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:

- The rise of DevOps
- What is infrastructure as code?
- The benefits of infrastructure as code
- How Terraform works
- How Terraform compares to other infrastructure as code tools

The Rise of 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, and Docker. 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.

The goal is to automate as much of the software delivery process as possible. That means that you manage your infrastructure not by clicking around a web page or manually executing shell commands, but through code. This is a concept that is typically called *infrastructure as code*.

What Is Infrastructure as Code?

The idea behind infrastructure as code (IAC) is that you write and execute code to define, deploy, update, and destroy your infrastructure. This represents an important shift in mindset in which you treat all aspects of operations as software—even those aspects that represent hardware (e.g., setting up physical servers). In fact, a key insight of DevOps is that you can manage almost *everything* in code, including servers, databases, networks, log files, application configuration, documentation, automated tests, deployment processes, and so on.

There are five broad categories of IAC tools:

- Ad hoc scripts
- Configuration management tools
- Server templating tools
- Orchestration tools
- Provisioning tools

Let's look at these one at a time.

Ad Hoc Scripts

The most straightforward approach to automating anything is to write an *ad hoc script*. You take whatever task you were doing manually, break it down into discrete steps, use your favorite scripting language (e.g., Bash, Ruby, Python) to define each of those steps in code, and execute that script on your server, as shown in Figure 1-1.

Figure 1-1. Running an ad hoc script on your server

For example, here is a Bash script called *setup-webserver.sh* that configures a web server by installing dependencies, checking out some code from a Git repo, and firing up an Apache web server:

```
# Update the apt-get cache
sudo apt-get update

# Install PHP and Apache
sudo apt-get install -y php apache2
```

```
# Copy the code from the repository
sudo git clone https://github.com/brikis98/php-app.git
/var/www/html/app

# Start Apache
sudo service apache2 start
```

The great thing about ad hoc scripts is that you can use popular, general-purpose programming languages and you can write the code however you want. The terrible thing about ad hoc scripts is that you can use popular, general-purpose programming languages and you can write the code however you want.

Whereas tools that are purpose-built for IAC provide concise APIs for accomplishing complicated tasks, if you're using a general-purpose programming language, you need to write completely custom code for every task. Moreover, tools designed for IAC usually enforce a particular structure for your code, whereas with a general-purpose programming language, each developer will use their own style and do something different. Neither of these problems is a big deal for an eight-line script that installs Apache, but it gets messy if you try to use ad hoc scripts to manage dozens of servers, databases, load balancers, network configurations, and so on.

If you've ever had to maintain a large repository of Bash scripts, you know that it almost always devolves into a mess of unmaintainable spaghetti code. Ad hoc scripts are great for small, one-off tasks, but if you're going to be managing all of your infrastructure as code, then you should use an IaC tool that is purpose-built for the job.

Configuration Management Tools

Chef, Puppet, Ansible, and SaltStack are all *configuration management tools*, which means that they are designed to install and manage software on existing servers. For example, here is an *Ansible Role* called *web-server.yml* that configures the same Apache web server as the *setup-webserver.sh* script:

```
- name: Update the apt-get cache
  apt:
    update_cache: yes

- name: Install PHP
  apt:
```

```
      name: php

  - name: Install Apache
    apt:
      name: apache2

  - name: Copy the code from the repository
    git: repo=https://github.com/brikis98/php-app.git dest=/var/www/html/app

  - name: Start Apache
    service: name=apache2 state=started enabled=yes
```

The code looks similar to the Bash script, but using a tool like Ansible offers a number of advantages:

Coding conventions

Ansible enforces a consistent, predictable structure, including documentation, file layout, clearly named parameters, secrets management, and so on. While every developer organizes their ad hoc scripts in a different way, most configuration management tools come with a set of conventions that makes it easier to navigate the code.

Idempotence

Writing an ad hoc script that works once isn't too difficult; writing an ad hoc script that works correctly even if you run it over and over again is a lot more difficult. Every time you go to create a folder in your script, you need to remember to check whether that folder already exists; every time you add a line of configuration to a file, you need to check that line doesn't already exist; every time you want to run an app, you need to check that the app isn't already running.

