Model-Driven DevOps

Increasing agility and security in your physical network through DevOps

STEVEN CARTER | JASON KING
with MIKE YOUNKERS and JOSH LOTHIAN

Model-Driven DevOps

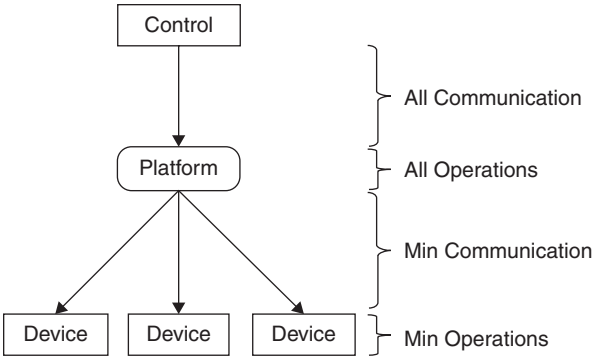

FIGURE 3-7 Better Scale Through Platforms

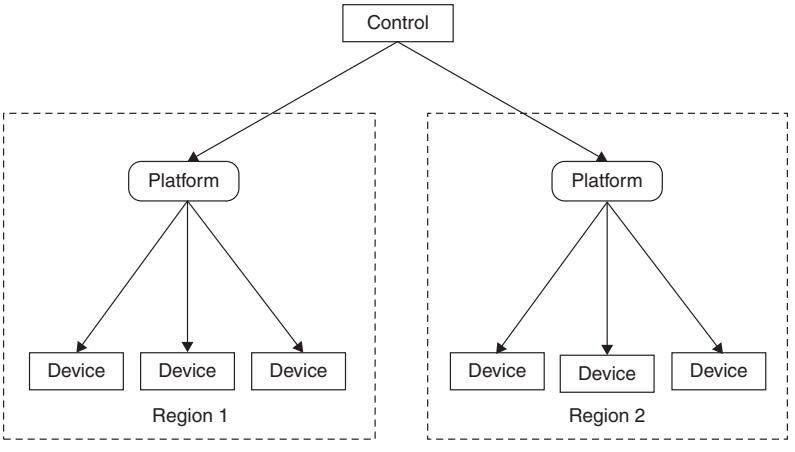

FIGURE 3-8 Scaling Platforms Geographically

Summary

This chapter covered the two main factors for successful infrastructure automation: APIs and platforms. We defined consumable infrastructure and demonstrated how the APIs coupled with data models discussed in the preceding chapter enable it. Furthermore, we illustrated how APIs enable deterministic and efficient machine-to-machine interactions and are critical for successful automation. Platforms enable you to better scale operations made via API, provision physical hardware, and enable you to build more complex automated services on top of your infrastructure. We also examined what makes cloud platforms so powerful and how those same platform concepts can be applied to on-prem infrastructure. In the following chapters, we cover how to represent your infrastructure as code and then how to take some of the high-level things you have learned so far and assemble them into a framework for infrastructure automation at scale.

Chapter 4

Infrastructure as Code

In the preceding chapter, you learned how to make infrastructure consumable via APIs and platforms to reduce the complexity of automation. With consumable infrastructure as its foundation, this chapter introduces the concept of infrastructure as code (IaC) and illustrates how it can largely be viewed as a data movement, translation, and validation exercise rather than a programming exercise. The many code snippets in this and the following chapter help illustrate the concepts that we are presenting. They are simplified versions of a more complete reference implementation presented in Chapter 6, "Implementation," which is also available via GitHub.

Compliance Is a Dirty Word

It was Monday, and Bob was sitting in his ACME Corp office, staring at his ticket queue. The queue was long and depressing, so he turned his thoughts to the future, the CIO's DevOps mandate, and what it meant for the network team. He was beginning to formulate some thoughts in this area, and they were somewhat encouraging. The first was that APIs really are the new CLI, and the second was that he needed a platform to help scale APIs beyond simple test cases. What he had learned so far was encouraging. However, at this stage, it seemed as though the team was a long way from achieving anything like real DevOps. The problem was that he did not know where to start. Frustrated, he turned his thoughts back to his ticket queue and the seemingly endless series of manual tasks that needed his attention.

Bob groaned audibly as he read the title of the first ticket, "Prepare network infrastructure for audit." ACME Corp was subject to several different IT compliance standards, such as PCI and ISO. Each year an audit was done to ensure ACME Corp met these standards. If the company did not, the consequences could be severe, ranging from fines to inability to operate. A failed audit could often lead to lost revenue and, potentially, a resume-generating event for people like Bob.

