cost driver as part of your cost estimation. Instead, choose the resources that make up the bulk of your bill. The example focuses on computing resources like servers, which often contribute significantly to the cost. You might implement cost estimation for other resources, like databases or messaging frameworks.

Always question the accuracy of your cost estimation! You cannot predict the resources you will create or how you use them. For example, you might find it hard to estimate the cost to transfer data between regions or services until it happens. Reconcile your cost estimate with your monthly bill and assess which cost drivers contribute to the difference.

A monthly comparison can help you identify any changes and budget for the actual cost based on a multiplier of the estimate. In the remainder of the chapter, we'll discuss ways to reduce cloud waste and optimize cost outside of proactive measures like testing or estimation.

Exercise 12.1

Given:

```
HOURS_IN_MONTH = 730
MONTHLY_BUDGET = 5000
DATABASE_COST_PER_HOUR = 5
NUM_DATABASES = 2
BUFFER = 0.1

def test_monthly_budget_not_exceeded():
  total = HOURS_IN_MONTH * NUM_DATABASES * DATABASE_COST_PER_HOUR
  assert total < MONTHLY_BUDGET + MONTHLY_BUDGET * BUFFER
```

Which are true? (Choose all that apply.)

- **A. The test will pass because the cost the database is within the budget.**
- **B. The test estimates the monthly cost of databases.**
- **C. The test does not account for different database types.**
- **D. The test calculates the monthly cost per database instance.**
- **E. The test includes a buffer of 10% for any cost overage as a soft mandatory policy.**

Find answers and explanations at the end of the chapter.

12.2 Reduce cloud waste

You can use infrastructure as code to implement some proactive measures to manage cost drivers for cloud computing. However, you need to combine them with other practices to continue to reduce and optimize costs. After all, your client in the example still doesn't appreciate a $10,000 cloud bill for a three-hour conference!

If you provision a large server but do not use all the CPU or memory, you have wasted unused CPU or memory. You have an opportunity to reduce your cloud computing cost! One approach you can take to improve your bill's state involves eliminating ***cloud waste***, unused or underutilized infrastructure resources.

DEFINITION **Cloud waste describes unused or underutilized infrastructure resources.**

In figure 12.5, you can reduce cloud waste by deleting, expiring, or stopping unused resources, scheduling or scaling instances based on usage, and assessing the right resource size or type for your system.

Reducing Cloud Waste
Stop untagged or unused resources.
Start and stop resources on a schedule.
Choose the right resource type, size, and reservation.
Enable autoscaling for resources.
Set a tag for resource expiration.

Figure 12.5. You can reduce cloud waste by removing unused resources or scheduling and sizing resources to accommodate usage patterns.

Identifying cloud waste often starts as the first response to a surprising cost on your public cloud bill. However, you can use these techniques in your datacenter, especially for a private cloud. While they do not provide an immediate short-term benefit, they help optimize data center resource usage and long-term cost reduction.

12.2.1 Stop untagged or unused resources

Sometimes, you and your team will create infrastructure resources for testing or other configuration. You end up forgetting about them until they show up on your cloud bill. As a first iteration of reducing cloud waste, you can identify unused resources and remove them.

Recall that you have a mission of reducing the cost of running a conference for your client. Could you reduce costs by identifying unused resources? Yes! Sometimes, our teams create resources for testing and forget to remove them.

For example, you retrieve a list of servers in your Google Cloud projects and examine them in table 12.2. While many of them in testing and production have tags, you notice two instances with no tags. The "n2d-standard-16" machines cost about $700 (7% of the total monthly bill).

Table 12.2. Cost of servers by type and environment

Machine Type	Environment	Number of Servers	Subtotal
n2d-standard-8	**Testing**	2	$400
n2d-standard-16	**Production**	2	$700
n2d-standard-32	**Production**	3	$2,900
Total			$4,000

You ask the team about the untagged instances in production. They created the servers for a sandbox to verify the application but never used them. Just to make sure, you check the server usage metrics over the month, and they all stayed at zero. You identified some cloud waste!

The team did create the servers with infrastructure as code. You delete the configuration and push the changes to remove the unused instances. Deleting the configuration removes the disks and resources attached to the instances. Fortunately, the cloud bill for your next conference reflects the reduction.

Why would you confirm usage of the server by metrics and team members? You do not want to remove the resource by accident. Sometimes, you have unexpected dependencies on what seems like an unused resource.

Make sure that any resources you intend to delete do not have additional dependencies. If you have concerns about deleting an untagged or unused resource, you can always stop the resource for a week or two, wait to determine if it breaks the system, and then delete it.

12.2.2 Start and stop resources on a schedule

Your next cloud bill comes back 7% less, thanks to the removal of unused servers. However, the finance team wants you to reduce it even more. You puzzle over this until you talk to one of the client's team members. They mention that they never run testing or use any of their infrastructure resources over the weekends. The client needs the platform available the weekend before the conference.

Could you find a way to turn off the servers on a Friday night and turn them on each Monday? You do not get charged for the 48 hours you shut down the servers. Scheduling a regular shutdown means cost reduction.

You discover that GCP defines an instance shutdown schedule with a compute resource policy (https://cloud.google.com/compute/docs/instances/schedule-instance-start-stop). Figure 12.6 outlines that you start the servers each Monday and shut them down each Saturday.

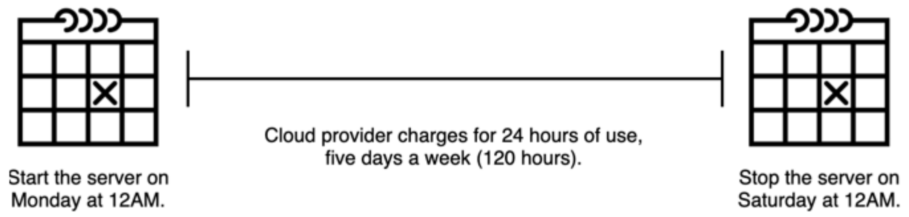

Start the server on Monday at 12AM.
Cloud provider charges for 24 hours of use, five days a week (120 hours).
Stop the server on Saturday at 12AM.

Figure 12.6. You can reduce cost by scheduling a resource to start and stop when not in use.

Shutting down instances on a schedule alleviates the cost of running servers. However, this technique *only* works if you understand the behavior of your system. Starting and stopping resources on a schedule can disrupt development work.

Some applications do not have fault tolerance and would continue to fail even if a resource restarts successfully. In general, most reboot schedules only run in testing environments. The schedule provides an opportunity for you to verify your system's resilience because you have a planned outage each weekend.

Listing 12.7 implements the resource policy for instance scheduling in GCP. The schedule expires the week before the conference, so it does not shut down the servers over the weekend. The development team might need to work on the platform in the few days before the conference.

Listing 12.7 Create resource policy for instance scheduling

```
def build(name, region, week_before_conference):
   expiration_time = datetime.strptime(     #D
       week_before_conference,     #D
       '%Y-%m-%d').replace(     #D
           tzinfo=timezone.utc).isoformat().replace(     #D
               '+00:00', 'Z')     #D
   return {
       'google_compute_resource_policy': {
           'weekend': {
               'name': name,
               'region': region,
               'description':
               'start and stop instances over the weekend',
               'instance_schedule_policy': {     #A
                   'vm_start_schedule': {     #B
                       'schedule': '0 0 * * MON'     #B
                   },     #B
                   'vm_stop_schedule': {     #C
                       'schedule': '0 0 * * SAT'     #C
                   },     #C
                   'time_zone': 'US/Central',     #E
                   'expiration_time': expiration_time     #D
               }
           }
       }
   }
```

#A Create a compute resource policy with an instance schedule.
#B Start the virtual machines every Monday at midnight.
#C Stop the virtual machines every Saturday at midnight.
#D Expire the schedule the week before the conference using the RFC 3339 date format.
#E Run the schedule in the US central time zone since development teams work in the central United States.

AWS equivalent

Other public cloud providers usually offer a similar automation capability to start and stop the virtual machines on a schedule. For example, AWS uses Instance Scheduler to start and stop servers and databases (https://aws.amazon.com/solutions/implementations/instance-scheduler/). Similarly, Azure uses a start/stop VMs workflow based on Azure functions (https://docs.microsoft.com/en-us/azure/azure-functions/start-stop-vms/overview).

If your public or private cloud platform does not offer scheduled shutdown capabilities, you will need to write your automation to run on a regular schedule. I implemented this before using various tools, including serverless functions, cron jobs on container orchestrators, or scheduled runs on continuous integration frameworks.

Instead of running seven servers 730 hours per month, you run it about 144 hours less (assuming three weekends in a month and a 48-hour shutdown). Using your cost estimation code, you update your calculation for 586 hours per month. It outputs that you reduced the overall cost by $700 (7% of the total monthly bill)!

The example adds the schedule to the testing environment. However, you could add a reboot schedule to a production environment if it has cyclical usage patterns. For example, the conference platform runs on-demand for three hours and only serves user traffic during the week. Shutting down servers and databases for 48 hours does not disrupt user traffic. However, you may not want to implement a reboot schedule in a production environment that serves requests continuously.

12.2.3 Choose the correct resource type and size

If your production environment needs to serve customers 24 hours a day, seven days a week, you can still reduce cloud waste without a resource schedule by assessing your resource types and sizes. Many resources do not utilize their CPU or memory fully.

Many times, we provision larger resources because we don’t know how much we need. After running a system for some time, you can adjust the size of the resource for its actual usage. In figure 12.7, you can reduce cost by changing a resource type, size, reservation, replicas, or even cloud provider.

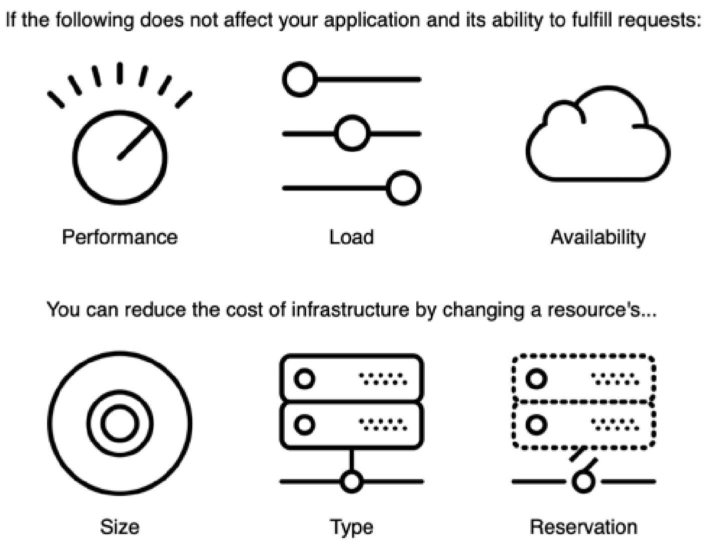

If the following does not affect your application and its ability to fulfill requests:
Performance
Load
Availability
You can reduce the cost of infrastructure by changing a resource's...
Size
Type
Reservation

Figure 12.7. You can change a resource's attributes to utilize it better and reduce its cost.

You decide to investigate your client's conference platform to find cloud waste in resource type and size. You realize you cannot reduce the cost of computing resources, so you check the database (Cloud SQL). In table 12.3, the team provisioned the production database for 4TB of SSD storage.

Table 12.3. Your cloud bill by offering and resource

Offering	Type	Environment	Number	Subtotal
Cloud SQL				$2,500
	db-standard-1, 400GB SSD, 600GB backup	Testing	1	$250
	db-standard-4, 4TB SSD, 6TB backup	Production	1	$2,250

After checking metrics and database usage, you realize that it only needs a 1TB SSD. You update the disk size of the database in your infrastructure as code. Fixing the size reduces the cost of the database by $1,350 (22.5% of the total monthly bill)!

You might not use a resource to its full potential in many other ways. You might consider changing the type of resource if it uses a more expensive machine type. You need to ask yourself and your team, "Do we need this high-performance database in my testing environment if we do not run performance tests?"

Probably not! Choosing the right size and type for a given environment may take a few iterations. You want to choose a resource type, size, and replicas that simulate production without making it an exact duplicate in cost.

For the conference example, you might have three "n2d-standard-32" instances in production and three "n2d-standard-8" instances in your testing environment. The configuration still tests the three application instances without incurring a cost of 72 CPUs.

Other times, you can change the resource's reservation type. GCP and many other cloud providers offer an "ephemeral" (also known as "spot" or "preemptible") resource type. The resource costs less, but the cloud provider reserves the right to stop the resource and give CPU or memory to another customer. While an ephemeral resource reservation can reduce cost, you need to carefully consider if your application and system can handle the disruption.

12.2.4 Enable autoscaling

You tried to identify as much cloud waste as possible in your environment but still want to reduce costs further. Many systems have customer usage patterns that do not require their CPU, memory, or bandwidth in the system every hour of every day.

For example, the conference platform only needed 100% of its capacity during the three hours of the conference! Could you have automatically increased or decreased the number of servers based on demand?

Figure 12.8 sets the target utilization to 75% of CPU usage so that the GCP managed instance group starts and stops servers to match the target metric. It increases and decreases the size of the group based on demand.