Code that works correctly no matter how many times you run it is called *idempotent code*. To make the Bash script from the previous section idempotent, you'd need to add many lines of code, including lots of if-statements. Most Ansible functions, on the other hand, are idempotent by default. For example, the *web-server.yml* Ansible role will install Apache only if it isn't installed already and will try to start the Apache web server only if it isn't running already.

Distribution

Ad hoc scripts are designed to run on a single, local machine. Ansible and other configuration management tools are

designed specifically for managing large numbers of remote servers, as shown in Figure 1-2.

Figure 1-2. A configuration management tool like Ansible can execute your code across a large number of servers

For example, to apply the *web-server.yml* role to five servers, you first create a file called *hosts* that contains the IP addresses of those servers:

```
[webservers]

11.11.11.11

11.11.11.12

11.11.11.13

11.11.11.14

11.11.11.15
```

Next, you define the following *Ansible playbook*:

```
- hosts: webservers
  roles:
  - webserver
```

Finally, you execute the playbook as follows:

```
ansible-playbook playbook.yml
```

This instructs Ansible to configure all five servers in parallel. Alternatively, by setting a parameter called `serial` in the playbook, you can do a *rolling deployment*, which updates the servers in batches. For example, setting `serial` to `2` directs Ansible to update two of the servers at a time, until all five are done. Duplicating any of this logic in an ad hoc script would take dozens or even hundreds of lines of code.

Server Templating Tools

An alternative to configuration management that has been growing in popularity recently are *server templating tools* such as Docker, Packer,

and Vagrant. Instead of launching a bunch of servers and configuring them by running the same code on each one, the idea behind server templating tools is to create an *image* of a server that captures a fully self-contained "snapshot" of the operating system (OS), the software, the files, and all other relevant details. You can then use some other IaC tool to install that image on all of your servers, as shown in Figure 1-3.

Figure 1-3. You can use a server templating tool like Packer to create a self-contained image of a server. You can then use other tools, such as Ansible, to install that image across all of your servers.

As shown in Figure 1-4, there are two broad categories of tools for working with images:

Virtual machines

A *virtual machine* (VM) emulates an entire computer system, including the hardware. You run a *hypervisor*, such as VMWare, VirtualBox, or Parallels, to virtualize (i.e., simulate) the underlying CPU, memory, hard drive, and networking. The benefit of this is that any *VM image* that you run on top of the hypervisor can see only the virtualized hardware, so it's fully isolated from the host machine and any other VM images, and it will run exactly the same way in all environments (e.g., your computer, a QA server, a production server). The drawback is that virtualizing all this hardware and running a totally separate OS for each VM incurs a lot of overhead in terms of CPU usage, memory usage, and startup time. You can define VM images as code using tools such as Packer and Vagrant.

Containers

A *container* emulates the user space of an OS.[2] You run a *container engine*, such as Docker, CoreOS rkt, or cri-o, to create isolated processes, memory, mount points, and networking. The benefit of this is that any container you run on top of the container engine can see only its own user space, so it's isolated from the host machine and other containers and will run exactly the same way in all environments (your computer, a QA server, a production server, etc.). The drawback is that all of the containers running on a single server share that server's OS kernel and hardware, so it's much more difficult to achieve the level of isolation and security

you get with a VM.[3] However, because the kernel and hardware are shared, your containers can boot up in milliseconds and have virtually no CPU or memory overhead. You can define container images as code using tools such as Docker and CoreOS rkt.

Figure 1-4. The two main types of images: VMs, on the left, and containers, on the right. VMs virtualize the hardware, whereas containers virtualize only user space.