For Bob, *compliance* was a dirty word. The network infrastructure was notorious for being out of compliance. Like most organizations, ACME Corp documented its compliance standards as a series of CLI or GUI commands in a Microsoft Word document. People like Bob took the various standards such as PCI and ISO, interpreted them, combined them to achieve an "approved" compliant configuration, and then documented that configuration as a series of CLI commands or GUI instructions in a Word doc. A human was then required to follow the document and manually verify actual device configurations against the approved configurations. Because this was an incredibly time-consuming exercise, and therefore costly, it happened only once a year, just prior to an audit.

Bob understood the need for compliance, but he also knew that ACME Corp's operating model meant that the network was almost immediately out of compliance the moment an audit was over. Because he and the rest of the network team are human, and they make changes by hand, they sometimes configured things differently from the standard, took shortcuts, or even made mistakes. When you added up all these factors, it meant that the actual network configuration could drift significantly from a "compliant" configuration, and the more time that went by between audits, the further it would drift.

Bob thought to himself, "What if, instead of writing our standards as a set of instructions for humans to follow, we could represent them in a more machine-readable format that could be more readily used for automation? It sounds great, but, what would that look like?" From his previous attempts at automation, he learned that CLI-based templates are difficult to create because the CLI is intended for human consumption, and they are difficult to maintain because the CLI is prone to change with OS upgrades. What he needed was something machine-readable and stable.

Although APIs provide some of the attributes he was looking for, his previous experience showed that each different device API required different structured data as input, which meant that he could not simply represent the ACME Corp compliance standards as one set of structured data suitable for input to an API. Instead, he needed a different set of structured data for each API, and this didn't seem much different from having to maintain different CLI templates for each type of device.

Lately, there has been a lot of industry buzz around something called OpenConfig, which is supposed to solve this problem. After an hour of googling OpenConfig, Bob discovered that it is essentially a set of vendor-agnostic data models intended for network device configuration and monitoring. Another few hours of googling data models and Bob came to understand that a unified data model that could describe all common network configuration-related tasks was exactly what he was after. If he could define the ACME Corp compliance standards in terms of the OpenConfig data model, then, theoretically, he would have a set of structured data that he could use to automate the compliance check across the entire network.

Bob was starting to get excited about the notion of automating compliance, but, unfortunately, his excitement was short-lived. For OpenConfig to provide a way to configure the network using a common data model, all devices in the network would need to support OpenConfig. The good news was that many devices in the ACME Corp network *did* support it. The bad news was that some had only partial support. This sounded like a job for a platform. If Bob could use a platform to translate OpenConfig to his various devices that have no or only partial support for OpenConfig, then the dream might still be alive. "Maybe, just maybe, this might work," he thought to himself.

Later that day, Bob had lunch with Larry, one of his coworkers. Larry's team was responsible for performing network compliance checks prior to an audit. Bob laid out his thoughts regarding models and automating the compliance check. "So, what do you think?" asked Bob. Larry, acutely aware of how painful the process is, answered, "I like the idea. After performing the compliance check by hand a few hundred times, I think we lose focus, and our accuracy starts to diminish rapidly after that. This approach would solve that problem and save us a ton of time. But where would all the data come from?"

"What data?"

"Your OpenConfig model is great for specifying the organization of the configuration data, but something has to supply values for things like interface IP addresses and VLAN IDs for each device. Normally, we keep that data in a spreadsheet and verify that what is configured on each device matches the values in the spreadsheet."

"I see," said Bob. "The spreadsheet data is what *really* defines that specific device. Without it, we have only a generic model. When we combine the spreadsheet data with the model, we get a configured device and, ultimately, a configured network." Bob was starting to get it. This was really all about the data. Make the data compliant with the ACME Corp standards, and the resulting network would be compliant as well.

It occurred to Bob that a spreadsheet was probably not the best way to store the data that defined the network. To automate the process of transforming this spreadsheet data into network configuration, he would have to consider an alternative format that was more machine-readable. He thought this might be an opportunity for infrastructure as code. If he could convert the ACME Corp spreadsheet data into a machine-readable format and feed that into a platform presenting a unified device model for the network, he would be able to represent the entire network "as code."

The notion of infrastructure as code is getting a great deal of attention lately in IT circles, and it was a phrase that the CIO often used when she talked about DevOps. Bob sensed an opportunity here. Infrastructure as code might not be DevOps, but it was a step in the right direction, and it solved a real business problem. For the first time in a long time, Bob was enthusiastic about his job again. He couldn't wait to tell somebody about it.

As it happened, Jane, the network team manager, was in her office trying to figure out how to respond to the CIO DevOps mandate when Bob burst through her open door and said, "I have an idea!"

Why Infrastructure as Code?