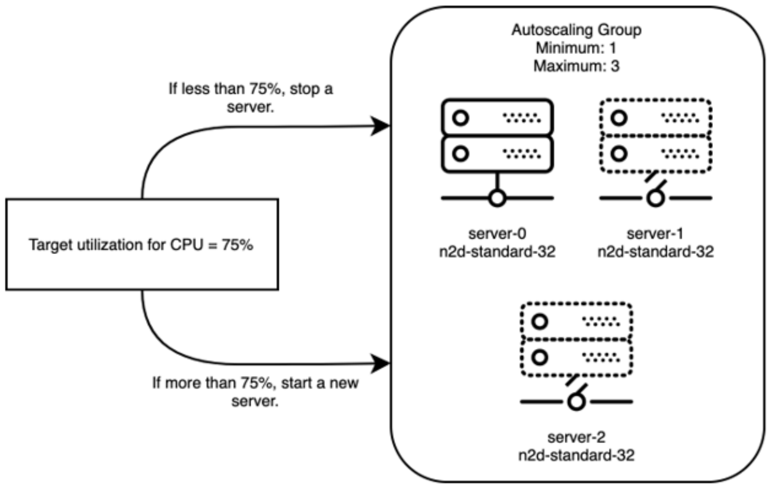


Figure 12.8. An autoscaling group includes a target utilization rate, which allows it to start and stop resources to adjust for usage automatically.

You added autoscaling for each group of servers. ***Autoscaling*** increases or decreases the number of resources in a group based on a metric, such as CPU or memory. Many public cloud providers offer an autoscaling group resource that you can create with infrastructure as code.

> **DEFINITION** Autoscaling is the practice of automatically increasing or decreasing the number of resources in a group based on a metric.

GCP autoscaling requires that you set a target metric to scale up or scale down resources to achieve. For most of the month with low traffic, you expect only to use one server. However, when peak traffic runs through your conference platform, you need a maximum of three. You decide to use a metric for CPU utilization and set the target to 75%.

You update the infrastructure as code for the servers. In listing 12.8, you replace the original servers and instance scheduling resource policy with a managed instance group and autoscaling policy. The autoscaling schedule starts each morning, increases or decreases instances to achieve 75% CPU utilization, and scales the instances to zero each evening.

Listing 12.8 Create an autoscaling group based on CPU utilization

```
def build(name, machine_type, zone,
         min, max, cpu_utilization,
         cooldown=60,
         network='default'):
   region = zone.rsplit('-', 1)[0]
   return [{     #A
       'google_compute_autoscaler': {
           name: {
               'name': name,
               'zone': zone,
               'target': '${google_compute_instance_group_manager.' +    #A
               f'{name}.id}}',
               'autoscaling_policy': {
                   'max_replicas': max,    #B
                   'min_replicas': 0,    #C
                   'cooldown_period': cooldown,
                   'cpu_utilization': {
                       'target': cpu_utilization    #D
                   },
                   'scaling_schedules': {    #E
                       'name': 'weekday-scaleup',    #E
                       'min_required_replicas': min,    #E
                       'schedule': '0 6 * * MON-FRI',    #E
                       'duration_sec': '57600',    #E
                       'time_zone': 'US/Central'    #E
                   }    #E
               }
           }
       }
   }]
```

#A Attach an instance group to the autoscaling resource. We omitted the instance group for clarity.
#B Set the maximum number of replicas to scale up if CPU utilization increases above 75%.
#C Set the minimum number of replicas to zero by default, which means it stops the virtual machines.

#D Use CPU utilization as the target metric for the autoscaling group.
#E Set a scaling schedule that increases the minimum number of replicas each Monday through Friday morning for the development team's usage patterns.

AWS equivalent

GCP attaches managed instance groups to autoscaling policies. GCP does not allow you to attach a resource policy. You must implement the schedule in the autoscaling group.

Other public cloud providers also offer autoscaling capabilities for servers and sometimes databases. AWS uses autoscaling groups. Azure uses autoscale rules for scale sets.

You can set a scaling schedule in the example to mimic a "weekend shutdown" that you previously implemented. In general, use the patterns for modules to create an autoscaling module. The module should set an opinionated, default target metric depending on your workload.

If you have a unique workload that does not fit the module’s preset target metrics, you can set its default to target CPU utilization or memory and assess its behavior over time. When you roll out the instance group, apply the blue-green deployment pattern from Chapter 9 to replace active workloads or instances. Rolling out a schedule and autoscaling group should not disrupt applications.

To encourage teams to use autoscaling groups and scheduling, you can create several policy tests to make sure your autoscaling group reduces cloud waste. For example, one test could verify that your infrastructure as code has no individual servers and only contains autoscaling groups. The test encourages the team to take advantage of elasticity.

Another test you can add involves checking the maximum replica limit. Suppose your application suddenly consumes a lot of CPU or memory, or a bad actor injects a cryptocurrency mining binary on your machine. In that case, you don't want the autoscaling group to automatically increase its capacity to 100 machines.

12.2.5 Set a resource expiration tag

You may reduce cloud waste by dynamically scaling up and down resources based on utilization, but you also need to accommodate on-demand, manually created resources. For example, the client team complains that they often need to create sandbox servers for further testing. However, they often forget about the servers. Can you "expire" the servers after some time if no one updates them?

You decide to update your tag module to attach a new tag for the *expiration date* in the testing environment. Recall that you can use the prototype pattern (Chapter 3) to establish standard tags. After applying the policy tests to check for tag compliance in Chapter 8, you know that each resource in the testing environment will have an expiration date.

For example, in figure 12.9, the team might create a server with an initial expiration date of February 2. However, they decide to update the server. As part of the change, the tag module retrieves the current date (February 5), adds seven days, and updates the tag on the server to a new date (February 12).

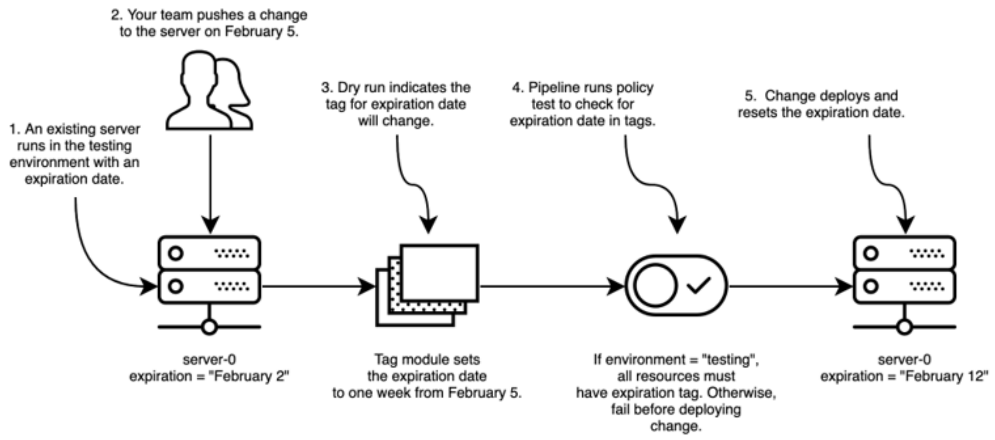

2. Your team pushes a change to the server on February 5.
1. An existing server runs in the testing environment with an expiration date.
3. Dry run indicates the tag for expiration date will change.
4. Pipeline runs policy test to check for expiration date in tags.
5. Change deploys and resets the expiration date.
server-0 expiration = "February 2"
Tag module sets the expiration date to one week from February 5.
If environment = "testing", all resources must have expiration tag. Otherwise, fail before deploying change.
server-0 expiration = "February 12"

Figure 12.9. Create an expiration date in the tag module that resets the expiration date of the module to one week from the change.

Why use set the expiration as part of the tag module? Your tag module should get applied across all your infrastructure as code. This allows you to establish a *default duration* of seven days and apply it to all infrastructure resources

You can also control when to apply the expiration tag as part of a module. The module only applies the expiration tag if you don't create a resource in the production environment or continuously run it in the testing environment. Listing 12.9 updates the prototype module for default tags with an expiration date.

Listing 12.9 Tag module with an expiration date

```
import datetime

EXPIRATION_DATE_FORMAT = '%Y-%m-%d'     #B
EXPIRATION_NUMBER_OF_DAYS = 7    #A

class DefaultTags():
   def __init__(self, environment, long_term=False):
       self.tags = {
           'customer': 'community',
           'automated': True,
           'cost_center': 123456,
           'environment': environment
       }
       if environment != 'prod' and not long_term:    #C
           self._set_expiration()    #C

   def get(self):
       return self.tags

   def _set_expiration(self):
       expiration_date = (    #A
           datetime.datetime.now() +    #A
           datetime.timedelta(    #A
               days=EXPIRATION_NUMBER_OF_DAYS)    #A
       ).strftime(EXPIRATION_DATE_FORMAT)     #B
       self.tags['expiration'] = expiration_date
```

#A Calculate the expiration date for seven days from the current date.
#B Format the date to a string of year, month, and day.
#C Set the expiration tag if you do not create the resource in production or mark it as a long-term resource.

You set one week as a default because it gives team members enough time to develop and test a resource. They can always renew another week if needed by running their delivery pipeline to update the tag automatically. However, you do need to enable an override to allow long-term resources in testing environments.

How do you enforce expiration date tagging by default but exempt resources from an expiration date? You can create a policy test with soft mandatory enforcement. A soft mandatory policy makes exceptions and audits long-term resources in testing environments.

Let’s write a test that enforces the expiration tag for each server resource. If the server does not exist in the list of exempt resources, it fails the test and stops the delivery pipeline from deploying all changes to production.

Listing 12.10 Test checks testing resources have an expiration date

```
import pytest


def test_all_nonprod_resources_should_have_expiration_tag(
       servers, server_exemptions):    #A
   noncompliant = []
   for name, values in servers.items():
       if 'expiration' not in values['labels'].keys() and \    #B
               name not in server_exemptions:    #C
           noncompliant.append(name)    #C
   assert len(noncompliant) == 0, \
       'all nonprod resources should have ' + \
       f'expiration tag, {noncompliant}'
```

#A Retrieve a list of servers in configuration and those exempt from the policy.
#B Check if the tag for expiration exists in server tags.
#C If you did not exempt the server, you must flag the server as non-compliant with the policy.

Adding the resource to an exemption list means that your teammates will carefully examine which resources persist in the testing environment. During peer review (Chapter 7), you can identify any new, persistent resource based on changes to the exemption list. A single source of persistent resources in a testing environment ensures that you can audit and discuss cost control early in the development process.

After implementing the expiration tag in infrastructure as code, you need to write a script that runs daily. Figure 12.10 shows the script's workflow. It checks if the expiration date matches the current date. If so, the automation deletes the resource.

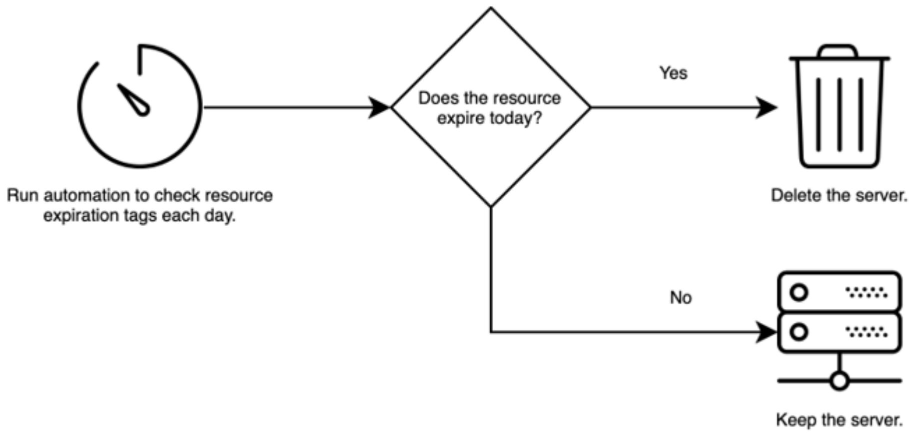

Run automation to check resource expiration tags each day.
Does the resource expire today?
Yes
Delete the server.
No
Keep the server.

Figure 12.10. Setting an expiration tag allows daily automation to verify whether or not to delete a temporary resource and thus reduce cost.

Why set an expiration date using infrastructure as code? The workflow of setting an expiration date using a tag module *builds in* the ability to renew the resource expiration date!

Rather than introduce development friction by adding separate automation, you build renewals into the development process.

For example, if a team still needs the resource, they can always re-run their infrastructure as code delivery pipeline to reset the expiration date for another seven days. Active changes to the resource also resets the expiration date. If you change your infrastructure, you probably still need the resource.

What happens when a resource expires and you still need it? You can always re-run your infrastructure as code and create a new one. Using infrastructure as code to add and renew the expiration date provides cost compliance and functional visibility across teams.

Infrastructure as code versus automated tagging

Sometimes, you'll find separate automation for automatic tagging. The automation adds the expiration date to infrastructure resources after their creation. While the automatic tagging means greater control for cost compliance, it also introduces drift between the actual and intended configurations. Furthermore, automatic expiration often confuses team members. Unless they paid attention to your communications, they might find their resource deleted after a few days!

You can always choose the expiration time interval to something other than several days. If you want to offer more flexibility to teams, you can offer a range of days through the tag module. I recommend calculating the absolute expiration date and adding it to the tag instead of the time interval for easier cleanup automation.