For example, here is a Packer template called *web-server.json* that creates an *Amazon Machine Image* (AMI), which is a VM image that you can run on AWS:

```
{

  "builders": [{
    "ami_name": "packer-example",
    "instance_type": "t2.micro",
    "region": "us-east-2",
    "type": "amazon-ebs",
    "source_ami": "ami-0c55b159cbfafe1f0",
    "ssh_username": "ubuntu"
  }],
  "provisioners": [{
    "type": "shell",
    "inline": [
      "sudo apt-get update",
      "sudo apt-get install -y php apache2",
      "sudo git clone https://github.com/brikis98/php-app.git
/var/www/html/app"
    ],
    "environment_vars": [
      "DEBIAN_FRONTEND=noninteractive"
    ]
  }]
}
```

This Packer template configures the same Apache web server that you saw in *setup-webserver.sh* using the same Bash code.[4] The only difference between the preceding code and previous examples is that this Packer template does not start the Apache web server (e.g., by calling `sudo service apache2 start`). That's because server templates are typically used to install software in images, but it's only when you run the image—for example, by deploying it on a server—that you should actually run that software.

You can build an AMI from this template by running `packer build webserver.json`, and after the build completes, you can install that AMI on all of your AWS servers, configure each server to run Apache when the

server is booting (you'll see an example of this in the next section), and they will all run exactly the same way.

Note that the different server templating tools have slightly different purposes. Packer is typically used to create images that you run directly on top of production servers, such as an AMI that you run in your production AWS account. Vagrant is typically used to create images that you run on your development computers, such as a VirtualBox image that you run on your Mac or Windows laptop. Docker is typically used to create images of individual applications. You can run the Docker images on production or development computers, as long as some other tool has configured that computer with the Docker Engine. For example, a common pattern is to use Packer to create an AMI that has the Docker Engine installed, deploy that AMI on a cluster of servers in your AWS account, and then deploy individual Docker containers across that cluster to run your applications.

Server templating is a key component of the shift to *immutable infrastructure*. This idea is inspired by functional programming and entails variables that are immutable, so after you've set a variable to a value, you can never change that variable again. If you need to update something, you create a new variable. Because variables never change, it's a lot easier to reason about your code.

The idea behind immutable infrastructure is similar: once you've deployed a server, you never make changes to it again. If you need to update something, such as deploy a new version of your code, you create a new image from your server template and you deploy it on a new server. Because servers never change, it's a lot easier to reason about what's deployed.

Orchestration Tools

Server templating tools are great for creating VMs and containers, but how do you actually manage them? For most real-world use cases, you'll need a way to do the following:

- Deploy VMs and containers, making efficient use of your hardware.

- Roll out updates to an existing fleet of VMs and containers using strategies such as rolling deployment, blue-green deployment, and canary deployment.
- Monitor the health of your VMs and containers and automatically replace unhealthy ones (*auto healing*).
- Scale the number of VMs and containers up or down in response to load (*auto scaling*).
- Distribute traffic across your VMs and containers (*load balancing*).
- Allow your VMs and containers to find and talk to one another over the network (*service discovery*).

Handling these tasks is the realm of *orchestration tools* such as Kubernetes, Marathon/Mesos, Amazon Elastic Container Service (Amazon ECS), Docker Swarm, and Nomad. For example, Kubernetes allows you to define how to manage your Docker containers as code. You first deploy a *Kubernetes cluster*, which is a group of servers that Kubernetes will manage and use to run your Docker containers. Most major cloud providers have native support for deploying managed Kubernetes clusters, such as Amazon Elastic Container Service for Kubernetes (Amazon EKS), Google Kubernetes Engine (GKE), and Azure Kubernetes Service (AKS).