The term *infrastructure as code (IaC)* is a part of the automation and DevOps conversation, but what does it mean? Using the term to define itself, IaC is the process of rendering the provisioning and configuration of infrastructure as code. But why? How does rendering configuration as code enable DevOps? Remember that DevOps started as a way to enable more agile software development. The tools used in DevOps operate on "code," so we need to render the infrastructure (that thing on which we are performing DevOps) as "code." "Code" is in quotes because not all of what we create will be actual program code. Much of what we talk about, in fact, is how to represent network configuration data in textual form. Representing infrastructure in textual form is important because this allows the use of source code managers (SCMs) like Git; however, as you see later, not all network configuration data is stored in textual form. Instead, some configuration data is better kept in a traditional database, and we discuss the reasons for this later.

What benefit does an SCM provide in the context of IaC? First and foremost, it is used to track changes to the state of the network. If the desired state of your network is defined as IaC, then an SCM allows you to track any modifications to that desired state. This capability allows you to know what changed and who changed it. Also, the SCM enables continuous integration and continuous deployment (CI/CD) by providing a hook to run the various stages of a CI/CD pipeline. We look more in depth at CI/CD in Chapter 5, "Continuous Integration/Continuous Deployment."

IaC means that we need to represent our infrastructure as code, but what "code" typically defines infrastructure? In the scope of model-driven DevOps, the code is composed of the code for the automation tooling (for example, Ansible Playbooks, Jinja templates, Python code) and the textual "code" (for example, YAML files, JSON files) that contain the data that describes your network. This data is referred to as the source of truth (SoT), as illustrated in Figure 4-1.

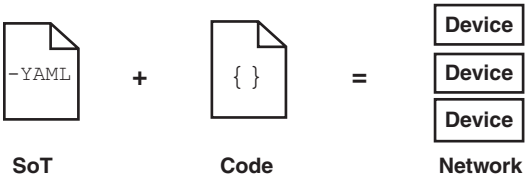

FIGURE 4-1 Infrastructure IaC

Source of Truth

The source of truth of a network is the central repository of all information that is needed to configure the network to a desired state. When constructed properly, your source of truth *is* your network. If your network were to be destroyed, you would be able to reconstitute the entire environment from the source of truth.

Unfortunately, *source of truth* is a misunderstood term in DevOps. Some might argue that the network itself is the source of truth. If that is the case, then it would be hard to do DevOps. If the network is the source of truth, then how do you check that there have been no changes to the source of truth? That would entail verifying that no device was touched or changed by human interaction. Then we need some central repository of state to check the network against, correct? That is our source of truth. Many operators fundamentally know this principle, but as Bob from ACME Corp discovered, it is a hard transition to make.

The opposite of your infrastructure being your source of truth is your infrastructure being immutable. That is, you never change your network; you simply replace it. This approach works well for cloud-native infrastructure such as Kubernetes in which you do a rolling replacement of an existing pod with an updated pod; however, it does not work quite as well for a physical network, except in cases of failure. One reason is that network infrastructure does not always respond well to a change. Sometimes changing an access control list (ACL) or a route causes some traffic perturbation. If the device has already failed or experienced network-affecting problems, however, it is often easier to just replace the device, push down the configuration, and troubleshoot the failed device in the laboratory.

Using a source of truth is also useful for provisioning a new device on the network. Clearly, a new device cannot hold its own "truth" because it is not yet configured. Therefore, the configuration must be created completely from the source of truth. So, both new devices and reconstructed devices due to failure should come from the source of truth. In fact, all operations are just a push of source of truth data into a device in whole or in part. An example of a partial push would be, rather than pushing the whole source of truth to a device, you might just want to update the NTP servers. In this case you would push out only that specific information instead of the entire configuration. In general, however, it is recommended that you push out all of the device's data so that you can completely test and validate all data for each change. With most APIs, the platform or the individual device can figure out what the actual changes are and apply them appropriately. Looking at all automation operations as simply a push of data from your source of truth into your infrastructure can greatly simplify your IaC.

Data Models

A large amount of information is needed to configure infrastructure. The scale and management of source of truth data are often not as important for applications or servers. Why is this? The reason is that building a single system is a well-defined procedure with relatively few permutations or inter-dependencies on other systems. Also, provisioning a system generally consists of configuring values like hostname, IPs, DNS, AAA, and packages. Each is a key/value pair (for example, nameservers = 8.8.8.8, 8.8.4.4) that defines the operation of that system, and there are relatively few of them.

This is not the case for a network element. If we take a standard 48-port top-of-rack (ToR) switch, each port could have a description, a state, a virtual local-area network (VLAN), a maximum transition unit (MTU), and so on. A single ToR could have hundreds of key/value pairs that dictate its operation. Multiply that across hundreds or even thousands of switches, and the number of key/value pairs grows rapidly. Collectively, these key/value pairs make up the source of truth of your network, and there can