Chapter 1. Why Terraform

Software isn't done when the code is working on your computer. It's not done when the tests pass. And it's not done when someone gives you a "ship it" on a code review. Software isn't done until you *deliver* it to the user.

Software delivery consists of all of the work you need to do to make the code available to a customer, such as running that code on production servers, making the code resilient to outages and traffic spikes, and protecting the code from attackers. Before you dive into the details of Terraform, it's worth taking a step back to see where Terraform fits into the bigger picture of software delivery.

In this chapter, you'll dive into the following topics:

- What is DevOps?
- What is infrastructure as code?
- What are the benefits of infrastructure as code?
- How does Terraform work?
- How does Terraform compare to other infrastructure-as-code tools?

What Is DevOps?

In the not-so-distant past, if you wanted to build a software company, you also needed to manage a lot of hardware. You would set up cabinets and racks, load them up with servers, hook up wiring, install cooling, build redundant power systems, and so on. It made sense to have one team, typically called Developers ("Devs"), dedicated to writing the software, and a separate team, typically called Operations ("Ops"), dedicated to managing this hardware.

The typical Dev team would build an application and "toss it over the wall" to the Ops team. It was then up to Ops to figure out how to deploy and run that application. Most of this was done manually. In part, that was unavoidable, because much of the work had to do with physically hooking up hardware (e.g., racking servers, hooking up network cables). But even the work Ops did in software, such as installing the application and its dependencies, was often done by manually executing commands on a server.

This works well for a while, but as the company grows, you eventually run into problems. It typically plays out like this: because releases are done manually, as the number of servers increases, releases become slow, painful, and unpredictable. The Ops team occasionally makes mistakes, so you end up with *snowflake servers*, wherein each one has a subtly different configuration from all the others (a problem known as *configuration drift*). As a result, the number of bugs increases. Developers shrug and say, "It works on my machine!" Outages and downtime become more frequent.

The Ops team, tired from their pagers going off at 3 a.m. after every release, reduce the release cadence to once per week. Then to once per month. Then once every six months. Weeks before the biannual release, teams begin trying to merge all of their projects together, leading to a huge mess of merge conflicts. No one can stabilize the release branch. Teams begin blaming one another. Silos form. The company grinds to a halt.

Nowadays, a profound shift is taking place. Instead of managing their own datacenters, many companies are moving to the cloud, taking advantage of services such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Instead of investing heavily in hardware, many Ops teams are spending all their time working on software, using tools such as Chef, Puppet, Terraform, Docker, and Kubernetes. Instead of racking servers and plugging in network cables, many sysadmins are writing code.

As a result, both Dev and Ops spend most of their time working on software, and the distinction between the two teams is blurring. It might

still make sense to have a separate Dev team responsible for the application code and an Ops team responsible for the operational code, but it's clear that Dev and Ops need to work more closely together. This is where the *DevOps movement* comes from.

DevOps isn't the name of a team or a job title or a particular technology. Instead, it's a set of processes, ideas, and techniques. Everyone has a slightly different definition of DevOps, but for this book, I'm going to go with the following:

> *The goal of DevOps is to make software delivery vastly more efficient.*

Instead of multiday merge nightmares, you integrate code continuously and always keep it in a deployable state. Instead of deploying code once per month, you can deploy code dozens of times per day, or even after every single commit. And instead of constant outages and downtime, you build resilient, self-healing systems and use monitoring and alerting to catch problems that can't be resolved automatically.

The results from companies that have undergone DevOps transformations are astounding. For example, Nordstrom found that after applying DevOps practices to its organization, it was able to increase the number of features it delivered per month by 100%, reduce defects by 50%, reduce *lead times* (the time from coming up with an idea to running code in production) by 60%, and reduce the number of production incidents by 60% to 90%. After HP's LaserJet Firmware division began using DevOps practices, the amount of time its developers spent on developing new features went from 5% to 40%, and overall development costs were reduced by 40%. Etsy used DevOps practices to go from stressful, infrequent deployments that caused numerous outages to deploying 25 to 50 times per day, with far fewer outages.[1]

There are four core values in the DevOps movement: culture, automation, measurement, and sharing (sometimes abbreviated as the acronym CAMS). This book is not meant as a comprehensive overview of DevOps (check out Appendix A for recommended reading), so I will just focus on one of these values: automation.