With all of the changes in the example, what happened to your client's cloud computing bill? Your bill went from a little over $10,000 to about $6,500 (a 35% reduction)! Your client appreciates the efficient use of their cloud resources.

In reality, you might not achieve the same dramatic cost reduction as the example. However, you can always apply the practices and techniques to your infrastructure as code to introduce minor changes that reduce cost when possible. Capturing cost reduction practices in infrastructure as code with tests ensures that everyone writes it with the constraint of cost in mind.

Exercise 12.2

Imagine you have three servers. You examine their utilization and notice the following:

- You need one server to serve the minimum traffic.
- You need three servers to serve the maximum traffic.
- The server handles traffic twenty-four hours a day, seven days a week.

What can you do to optimize the cost of the servers for the next month?

A. Schedule the resources to stop on the weekends.
B. Add an `autoscaling_policy` to scale based on memory.
C. Set an expiration of three hours for all servers.
D. Change the servers to smaller CPU and memory machine types.
E. Migrate the application to containers and pack applications more densely on servers.

Find answers and explanations at the end of the chapter.

12.3 Optimize cost

You can apply other infrastructure as code principles and practices to reduce cloud waste and manage cost drivers. In figure 12.11, techniques like building environments on-demand, updating routes between regions, or testing in production can use infrastructure as code to further optimize cost.

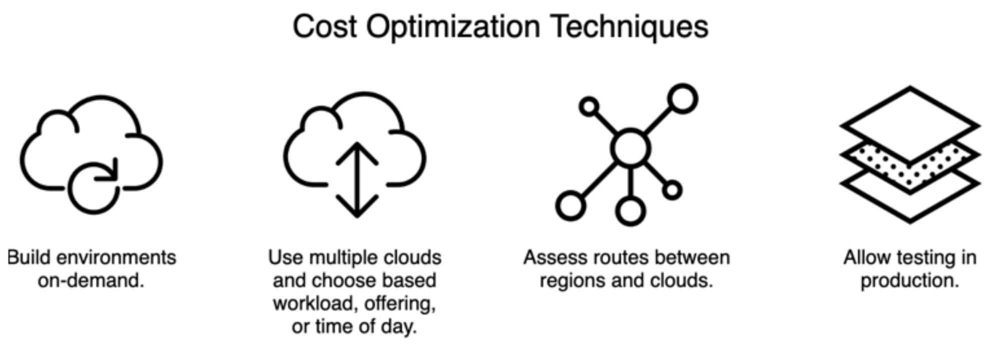

Cost Optimization Techniques
Build environments on-demand.
Use multiple clouds and choose based workload, offering, or time of day.
Assess routes between regions and clouds.
Allow testing in production.

Figure 12.11. Cost optimization requires infrastructure as code practices for scaling and deploying infrastructure resources.

In particular, the principles of reproducibility, composability, and evolvability can help with creative techniques to further optimize cost. These techniques include reproducing environments on-demand can reduce persistent testing environments, composing infrastructure across cloud providers, and evolving production infrastructure across regions and cloud providers.

Recall that you reduce your client's cloud computing cost for a conference by 35%. A year later, the finance team asks for your help to optimize the cost of their platform. They've grown their business and want to optimize costs across hundreds of customers and a managed service.

12.3.1 Build environments on-demand

From a broader perspective, you need to examine which environments exist in testing and production. You might start by reducing cloud waste across all environments. However, as your company grows, it adds more environments to support and test more products.

Imagine you examine your client's infrastructure. They have many different testing environments. You determine that three or four of them run continuously and support specialized testing. For example, the quality assurance (QA) team uses one of the environments for performance testing twice a year. For the rest of the year, these environments remain dormant.

You decide to *remove the continuously running environments*. If the QA team wants an environment for performance testing, they can do it on-demand. They copy the production environment with its factory and builder modules that allows inputs. The modules provide flexibility to specify variables and parameters for different environments.

Figure 12.12 shows the rest of the workflow for creating an on-demand environment. The QA team copies the infrastructure as code to a new repository specific to the testing environment in the organization's multi-repository structure. They update the parameters and variables, run their tests, and remove the environment.

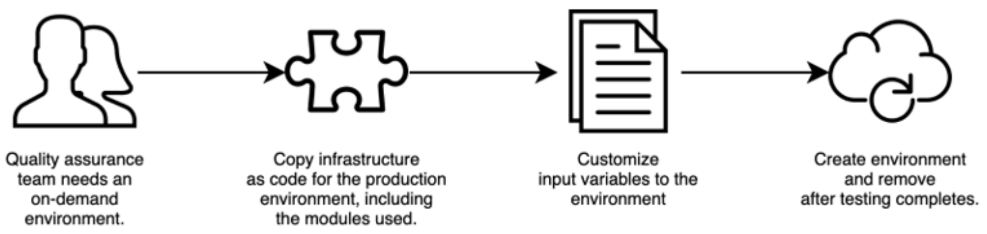

Quality assurance team needs an on-demand environment.
Copy infrastructure as code for the production environment, including the modules used.
Customize input variables to the environment
Create environment and remove after testing completes.

Figure 12.12. You can copy the configuration for production to create and customize on-demand environments for testing.

Why use reproducibility to create new environments on-demand and delete them? A new, updated environment ensures that the latest configuration matches production. If you only use the environment once a year, you do not want to keep it constantly running for elevent months.

While it takes some time to create a new environment, you probably take the same amount of time trying to fix drift between your testing and production environment. Identifying unnecessary long-running environments and switching them to an on-demand model can help mitigate some costs, especially if you can easily recreate them.

12.3.2 Use multiple clouds

With a few cloud provider options, you may also consider deploying to other clouds and optimizing cost based on resource, workload, and time of day. Infrastructure as code can help standardize and organize your configuration for multiple clouds. Deploying to multiple clouds can accommodate specialized workloads or teams that want specific infrastructure resources.

For example, imagine your client uses Google Cloud Dataflow for stream data processing. However, the cost varies depending on the type of pipeline. You convince some reporting teams to convert a few of the batch processing pipelines to Amazon Elastic MapReduce (EMR) to reduce the overall cost.

In figure 12.13, the report service team switches their infrastructure as code to use the Amazon EMR module. To minimize the disruption of jobs, they use the blue-green deployment pattern from Chapter 9 to gradually increase the number of jobs they run in Amazon EMR.

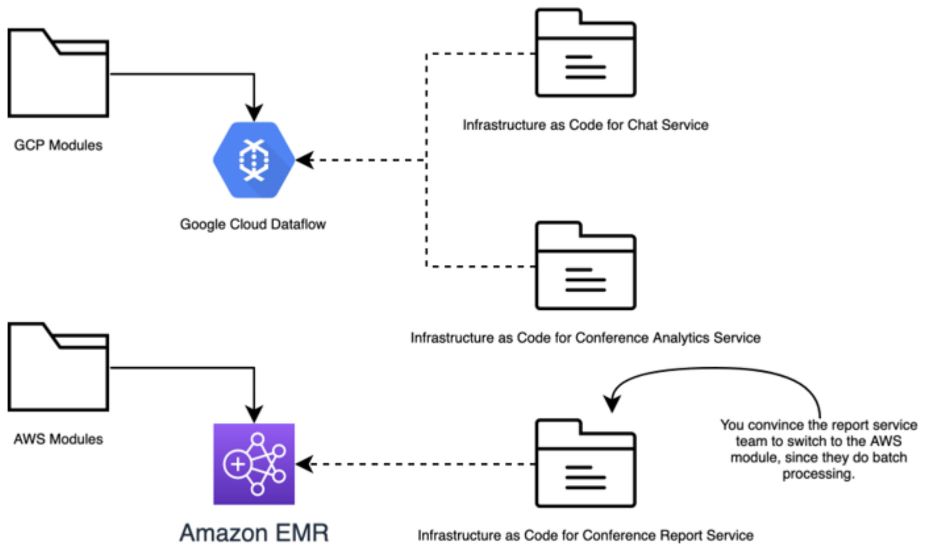

GCP Modules
Google Cloud Dataflow
Infrastructure as Code for Chat Service
Infrastructure as Code for Conference Analytics Service
AWS Modules
Amazon EMR
Infrastructure as Code for Conference Report Service
You convince the report service team to switch to the AWS module, since they do batch processing.

Figure 12.13. The report service switches its batch-processing job to use Amazon EMR instead of Google Cloud Dataflow by referencing a different module.

The principle of composability becomes an important part of a multi-cloud configuration. Infrastructure as code makes it easier to manage and identify infrastructure resources across different cloud providers. Using modules to express dependencies between clouds also helps you evolve your resources over time.

In Chapter 5, we separated infrastructure as code configuration into different folders based on tools and providers. Many infrastructure as code tools do not offer a unified data model for cloud provider resources. I recommend different modules for each cloud your plan to support because it reduces the complexity and supports isolated testing of modules. Maintaining separate modules for each cloud provider also enables easy identification of infrastructure resources and providers.

12.3.3 Assess data transfer between regions and clouds

With the adoption of multiple clouds, you might discover that you didn't reduce your overall cloud computing bill. You need to consider multiple clouds carefully because providers will charge for data transfer between regions and outside their cloud network. Data transfer costs add up in surprising ways!

You check your client's cloud computing bill and notice that a lot of the cost comes from data transfers across regions and out of the network. After some investigation, you discover that many of the services and testing environments communicate across regions, availability zones, and over the public internet.

For example, integration tests for the chat service in "us-central1-a" use the public IP address of the user profile service in "us-central1-b"! You realize that all of the services in the integration testing environment do not need to test across regions, availability zones, or out of the network.

Integration testing only tests the functionality of the service relative to other services and not the system itself. Figure 12.14 uses the refactoring techniques from chapter 10 to consolidate the infrastructure resources in the integration testing environment into one availability zone.

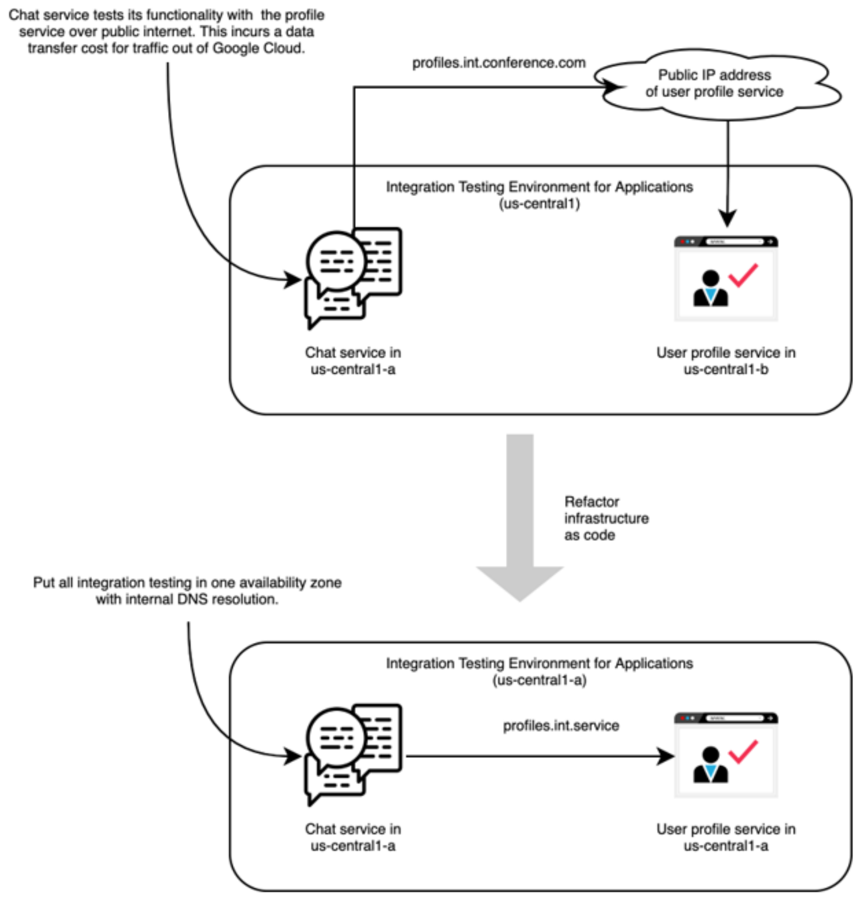

Chat service tests its functionality with the profile service over public internet. This incurs a data transfer cost for traffic out of Google Cloud.
profiles.int.conference.com
Public IP address of user profile service
Integration Testing Environment for Applications (us-central1)
Chat service in us-central1-a
User profile service in us-central1-b
Refactor infrastructure as code
Put all integration testing in one availability zone with internal DNS resolution.
Integration Testing Environment for Applications (us-central1-a)
profiles.int.service
Chat service in us-central1-a
User profile service in us-central1-a

Figure 12.14. Refactor your infrastructure as code to support an integration testing environment in a single availability zone and resolve to a private IP address.

What happens if the availability zone fails? You can always switch the infrastructure as code to a different zone or region. The applications still communicate over a private network and do not charge for data transfer out of Google Cloud's network or between regions and zones.

Favor private over public networking, not just for security but also for cost and efficiency. If you use multiple clouds, know which resources need to communicate across clouds. Sometimes, you might find it more cost-efficient to shift an entire set of services to another cloud than pay for data transfer between clouds. Applying the change and refactoring techniques in Chapters 9 and 10 can help consolidate services and communications.