Once you have a working cluster, you can define how to run your Docker container as code in a YAML file:

```
apiVersion: apps/v1

# Use a Deployment to deploy multiple replicas of your Docker
# container(s) and to declaratively roll out updates to them
kind: Deployment

# Metadata about this Deployment, including its name
metadata:
  name: example-app

# The specification that configures this Deployment
spec:
  # This tells the Deployment how to find your container(s)
  selector:
    matchLabels:
      app: example-app

  # This tells the Deployment to run three replicas of your
  # Docker container(s)
  replicas: 3
```

```
    # Specifies how to update the Deployment. Here, we
    # configure a rolling update.
    strategy:
      rollingUpdate:
        maxSurge: 3
        maxUnavailable: 0
      type: RollingUpdate

    # This is the template for what container(s) to deploy
    template:

      # The metadata for these container(s), including labels
      metadata:
        labels:
          app: example-app

      # The specification for your container(s)
      spec:
        containers:

          # Run Apache listening on port 80
          - name: example-app
            image: httpd:2.4.39
            ports:
              - containerPort: 80
```

This file instructs Kubernetes to create a *Deployment*, which is a declarative way to define:

- One or more Docker containers to run together. This group of containers is called a *Pod*. The Pod defined in the preceding code contains a single Docker container that runs Apache.
- The settings for each Docker container in the Pod. The Pod in the preceding code configures Apache to listen on port 80.
- How many copies (aka *replicas*) of the Pod to run in your cluster. The preceding code configures three replicas. Kubernetes automatically figures out where in your cluster to deploy each Pod, using a scheduling algorithm to pick the optimal servers in terms of high availability (e.g., try to run each Pod on a separate server so a single server crash doesn't take down your app), resources (e.g., pick servers that have available the ports, CPU, memory, and other resources required by your containers), performance (e.g., try to pick servers with the least load and fewest containers on them), and so on. Kubernetes also constantly monitors the cluster to ensure that there are always three replicas running, automatically replacing any Pods that crash or stop responding.

- How to deploy updates. When deploying a new version of the Docker container, the preceding code rolls out three new replicas, waits for them to be healthy, and then undeploys the three old replicas.

That's a lot of power in just a few lines of YAML! You run `kubectl apply -f example-app.yml` to instruct Kubernetes to deploy your app. You can then make changes to the YAML file and run `kubectl apply` again to roll out the updates.

Provisioning Tools

Whereas configuration management, server templating, and orchestration tools define the code that runs on each server, *provisioning tools* such as Terraform, CloudFormation, and OpenStack Heat are responsible for creating the servers themselves. In fact, you can use provisioning tools to not only create servers, but also databases, caches, load balancers, queues, monitoring, subnet configurations, firewall settings, routing rules, Secure Sockets Layer (SSL) certificates, and almost every other aspect of your infrastructure, as shown in Figure 1-5.

For example, the following code deploys a web server using Terraform:

```
resource "aws_instance" "app" {
  instance_type      = "t2.micro"
  availability_zone = "us-east-2a"
  ami                = "ami-0c55b159cbfafe1f0"

  user_data = <<-EOF
              #!/bin/bash
                sudo service apache2 start
                EOF
}
```

Don't worry if you're not yet familiar with some of the syntax. For now, just focus on two parameters:

ami
: This parameter specifies the ID of an AMI to deploy on the server. You could set this parameter to the ID of an AMI built from the *web-server.json* Packer template in the previous section, which has PHP, Apache, and the application source code.

user_data
: This is a Bash script that executes when the web server is booting. The preceding code uses this script to boot up Apache.

In other words, this code shows you provisioning and server templating working together, which is a common pattern in immutable infrastructure.

Figure 1-5. You can use provisioning tools with your cloud provider to create servers, databases, load balancers, and all other parts of your infrastructure

The Benefits of Infrastructure as Code

Now that you've seen all the different flavors of IaC, a good question to ask is, why bother? Why learn a bunch of new languages and tools and encumber yourself with yet more code to manage?

The answer is that code is powerful. In exchange for the upfront investment of converting your manual practices to code, you get dramatic improvements in your ability to deliver software. According to the 2016 State of DevOps Report, organizations that use DevOps practices, such as IaC, deploy 200 times more frequently, recover from failures 24 times faster, and have lead times that are *2,555* times lower.