12.3.4 Test in production

Even after shifting to multiple clouds and optimizing data transfer, you might find that your testing environment cannot fully mimic production and costs too much to run. At some point, you cannot get away with testing in an isolated environment. Rather than simulate the production environment in its entirety, you can continue to optimize costs by testing directly in the production environment.

In the case of the conference platform, you help the video service team implement changes to test in production. In figure 12.15, the team stages a new set of infrastructure resources with the expected changes hidden behind a feature flag. Then, they toggle the flag to direct all applications and user traffic and verify its functionality in production. The two sets of resources run simultaneously for a few weeks. After a few weeks, they delete the old infrastructure resources.

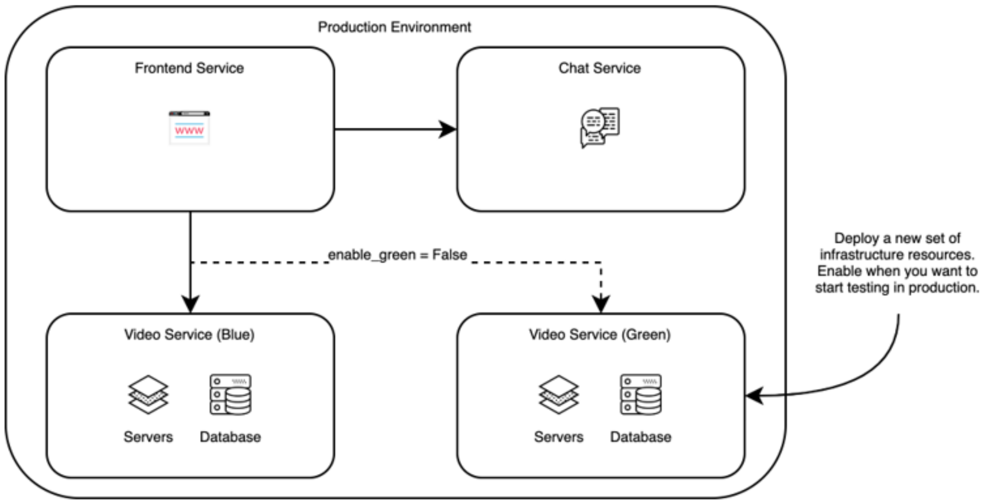


Figure 12.15. Testing in production for infrastructure as code applies blue-green deployments and feature flagging for a set of resources.

The team tested their service in production using blue-green services without using a testing environment. ***Testing in production*** involves a set of practices that enable you to run tests against production data and systems.

> **DEFINITION** Testing in production describes a set of practices that allow you to run tests against production data and systems.

In software development, you use techniques like feature flagging to hide certain functionality in production for testing or canary deployments to identify if you want to test it with a small group of users first before offering it to everyone on the platform.

For infrastructure as code, testing in production does not fit entirely with the software development practices. You don't want to test if it works with a small group of users. You just want to know if you create broken infrastructure systems that might *affect* the users! You can apply a few techniques, like feature flagging and canary deployment. We did this in chapters 9 and 10.

You could test infrastructure as code directly in production without a blue-green deployment pattern or feature flagging. However, you will need an established roll-forward plan in case of failure. I worked in one organization that depended entirely on local testing before pushing changes to production. If our change failed, we would attempt to update the system to a previous state. If all else failed, we would create a whole new infrastructure environment and direct all application and user traffic to the new environment.

You might go through all cost optimization, cloud waste reduction, and cost driver control techniques and *still* never fully optimize your cloud bill! Over time, your organization's usage and product demands change.

If you have proper monitoring and instrumentation in your systems, you might discover that you have periods of greater or lower demand. For example, your client has the most demand for their platform in May, June, October, and November for peak conference season.

Elasticity and re-architecture.

You might need to re-architect your systems to take advantage of public cloud ***elasticity***, the ability to scale up or down, and reduce cost over time. Some software architectures do not make it easy for you to scale up or down resources dynamically. The solution often requires a refactor or re-platform of the application to improve system cost efficiency.

Understanding your system's resource usage and requirements over time can help you further optimize costs beyond the techniques I described in this chapter. You could reduce costs even further by negotiating a contract with the infrastructure provider. Choosing a new pricing model lets you save on a certain number of reserved instances or volume-based discounts.

The longer you run the system, the more metrics you aggregate. The information can help you iterate on cost driver control, cloud waste reduction, and overall cost optimization in infrastructure as code. You can also use the information to negotiate with your cloud provider and reduce the surprises on your cloud computing bill!

12.4 Exercises and Solutions

Exercise 12.1

Given:

```
HOURS_IN_MONTH = 730
MONTHLY_BUDGET = 5000
DATABASE_COST_PER_HOUR = 5
NUM_DATABASES = 2
BUFFER = 0.1

def test_monthly_budget_not_exceeded():
  total = HOURS_IN_MONTH * NUM_DATABASES * DATABASE_COST_PER_HOUR
  assert total < MONTHLY_BUDGET + MONTHLY_BUDGET * BUFFER
```

A. Which are true? (Choose all that apply.)

A. The test will pass because the cost the database is within the budget.

B. The test estimates the monthly cost of databases.

C. The test does not account for different database types.

D. The test calculates the monthly cost per database instance.

E. The test includes a buffer of 10% for any cost overage as a soft mandatory policy.

Answer:

The answers are B, C, and E. The total estimated cost of the databases per month is $7,300, while the monthly budget with a 10% buffer is $5,500, which means the test fails. The test itself does not account for different database types or calculate monthly cost per database instance. It calculates based on the hourly rate and the number of databases. Adding a 10% buffer to the monthly budget creates a soft mandatory policy for the budget. Small changes that result in minor cost overage will reduce the friction of delivery to production but flag any major cost changes.

Exercise 12.2

Imagine you have three servers. You examine their utilization and notice the following:

- You need one server to serve the minimum traffic.
- You need three servers to serve the maximum traffic.
- The server handles traffic twenty-four hours a day, seven days a week.

What can you do to optimize the cost of the servers for the next month?

A. Schedule the resources to stop on the weekends.
B. Add an `autoscaling_policy` to scale based on memory.
C. Set an expiration of three hours for all servers.
D. Change the servers to smaller CPU and memory machine types.
E. Migrate the application to containers and pack applications more densely on servers.

Answer:

You can add an `autoscaling_policy` to scale based on memory (B). You cannot schedule a resource stop on weekends because you need at least one server on weekends. Setting an expiration or downsizing the servers do not help with cost in this situation. Migrating applications to containers involves a long-term solution that will not optimize cost for the next month.

12.5 Summary

- Before changing infrastructure as code to optimize cost, identify the cost drivers (resources or activities) that may affect the total cost.
- Manage cost drivers in infrastructure, such as compute resources, by adding policy tests to check for resource type, size, and reservation.
- Cost estimation parses your infrastructure as code for resource attributes and generates an estimate of their cost based on a cloud provider's API.
- You can add a policy test to check the output of cost estimation to approximate whether or not you exceeded your budget.
- Cloud waste describes unused or underutilized infrastructure resources.
- Eliminating cloud waste can help decrease your cloud computing cost.
- Reduce cloud waste by removing or stopping untagged or unused resources, starting and stopping resources on a schedule, right-sizing infrastructure resources for better utilization, enabling autoscaling, and tagging a resource expiration date.
- Autoscaling increases or decreases the number or amount of resources in a given group based on a metric, such as CPU or memory.
- Techniques for optimizing cloud computing cost involve building environments on-demand, using multiple clouds, assessing data transfer, and testing in production.
- Testing in production uses practices like blue-green deployments and feature flagging to test infrastructure changes without a testing environment.

13 Managing tools

This chapter covers

- **Assessing the use of open-source infrastructure modules and tools**
- **Applying techniques to update or migrate infrastructure as code tools**
- **Implementing module patterns for event-driven infrastructure as code**

You learned how to write infrastructure as code, update it with your team using delivery pipelines and testing, and manage its security and cost within your organization! As you evolve your infrastructure system, you adapt these patterns and practices and adjust them to fit new workflows and use cases. Similarly, tools change but should not disrupt the patterns and practices to scale, collaborate, and operate your infrastructure.

Updating your tool can involve several actions. You could upgrade to a new version, replace it with a new tool, or handle more dynamic use cases for infrastructure as code. This chapter discusses common patterns and practices to handle updates to infrastructure as code tools.

The patterns apply to any tools that cover *provisioning, configuration management, and image building* use cases. You'll find they also apply to software development, although I adapted them in an opinionated way to infrastructure. Use these patterns and practices to mitigate the blast radius of an update, scale the new tool across teams, and continue evolving your system to support business requirements.

Code examples in this chapter

This chapter does not include any code listings. Adding an example means introducing yet another tool. I describe the patterns and approaches at a higher level. You can apply the techniques to any tool that supports domain-specific languages (DSLs) or programming languages.

You read the many chapters in this book and practiced infrastructure as code in different industries and companies. You've built a reputation for setting up and scaling infrastructure as code practices. One day, a social media company offers you a role in their platform team.

The company has already established its infrastructure as code practice for a few years now. They need your help to maintain and keep the tools they use for infrastructure as code updated. You accept and get a backlog of projects to start on the very first day.

13.1 Open source tools and modules

The accessibility of version control and public repositories makes it simpler to search for an existing tool or infrastructure module instead of writing your own. You can go onto GitHub or any other service to find the automation and tooling to serve your needs. However, you need to make sure you do your due diligence before introducing it to your organization.

For example, let's imagine the team that maintains the social media's feed functionality approaches you. They searched online and found an infrastructure module to create a database. They want to use it. The team hopes to speed up their development process and avoid waiting for another team to review their database configuration. Why reinvent the wheel, after all?

You offer to help review the module for security and best practices. Before you introduce the module to your organization, you use figure 13.1 to assess the database module for its functionality, security, and lifecycle before officially adopting it in your organization.

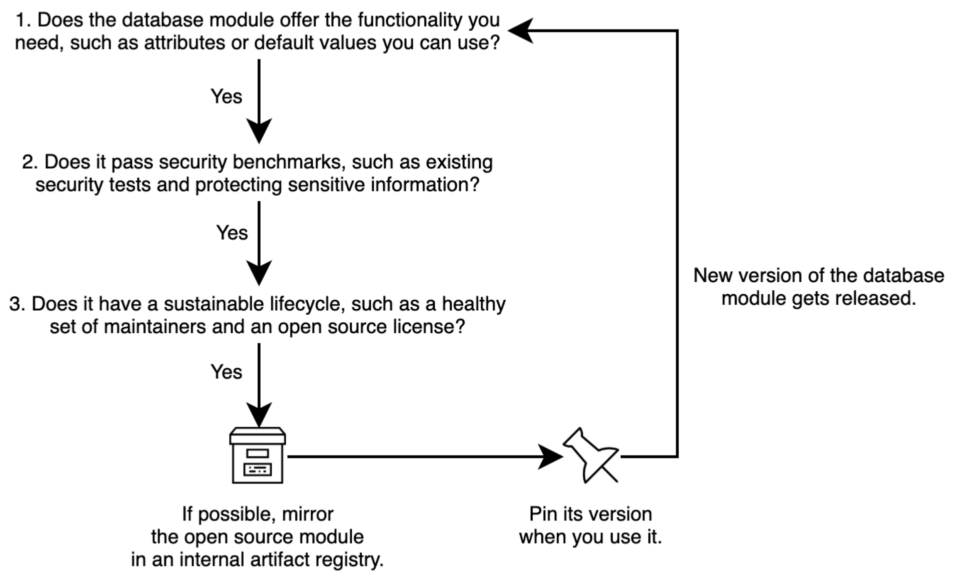

1. Does the database module offer the functionality you need, such as attributes or default values you can use?
Yes
2. Does it pass security benchmarks, such as existing security tests and protecting sensitive information?
Yes
3. Does it have a sustainable lifecycle, such as a healthy set of maintainers and an open source license?
Yes
If possible, mirror the open source module in an internal artifact registry.
Pin its version when you use it.
New version of the database module gets released.

Figure 13.1. You assess the functionality, security, and lifecycle of a tool or module before you use it.

Each time the open-source maintainers release a new database module, you reassess the module. You can apply this decision workflow to safely and securely adopt external infrastructures as code modules and tools. You want to make sure you can use the tool or module and avoid insecure configuration in your infrastructure system that would allow a bad actor to exploit your system.

13.1.1 Functionality

You may find a module or tool very promising to start. It allows you to configure the attributes you need very flexibly. However, recall from Chapters 2 and 3 that modules should include some opinionated defaults. Without them, a module or tool with too much flexibility may lead to one-off configurations that eventually break your system.

You encourage the feed team to verify the default values in the database module. In figure 13.2, the feed team assesses the database module. The module uses very opinionated defaults, pins the version of the database, and tests compatibility thoroughly.

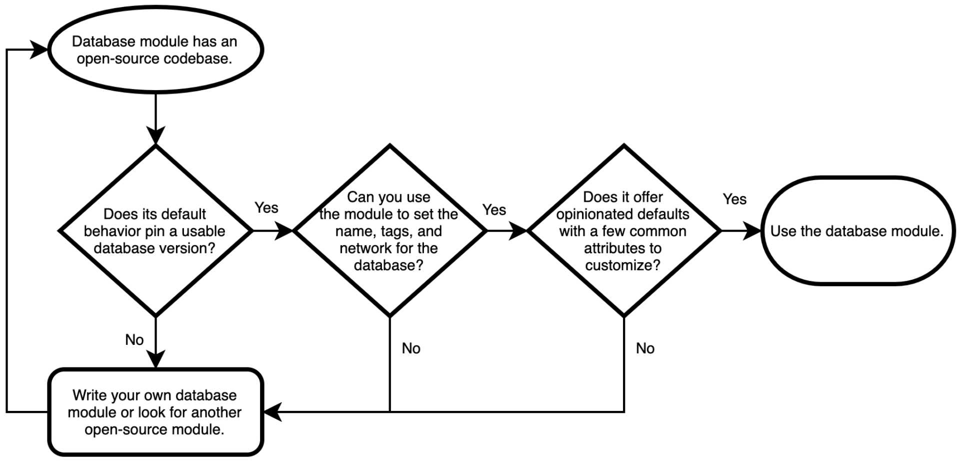

Database module has an open-source codebase.
Does its default behavior pin a usable database version?
Yes
Can you use the module to set the name, tags, and network for the database?
Yes
Does it offer opinionated defaults with a few common attributes to customize?
Yes
Use the database module.
No
No
No
Write your own database module or look for another open-source module.

Figure 13.2. Decision flow for assessing usage of module or tool based on functionality.

The documentation notes that the module pins the database version to test the configuration and its compatibility with specific versions thoroughly. The feed team confirms they use the database version and approves the module’s default value for it.

Next, they assess the input variables for the module. The database module allows them to set the attributes they need, such as database name. It also lets them set the tagging and network. The feed team confirms they do not need to set more than those variables.

Since the module does not offer every attribute as an input variable and offers opinionated defaults that fit the team, you approve the module based on functionality. In general, review the module’s documentation and commit history. If the module tests for version compatibility and a set of opinionated defaults that suit your functionality, you can move forward with a security assessment.

If you find the module does not offer some specific default values or input variables you need to change, you can find another module, write your own, or proceed to use the module with its limitations in mind. Similarly, you can apply the decision flow to a tool and find it lacking. A single tool cannot do everything! You must balance its flexibility in functionality with its predictability in making infrastructure changes.

13.1.2 Security

While you might first assess the functionality of a module or tool, you should next evaluate its security. Security tends to serve as the make-or-break criteria for whether or not you should use an open-source tool or module. Without carefully assessing an open-source module or tool for secure configuration or code, you may inadvertently open a door for a bad actor to exploit the system.

Before the feed team can use their database module, they need you to check the module for any security concerns. In figure 13.3, you check if the database module exposes or outputs sensitive information, sends information to third party endpoints, and passes existing security and compliance tests.

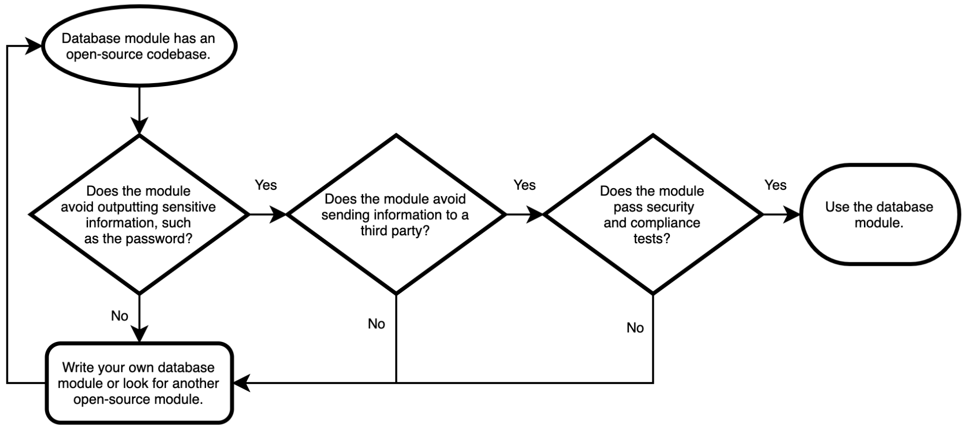

Database module has an open-source codebase.
Does the module avoid outputting sensitive information, such as the password?
Yes
Does the module avoid sending information to a third party?
Yes
Does the module pass security and compliance tests?
Yes
Use the database module.
No
No
No
Write your own database module or look for another open-source module.

Figure 13.3. Decision flow for assessing usage of module or tool based on security.

In the example, the database module does not output the password or any sensitive configuration or send information to a third party endpoint. The module also passes your security and compliance tests for databases you wrote in chapter 8. Why should you verify all three checks before adopting a module?

A module may accidentally expose or output sensitive information. For example, a configuration could accidentally output a password during a dry run. If it does, make sure you have a way to mitigate or mask the password and rotate it.

Similarly, a module should not write or send information to an unauthorized third party. A bad actor might add a minor configuration that sends your network information to an HTTP endpoint. Review each resource and check they do not send anything to a third party.

Security and open source

Software supply chain attacks occur when an actor includes malicious code in a vendor's software, which gets shipped out to customers and compromises their data and systems. The benefit of open source means that you can examine what goes into code before you, as a customer, decide to use it.

In this section, I recognize that I greatly oversimplified the risk assessments and guardrails for defending against supply chain attacks. For more information, a whitepaper by the National Institute of Standards and Technology (https://www.cisa.gov/sites/default/files/publications/defending_against_software_supply_chain_attacks_508_1.pdf) better organizes some of the practices.

Finally, run existing security and compliance tests you wrote for the infrastructure resources. You want *secure* and *compliant* resources. Otherwise, you need to go back and update the module to suit your requirements.

Deploy the module in your infrastructure as code in an isolated testing environment. Then run the security and compliance tests against the module. Isolating the module in its own environment ensures that you do not introduce non-compliant configurations to runnign environments.

Keep in mind that not all security and compliance tests will pass. Those that fail need minor refactoring to work with the module. After all the tests pass, you can approve the module as secure for use.

Tools follow a similar decision workflow as you assess their security. However, security and compliance testing for infrastructure as code tools may include static code analysis and additional reviews from your organization. For tools that output configuration or other information during dry runs, you will want to apply the remediation steps from Chapter 8.

13.1.3 Lifecycle

You examined the functionality and security of the module, but the compliance team raises a very good question. They ask about *who* maintains the public module. Your organization will need to take private (or maybe public) ownership of the module or tool if its maintainers no longer update it.

In figure 13.4, you examine the documentation to understand the lifecycle of the module and who maintains it. If the database module has company sponsorship and an appropriate license, you can likely use it.

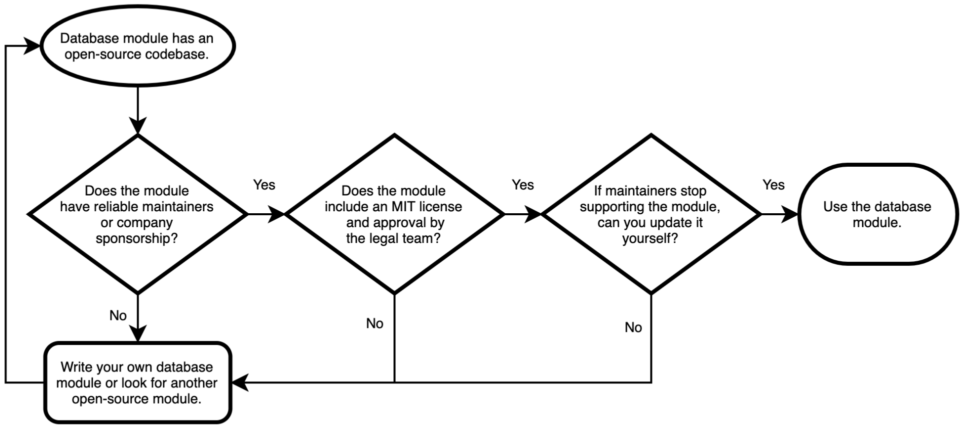

Database module has an open-source codebase.
Does the module have reliable maintainers or company sponsorship?
Yes
Does the module include an MIT license and approval by the legal team?
Yes
If maintainers stop supporting the module, can you update it yourself?
Yes
Use the database module.
No
No
No
Write your own database module or look for another open-source module.

Figure 13.4. Decision flow for assessing usage of module or tool based on tool or module lifecycle.

You examine the maintainers of the database module. The maintainers come from a well-known technology company and have many contributors, which means the project fosters an active community. They update and release a new version of the module every few months. Each contribution must pass a testing suite to verify that changes do not break the module.

Next, you retrieve information about the database module’s open source license. You do not know how the open source license affects your company, so contact your legal team. They review the license before the feed team can use it.

The module includes an MIT license. As a ***permissive open source license***, the MIT license means if you fork (maintain a copy) or modify the database module, you must include a copy of the license and the original copyright notice. If the maintainers deprecate the module or tool, the permissive license type allows you to modify and update the module yourself.

> **DEFINITION: A permissive open source license allows you to fork or modify code as long as you include a copy of the license and original copyright notice.**

Your legal team approves the license because the module poses minimal risk to the overall infrastructure configuration. The company can edit the module if needed but does not have to release it publicly. Perhaps you can even contribute to the open-source module yourself, pending legal approval.

A module or tool can also include an open-source license in the copyleft category. The ***copyleft*** category of licenses contains a clause that you must release the codebase with your modifications.

> **DEFINITION: A copyleft open source license allows you to fork or modify code as long as you release the codebase with your modifications.**

The copyleft category of licenses often includes more restrictions on modifying and distributing the tool or module. Your company's legal team will assess whether or not they can use open-source infrastructure as code with more restrictive licenses.

Open source licenses

For more information on open source licenses, review the licenses and standards outlined by the Open Source Initiative (https://opensource.org/licenses).

The feed team (overjoyed) can use the database module. You recommend the team pins the module version by mirroring it to an internal artifact registry. Mirroring the module ensures that teams can only use the approved module in the internal registry. If the public endpoint for the module goes down, they always have a copy in the internal registry. Each time the maintainers release a new module version, you must check the changes and approve the latest version.

Consider contributing back to open source if your organization allows it. Forking an open-source module or tool and maintaining it yourself while keeping it updated with the public release has an operational overhead. You can dedicate many hours reconciling changes between the open-source and your version. Creating a process to contribute changes directly to the public release helps reduce the overhead of maintaining unique changes that may break your infrastructure.

13.2 Upgrading tools

When you run your infrastructure as code practice for a few years, you inevitably reach a point where you need to upgrade your tool or the plug-ins it uses. A wider gap in your tool version to the latest version can make it much harder for you to update your infrastructure with minimal disruption!

We learned about this challenge for module versioning in Chapter 5. I'll cover some of the considerations and patterns you can use when you need to upgrade your tool.

The challenge of upgrading

You will not find magic tools to migrate everything perfectly for you. Upgrading a tool will always have challenging obstacles. Unique patterns in your infrastructure as code (such as inline scripting) may break the system during the upgrade! As a result, try to limit the complexity of your infrastructure as code logic when possible.

Imagine you audit the company's infrastructure as code tools (provisioning, configuration management, and image building). Most of the infrastructure as code uses version 1.7, while the tool just released its latest version at 4.0. Many of them lag two or three versions behind the latest. Your first major project involves updating them.

13.2.1 Pre-upgrade checklist

Before you start upgrading a tool or plugin, you need to check a few essential practices that will help minimize potential disruption to your infrastructure. Your checklist should include a few steps to decouple, stabilize, and reconcile infrastructure as code.

Figure 13.5 shows this checklist. You decouple all dependencies, check versioning, pin all versions, and deploy your infrastructure as code to reduce drift.

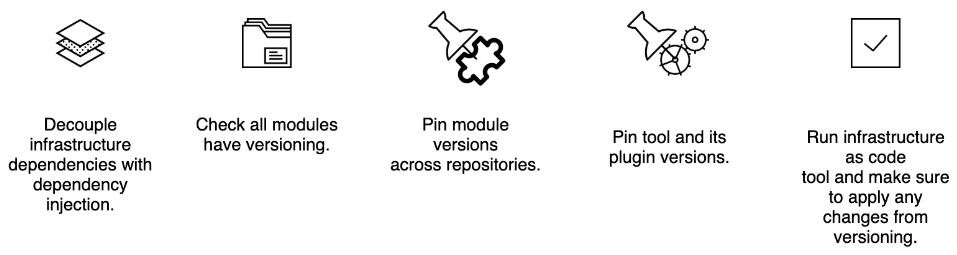

Decouple infrastructure dependencies with dependency injection.
Check all modules have versioning.
Pin module versions across repositories.
Pin tool and its plugin versions.
Run infrastructure as code tool and make sure to apply any changes from versioning.

Figure 13.5. Before you upgrade a tool, your checklist should include pinning and checking all versions for modules, plug-ins, and tools.

If the tool upgrade adds or removes fields, you need to pass the correct information expected by each resource subset. Dependency injection offers a layer of abstraction for configuration attributes between infrastructure resources (refer to Chapter 4). It protects each subset from changes to the other.

You check that you added module versioning to all infrastructure as code modules (Chapter 5). Similarly, make sure any infrastructure as code or repositories using the module pins to a specific version.

For example, one of the teams at the social media company always uses the latest version of the module. You add the module version of 2.3.1 to their repository. When you release module version 3.0.0 with the tool upgrade, their infrastructure as code will not upgrade with breaking changes. You could update a module and push a breaking change to everyone using it without pinning the module version!

You also verify that each team pins the current version of the tool and its plugin versions. While the plugins may not have forward compatibility, you want to preserve the current version and avoid adding a new configuration in a different version. Finally, you push all your pinned module, tool, and plugin versions to each infrastructure as code module and configuration. You ensure the version pinning does not introduce new changes.

After you complete your pre-upgrade checklist, you must plan your tool upgrade path. Figure 13.6 shows that upgrading from 1.0 to 4.0 of the tool introduces some breaking changes! You decide you can upgrade to 3.0, which provides backward compatibility with 1.7. Then you can upgrade from 3.0 to 4.0, which should reduce any other breaking changes.

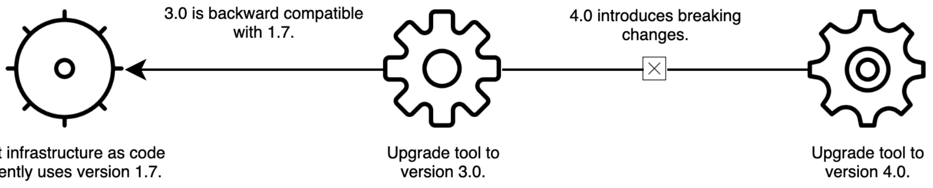

3.0 is backward compatible with 1.7.
4.0 introduces breaking changes.
Most infrastructure as code currently uses version 1.7.
Upgrade tool to version 3.0.
Upgrade tool to version 4.0.

Figure 13.6. You plan your tool upgrade path and account for backward-compatible versions and versions with breaking changes.

Do not upgrade to the latest version immediately! Instead, examine a list of breaking changes in your tool and assess whether or not you can accommodate those changes. I avoid upgrading more than two versions (or subversions in case of beta releases) at one time. Most tools have breaking changes in either behavior or syntax.

Consider testing the upgrade in a testing environment before rolling it out to production. Based on your system and its testing environment, you can identify if a tool upgrade may disrupt infrastructure. Upgrading will never go as smoothly as you expect, and a testing environment helps identify significant issues before you upgrade production.

13.2.2 Backward compatibility

Many infrastructures as code tools offer some kind of backward compatibility for changes. They typically support old and new features for a version or two before deprecating the old feature. Even if a tool supports backward compatibility, make sure to port and refactor to new features as soon as you can!

Your example upgrades the tool from version 1.7 to 3.0. Fortunately, version 3.0 does support backward compatibility with 1.7. It offers new features but no breaking changes that would affect your infrastructure as code. Just in case, you take a careful approach to the upgrade.

You start with the feed team since they agree to your help during the upgrade. The feed team deploys all changes and makes sure they do not add new changes to infrastructure as code. Then, you examine the configuration for the best way to upgrade the tool without disrupting the social media feed.

In figure 13.7, you apply the refactoring techniques from chapter 10 to upgrading the tool. You start with high-level resources because other resources do not depend on them, deploy changes, test the system, and upgrade the lower-level resources.

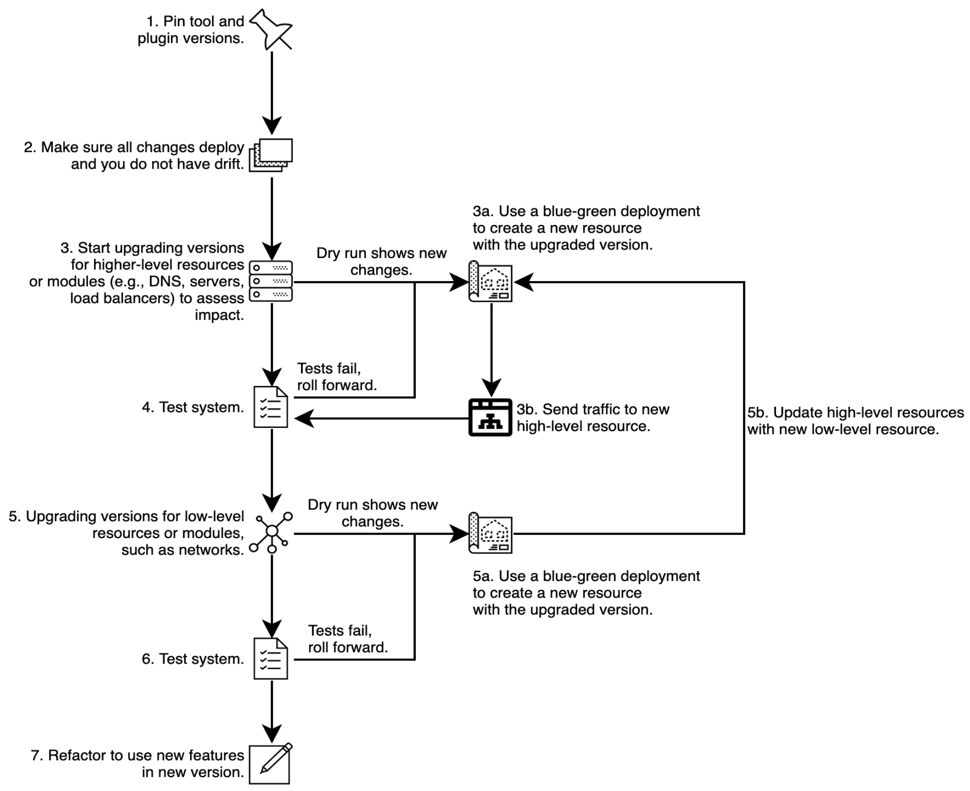

1. Pin tool and plugin versions.
2. Make sure all changes deploy and you do not have drift.
3. Start upgrading versions for higher-level resources or modules (e.g., DNS, servers, load balancers) to assess impact.
Dry run shows new changes.
3a. Use a blue-green deployment to create a new resource with the upgraded version.
Tests fail, roll forward.
4. Test system.
3b. Send traffic to new high-level resource.
5b. Update high-level resources with new low-level resource.
5. Upgrading versions for low-level resources or modules, such as networks.
Dry run shows new changes.
5a. Use a blue-green deployment to create a new resource with the upgraded version.
Tests fail, roll forward.
6. Test system.
7. Refactor to use new features in new version.

Figure 13.7. Apply refactoring techniques to upgrade a backward-compatible tool version from high-level to low-level resources.

You notify the feed team that you will upgrade DNS and load balancer infrastructure resources first. Other resources do not depend on them. Updating them first allows you to test whether or not your upgrade patterns work. You can upgrade the tool version for the resource by changing the version and running the infrastructure as code. You check the dry run and tests in your delivery pipeline to ensure you did not disrupt anything.

The high-level resources like DNS and load balancer update without any disruption. Next, you move to lower-level resources. You start to update the servers with the rolling update pattern from Chapter 10. Rather than upgrade all servers simultaneously, you start the upgrade on one and quickly run into a problem.

The production server configuration has an override script that breaks when you upgrade the tool. Fortunately, you only affected one server since you used a rolling update. After all, you want to keep the social media feed running and available.

In figure 13.8, you apply the roll-forward techniques from Chapter 11 to fix the server that no longer works. You implement a manual fix and debug the old server for the problem.

Once you fix the problem in the testing environment, you push out a change to the override script and proceed with your rolling update of servers.

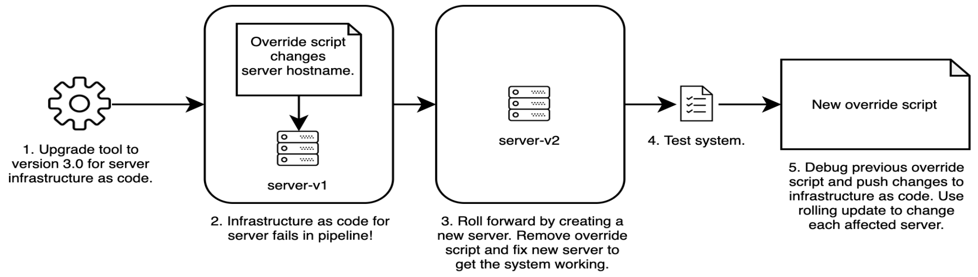

Override script changes server hostname.
server-v1
server-v2
New override script
1. Upgrade tool to version 3.0 for server infrastructure as code.
2. Infrastructure as code for server fails in pipeline!
3. Roll forward by creating a new server. Remove override script and fix new server to get the system working.
4. Test system.
5. Debug previous override script and push changes to infrastructure as code. Use rolling update to change each affected server.

Figure 13.8. You can use a rolling update pattern to minimize the blast radius of a failed tool upgrade and roll forward if the change fails.

The servers pass the test. You move to the lowest-level resource, the network. Just in case, you deploy a second version of the network using a blue-green deployment pattern from Chapter 9. After deploying everything onto the new network, you run all the end-to-end tests and check that the system still works. You completed the tool upgrade!

Why revisit patterns like refactoring, blue-green deployment, and rolling upgrades? You want to minimize the blast radius of a potential failure. Many of these patterns seem repetitive, but they offer a structured, less risky approach to upgrading your system. Changing your tool *does* change your infrastructure, so you can apply very similar techniques with the same result.

In general, use rolling upgrades for infrastructure as code upgrades to servers or other compute resources. Blue-green deployment helps tool upgrades for high-risk, low-level infrastructure resources. You can usually update high-level resources in-place.

However, you can't prevent every failure. In that case, use the roll-forward practices and patterns from Chapter 11 if the system breaks. High-level resources can revert the configuration in-place, while lower-level resources benefit from creating new resources with previous changes.

13.2.3 Breaking changes in upgrades

Every once in a while, you'll find your infrastructure tool or plugin releases a new version that includes breaking changes. New versions with breaking changes often happen with early versions of a tool, such as when the tool handles new or edge use cases. If you need to do a tool upgrade with breaking changes or features, use the techniques for making changes in Chapter 9.

For the social media company, you upgraded the feed team's tool from version 1.7 to 3.0. However, upgrading from version 3.0 to 4.0 involves some breaking changes! Version 4.0 contains backend schema changes that may affect your resources. How can you update your infrastructure to version 4.0 without impacting the system?

Remember in Chapter 4, I mentioned the existence of the *tool state*. Tools keep a copy of the infrastructure state to detect drift between actual resource state and configuration and track which resources it manages. Tool state differs from the actual infrastructure state. When you update a tool, you want to break out the old tool state from the new one.

In our example, you want to isolate the old tool state with version 3.0 from a new tool state with version 4.0. Separating the tool state minimizes the blast radius of a potential failure by isolating the infrastructure resources a tool needs to change. Fewer resources means faster recovery and potentially less impact to other parts of the system.

In figure 13.9, every team in the social media company separates their tool state into a different location. Separate tool states ensures that changes to the feed team's blue infrastructure does not affect the green infrastructure.

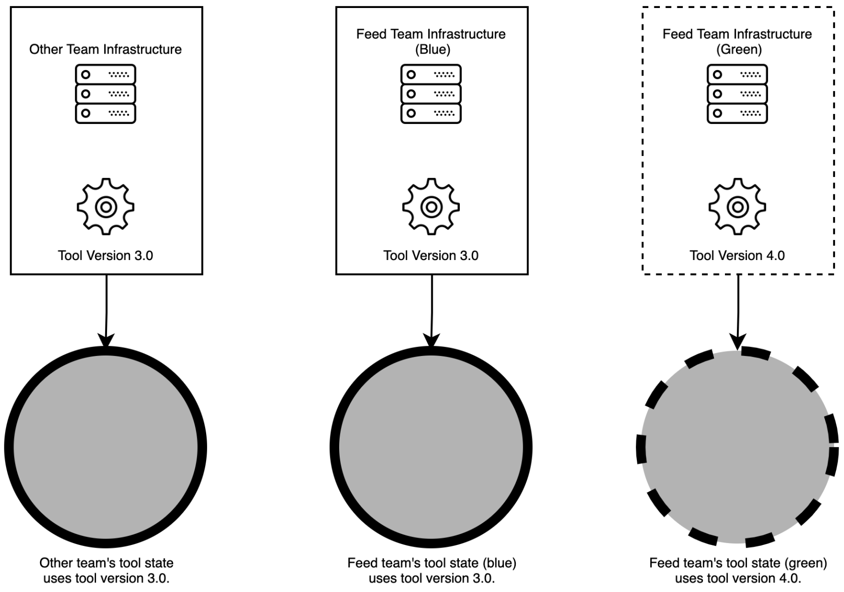

Other Team Infrastructure
Tool Version 3.0
Feed Team Infrastructure (Blue)
Tool Version 3.0
Feed Team Infrastructure (Green)
Tool Version 4.0
Other team's tool state uses tool version 3.0.
Feed team's tool state (blue) uses tool version 3.0.
Feed team's tool state (green) uses tool version 4.0.

Figure 13.9. A tool state captures the infrastructure state for the tool to make comparisons and can exist in separate locations for the tool to use.

First, you pin tool and module versions across the feed team's infrastructure as code. Next, you update infrastructure modules to tool version 4.0. You release a new version of the module, making a note of breaking changes.

Next, you copy the existing configuration into a new folder. Each new folder creates a new tool state. Find the network folder and create it using tool version 4.0. You should now have the original "blue" resources from tool version 3.0 and the new "green" resources from tool version 4.0.

You applied a ***blue-green deployment strategy to the tool state*** to create new resources with the new tool! Creating a new set of resources with a new version ensures that any breaking changes will not affect the existing infrastructure.

> **DEFINITION**: Blue-green deployment for tool state is a pattern that creates a new subset of infrastructure resources with a new tool version. You gradually shift traffic from the old set of resources (blue) to the new set of resources (green). The pattern isolates breaking changes to the new set of resources for testing.

After creating the low-level resources, copy the high-level resources to a new folder. Update its dependencies to use the low-level resources from tool version 4.0. After all, you want all resources using the new tool version.

Figure 13.10 summarizes the strategy. Create the high-level resources, run your tests, and send traffic to the new resources.

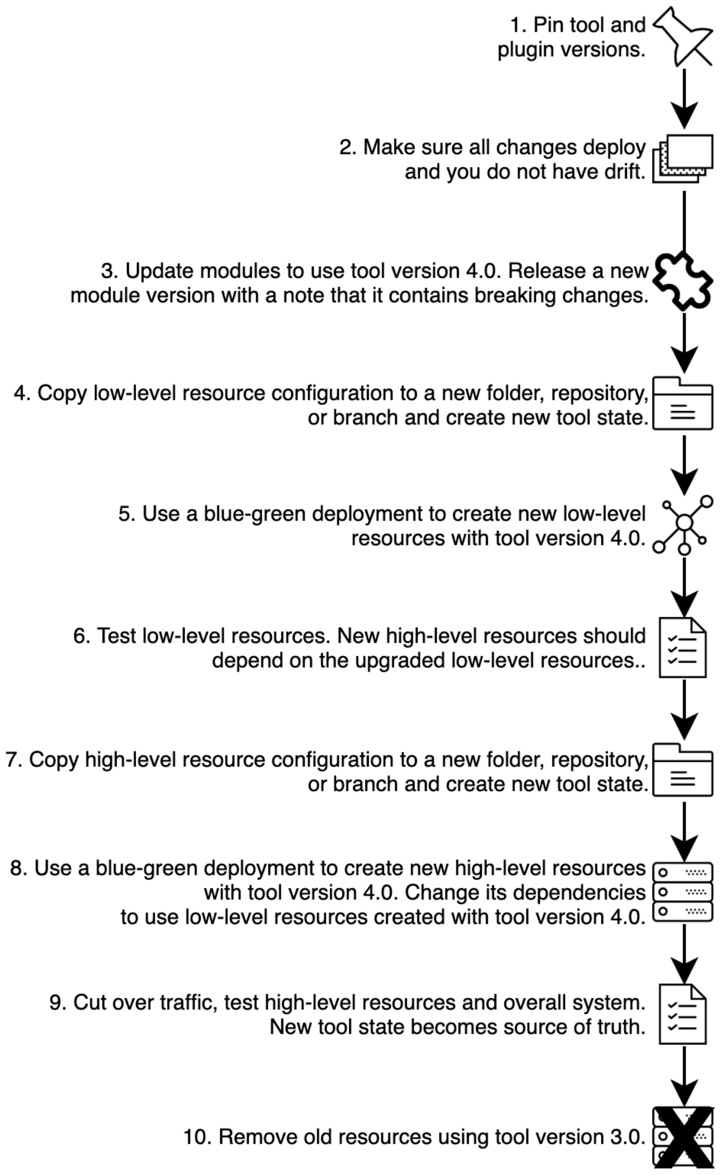

1. Pin tool and
plugin versions.
2. Make sure all changes deploy
and you do not have drift.
3. Update modules to use tool version 4.0. Release a new
module version with a note that it contains breaking changes.
4. Copy low-level resource configuration to a new folder, repository,
or branch and create new tool state.
5. Use a blue-green deployment to create new low-level
resources with tool version 4.0.
6. Test low-level resources. New high-level resources should
depend on the upgraded low-level resources..
7. Copy high-level resource configuration to a new folder, repository,
or branch and create new tool state.
8. Use a blue-green deployment to create new high-level resources
with tool version 4.0. Change its dependencies
to use low-level resources created with tool version 4.0.
9. Cut over traffic, test high-level resources and overall system.
New tool state becomes source of truth.
10. Remove old resources using tool version 3.0.

Figure 13.10. With breaking changes, consider creating a new resource in a different state and cutting over traffic to the resource with the latest version of the tool.

This approach differs from the blue-green deployment in Chapter 9. You create a new set of resources *and* an entirely new tool state. If you loosely coupled your dependencies, subsets of resources can have different tool versions without affecting the system's functionality.

Recall that you can work on infrastructure as code with different repository structures (Chapter 5) and branching models (Chapter 7). Depending on your repository structure and branching model, you can isolate your tool state differently. You can always merge a new branch and alter its pipeline to deploy infrastructure or copy the separate folder into your main configuration for a repository.

Suppose your organization has an opinionated approach to delivery pipelines and only allows production deployment on the main branch. In that case, you can create a new repository for the tool upgrade and archive the old one.

You can apply an in-place upgrade if you dry run and test the changes thoroughly. If it fails, I create an entirely new resource in a new tool state with the upgraded tool. When in doubt, refer to a tool's upgrade documentation. As a general practice, I will try an in-place upgrade if the tool offers some kind of migration tool or script to ease the update.

13.3 Replacing tools

I attempted to keep this book as tool-agnostic as possible but offer concrete examples to demonstrate the patterns and practices. I recognize I will probably have to update this book and replace all the tools with the latest and greatest technology! When running infrastructure as code for a while, you inevitably change tools for improved functionality or vendor support. What patterns should we use when migrating to a new tool?

Much of the patterns and practices in this book should help protect your system from these changes. Scoping and modularizing your infrastructure with patterns from Chapter 3 and decoupling your dependencies with patterns from Chapter 4 allows your teams to use tools that fit their use case and replace them as needed. If you do not have these patterns in place, you will encounter some difficulties migrating to a new tool.

Imagine completing your tool upgrades across the infrastructure as code for the social media company. You think about taking a break when the network team asks for your help. They want to move from a vendor's domain-specific language (DSL) to an open-source DSL. Their configuration will need an additional review from the social media feed team, which does not know anything about the vendor.

Your research cannot find a vendor or open-source script to facilitate a straightforward migration. You wished you had something that could translate the vendor's DSL to the open-source one. Without it, you and the network team need to proceed with the migration carefully.

13.3.1 New tool supports import

You could not find automation to "translate" between tools. However, you can apply patterns in this book to migrate across tools. Some tools support an import capability to add new resources to a tool's state. You can use the practices in Chapter 2 to migrate existing resources to the new tool.

In figure 13.11, you upgrade the modules to use the new open-source DSL. You also update its tests, which pass. To start, identify some low-level resources you can change. You create a separate folder, branch, or repository to isolate the new DSL from the vendor DSL. After writing the configuration for the new DSL, you import existing resources into the state of the new DSL.

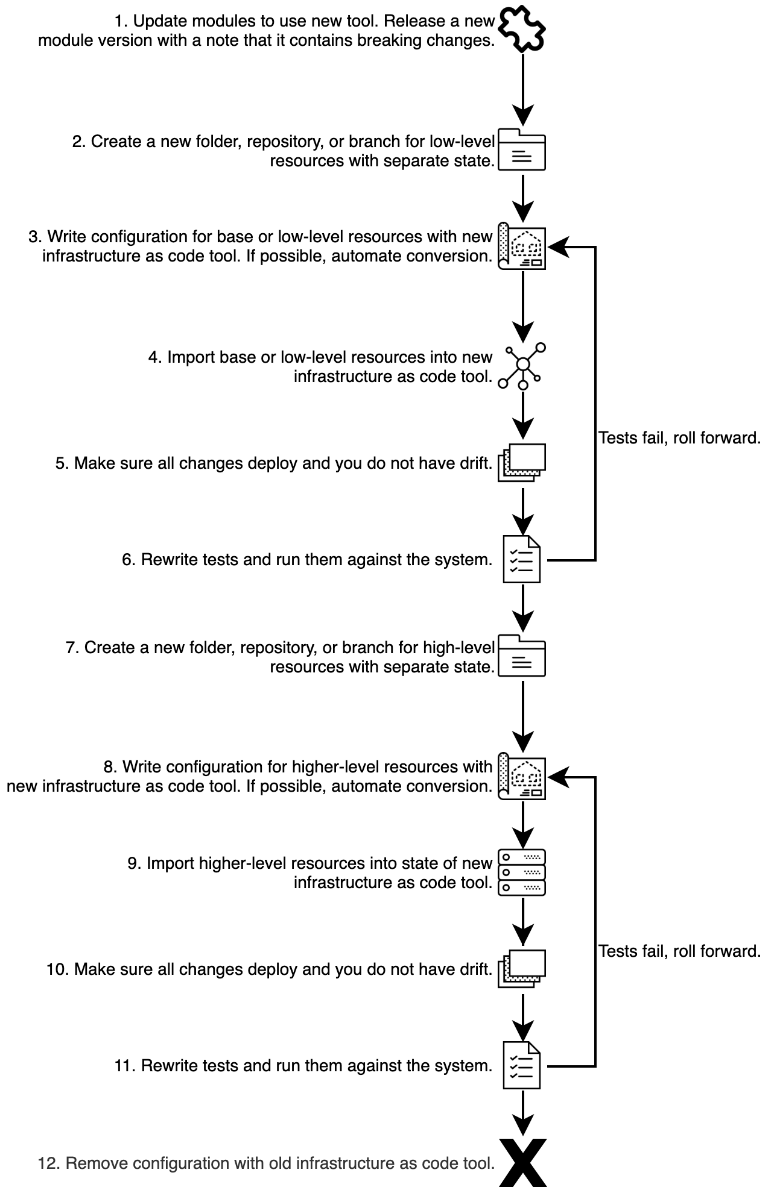

1. Update modules to use new tool. Release a new module version with a note that it contains breaking changes.
2. Create a new folder, repository, or branch for low-level resources with separate state.
3. Write configuration for base or low-level resources with new infrastructure as code tool. If possible, automate conversion.
4. Import base or low-level resources into new infrastructure as code tool.
5. Make sure all changes deploy and you do not have drift.
6. Rewrite tests and run them against the system.
Tests fail, roll forward.
7. Create a new folder, repository, or branch for high-level resources with separate state.
8. Write configuration for higher-level resources with new infrastructure as code tool. If possible, automate conversion.
9. Import higher-level resources into state of new infrastructure as code tool.
10. Make sure all changes deploy and you do not have drift.
11. Rewrite tests and run them against the system.
Tests fail, roll forward.
12. Remove configuration with old infrastructure as code tool.

Figure 13.11. A new tool that supports importing resources allows you to migrate to a new tool without changing existing resources.

Once again, you rewrite the tests to test the new syntax of the open-source DSL. They pass, and you proceed to write configuration and import high-level resources. Finally, you delete the infrastructure as code.

Throughout each cycle of writing new infrastructure as code with the open-source DSL, you want to check your dry run and rewrite your tests. The dry run shows if the defaults for the new tool do not match your existing state. You need to update the new infrastructure as code if they do not match and fix the drift.

13.3.2 No import capability

Some tools do not support import capabilities, and you need to create new resources for the new tool. Imagine if the network team asked you to help them convert from one vendor DSL to another. However, the new vendor DSL does not allow the import of existing resources into its state.

If your new tool does not support importing existing resources, you need to recreate resources with a blue-green deployment strategy for the tool state. In figure 13.12, you start by writing new infrastructure as code for the low-level resources and refactor tests for the new tool. As you repeat the process and finish migrating high-level resources, you cut over traffic and test the entire system.

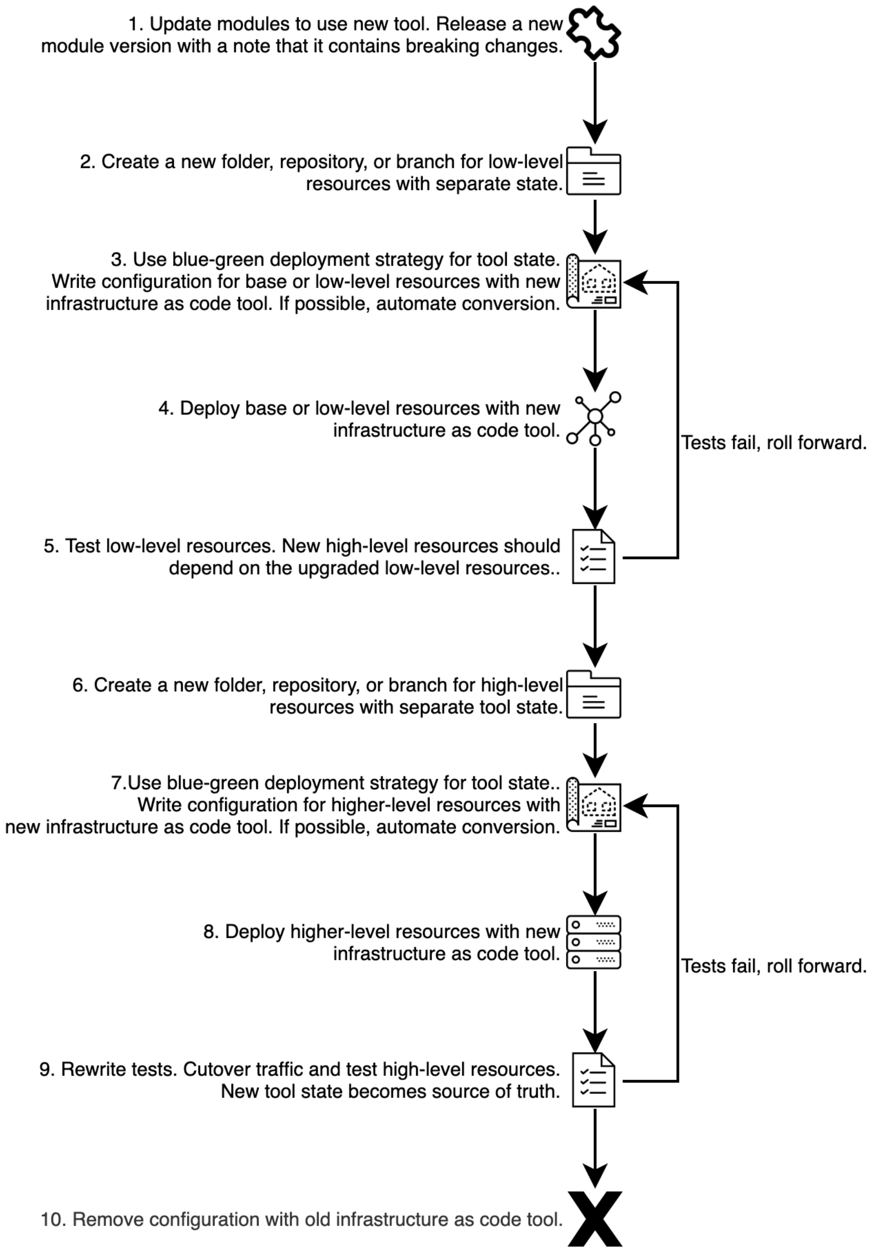

1. Update modules to use new tool. Release a new module version with a note that it contains breaking changes.
2. Create a new folder, repository, or branch for low-level resources with separate state.
3. Use blue-green deployment strategy for tool state. Write configuration for base or low-level resources with new infrastructure as code tool. If possible, automate conversion.
4. Deploy base or low-level resources with new infrastructure as code tool.
5. Test low-level resources. New high-level resources should depend on the upgraded low-level resources..
Tests fail, roll forward.
6. Create a new folder, repository, or branch for high-level resources with separate tool state.
7.Use blue-green deployment strategy for tool state.. Write configuration for higher-level resources with new infrastructure as code tool. If possible, automate conversion.
8. Deploy higher-level resources with new infrastructure as code tool.
9. Rewrite tests. Cutover traffic and test high-level resources. New tool state becomes source of truth.
Tests fail, roll forward.
10. Remove configuration with old infrastructure as code tool.

Figure 13.12. A new tool that cannot import resources requires a tool migration using a blue-green deployment strategy.

The pattern of migrating tools remains consistent, import capability or none. However, migration without an import capability takes more effort because you recreate the system. Even if you had a magical script to migrate from one tool to another, you might consider applying some of these patterns and practices to avoid breaking critical infrastructure resources, like the network.

You will always replace or add tools to your infrastructure ecosystem. Your organization will choose the tools that fit its architectural objectives. Applying the techniques to modularize, isolate, and manage infrastructure as code will accommodate your infrastructure as code evolution. I always return to the practices and patterns to make infrastructure, infrastructure as code, module, tool, and organizational changes and mitigate their risk to critical systems.

Regardless of whether or not a tool has import capabilities, you need to rewrite the tests for each resource you refactor. Figure 13.13 shows that you must refactor the unit and contract tests for each module and subset of resources as you migrate. However, your end-to-end and integration tests may stay the same.

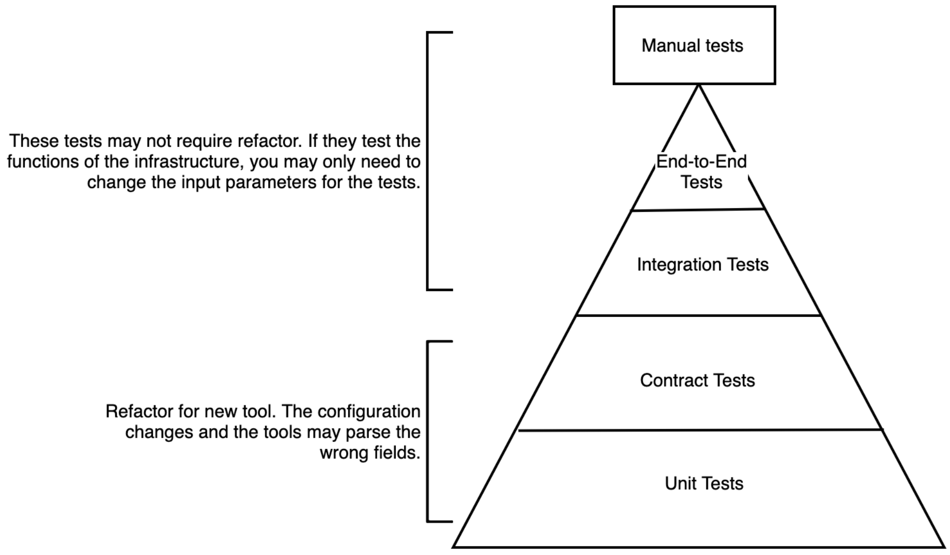

Manual tests
End-to-End Tests
Integration Tests
Contract Tests
Unit Tests
These tests may not require refactor. If they test the functions of the infrastructure, you may only need to change the input parameters for the tests.
Refactor for new tool. The configuration changes and the tools may parse the wrong fields.

Figure 13.13. You will need to rewrite the unit and contrasts tests when you upgrade a tool, while the integration, end-to-end, and manual tests may stay mostly the same.

A new tool will affect unit and contract tests because the tool uses a different state and metadata format. Tests cannot parse the correct information from the new tool. Integration and end-to-end tests likely stay the same because they evaluate the functions of infrastructure and not the tool itself.

Refactoring the tests as you go along allows you to add more tests, remove redundant tests, or make updates to broader security or policy tests. You should update your manual, integration, and end-to-end tests with new input parameters because you have different resources. However, the tests themselves do not change too much because they test the system's functionality, not infrastructure attributes.

13.4 Event-driven infrastructure as code

Most of the book covers the practice of collaborating and writing infrastructure as code to reduce the impact of a potential failure of critical systems. Once you get familiar with the principles and practices, you can extend them to more dynamic use cases.

For example, a development team wants some very dynamic automation with their infrastructure as code. Whenever the development team deploys a new instance of an application, the team needs to update a firewall rule to allow access from the instance to the database. Rather than push a new instance and then remember to update the firewall rule later, the team wants some automation to run an infrastructure module to configure the firewall rule after an application instance starts.

Figure 13.14 demonstrates the automation you implement. You deploy a new application with a new IP address. An automated script captures the new IP address and runs some infrastructure as code. The configuration updates the firewall with the new IP address. This automation repeats each time you deploy a new application with a new IP address.

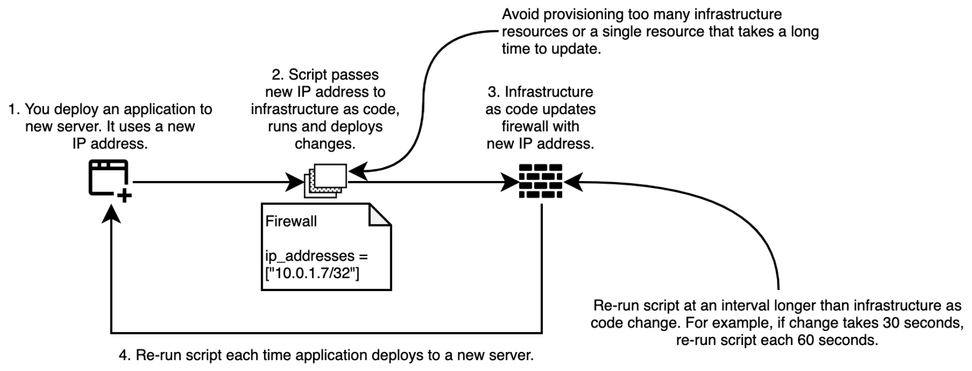

Avoid provisioning too many infrastructure resources or a single resource that takes a long time to update.
2. Script passes new IP address to infrastructure as code, runs and deploys changes.
1. You deploy an application to new server. It uses a new IP address.
3. Infrastructure as code updates firewall with new IP address.
Firewall
ip_addresses = ["10.0.1.7/32"]
Re-run script at an interval longer than infrastructure as code change. For example, if change takes 30 seconds, re-run script each 60 seconds.
4. Re-run script each time application deploys to a new server.

Figure 13.14. Whenever an application instance gets a new IP address, the infrastructure module updates the firewall rule with the new IP address.

You may consider running an infrastructure module automatically when the system changes. ***Event-driven infrastructure as code*** means running a minimally-scoped infrastructure module to configure infrastructure in response to an event. You can use the automation to update another resource or fix the system based on the event.

DEFINITION: Event-driven infrastructure as code runs a minimally-scoped infrastructure module to configure infrastructure in response to an event.

Updating the application equals an event. Some script, application, or automation detects the event and responds by running an infrastructure module! You can write your own or find an open-source tool to identify and respond to the event. Some real-world automation you can use to detect and react to an event includes an operator for Kubernetes, a serverless function, or an application that consumes from an event queue.

Isn't this GitOps?

In Chapter 7, I mentioned that a number of practices and patterns in this book lean toward GitOps. GitOps combines declarative configuration, drift detection, version control, and continuous deployment. The approach achieves event-driven infrastructure as code. I think of GitOps as a subset of event-driven infrastructure as code because its automation responds to events in configuration drift.

If the GitOps framework detects drift, it runs automation to reconcile the configuration. For example, the container orchestrator Kubernetes uses controllers to automatically reconcile the declarative configuration with resource state. However, event-driven infrastructure as code describes infrastructure as code automated by a broader set of events, not just drift.

The use case for event-driven infrastructure as code has become more prevalent with dynamic services and applications. If you do use event-driven infrastructure as code, keep a few practices in mind.

- Avoid infrastructure resources that take a long time to create or configure. You do not want to include an infrastructure resource that takes an hour to create.
- Do not add too many resources to an event-driven module. Otherwise, you will take a long time to create so many instances.
- Balance the time it takes to run a change with the module versus the time interval between events.
- Combine the module practices from Chapter 2 with the testing patterns from Chapter 6 to verify that event-driven infrastructure as code runs quickly and correctly.

Some events happen with high frequency. You need infrastructure that deploys faster than the event's frequency or an automation script to batch changes at a specific interval. You should choose the most minimal subset of infrastructure resources to deploy with event-driven infrastructure as code.

From static infrastructure as code that you deploy based on code commits to dynamic infrastructure as code run for an event, you apply the same patterns, practices, and principles to managing and collaborating. Your objectives, your team's requirements, and your organization's business will evolve and change over time - hopefully, your infrastructure as code practice grows with it. Keep in mind your testing strategy, the cost of infrastructure, and maintaining its security and compliance.

13.5 Summary

- Review the functionality, security, and lifecycle of an open-source tool or module before adopting it in your organization.
- An open-source tool or module should have default values or behaviors that offer predictability and stability in changes. Otherwise, you will have to write a layer of code to add opinionated default attributes within your organization.
- Protect your infrastructure from supply chain attacks by scanning an infrastructure tool or module for security, checking for third-party data collection, and running your security and compliance tests.
- The type and number of maintainers and the license associated with the tool or module also affect your organization's use.
- Open-source tools and modules can have two categories of licenses: permissive and copyleft.
- A permissive license lets you modify and update the module or tool as long as you include a copy of the license and its original copyright notice.
- A copyleft license allows you to modify and update the module or tool as long as you include a copy of the license, its original copyright notice, and release your copy open source.
- Before upgrading your infrastructure as code tool, decouple infrastructure dependencies and pin module, plugin, and tool versions.
- Start a tool update with backward compatibility by refactoring high-level resources mutably and proceeding to low-level resources.
- A blue-green deployment strategy for tool state means creating a new set of infrastructure resources with a state separate from the existing configuration.
- Tool state refers to a copy of the infrastructure state that an infrastructure as code tool uses to detect drift or resources under its management.
- Start a tool update with breaking changes by applying a blue-green deployment strategy to the tool state, starting from low-level to high-level resources.
- When replacing one tool with another, use the new tool's import capability to migrate existing resources to the new tool. Start with low-level resources and apply them to high-level resources. Then, remove the configuration for the old tool.
- If a new tool does not have an importing feature for existing infrastructure resources, you will need to apply a blue-green deployment strategy for the tool state.
- Event-driven infrastructure as code runs a minimal infrastructure as code module in response to a system event to automate infrastructure changes.
- Ensure that a module for event-driven infrastructure as code deploys quickly with as few resources as possible.