When your infrastructure is defined as code, you are able to use a wide variety of software engineering practices to dramatically improve your software delivery process, including the following:

Self-service

> Most teams that deploy code manually have a small number of sysadmins (often, just one) who are the only ones who know all the magic incantations to make the deployment work and are the only ones with access to production. This becomes a major bottleneck as the company grows. If your infrastructure is defined in code, the entire deployment process can be automated, and developers can kick off their own deployments whenever necessary.

Speed and safety

> If the deployment process is automated, it will be significantly faster, since a computer can carry out the deployment steps far faster than a person; and safer, given that an automated process will be more consistent, more repeatable, and not prone to manual error.

Documentation

Instead of the state of your infrastructure being locked away in a single sysadmin's head, you can represent the state of your infrastructure in source files that anyone can read. In other words, IaC acts as documentation, allowing everyone in the organization to understand how things work, even if the sysadmin goes on vacation.

Version control

You can store your IaC source files in version control, which means that the entire history of your infrastructure is now captured in the commit log. This becomes a powerful tool for debugging issues, because any time a problem pops up, your first step will be to check the commit log and find out what changed in your infrastructure, and your second step might be to resolve the problem by simply reverting back to a previous, known-good version of your IaC code.

Validation

If the state of your infrastructure is defined in code, for every single change, you can perform a code review, run a suite of automated tests, and pass the code through static analysis tools—all practices that are known to significantly reduce the chance of defects.

Reuse

You can package your infrastructure into reusable modules, so that instead of doing every deployment for every product in every environment from scratch, you can build on top of known, documented, battle-tested pieces.[5]

Happiness

There is one other very important, and often overlooked, reason for why you should use IaC: happiness. Deploying code and managing infrastructure manually is repetitive and tedious. Developers and sysadmins resent this type of work, since it involves no creativity, no challenge, and no recognition. You could deploy code perfectly for months, and no one will take notice—until that one day when you mess it up. That creates a stressful and unpleasant environment. IaC offers a better alternative that allows computers

to do what they do best (automation) and developers to do what they do best (coding).

Now that you have a sense of why IaC is important, the next question is whether Terraform is the best IaC tool for you. To answer that, I'm first going to do a very quick primer on how Terraform works, and then I'll compare it to the other popular IaC options out there, such as Chef, Puppet, and Ansible.

How Terraform Works

Here is a high-level and somewhat simplified view of how Terraform works. Terraform is an open source tool created by HashiCorp and written in the Go programming language. The Go code compiles down into a single binary (or rather, one binary for each of the supported operating systems) called, not surprisingly, `terraform`.

You can use this binary to deploy infrastructure from your laptop or a build server or just about any other computer, and you don't need to run any extra infrastructure to make that happen. That's because under the hood, the `terraform` binary makes API calls on your behalf to one or more *providers*, such as AWS, Azure, Google Cloud, DigitalOcean, OpenStack, and more. This means that Terraform gets to leverage the infrastructure those providers are already running for their API servers, as well as the authentication mechanisms you're already using with those providers (e.g., the API keys you already have for AWS).

How does Terraform know what API calls to make? The answer is that you create *Terraform configurations*, which are text files that specify what infrastructure you want to create. These configurations are the "code" in "infrastructure as code." Here's an example Terraform configuration:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}

resource "google_dns_record_set" "a" {
  name         = "demo.google-example.com"
  managed_zone = "example-zone"
  type         = "A"
  ttl          = 300
```

```
    rrdatas        = [aws_instance.example.public_ip]
  }
```

Even if you've never seen Terraform code before, you shouldn't have too much trouble reading it. This snippet instructs Terraform to make API calls to AWS to deploy a server and then make API calls to Google Cloud to create a DNS entry pointing to the AWS server's IP address. In just a single, simple syntax (which you'll learn in Chapter 2), Terraform allows you to deploy interconnected resources across multiple cloud providers.

You can define your entire infrastructure—servers, databases, load balancers, network topology, and so on—in Terraform configuration files and commit those files to version control. You then run certain Terraform commands, such as `terraform apply`, to deploy that infrastructure. The `terraform` binary parses your code, translates it into a series of API calls to the cloud providers specified in the code, and makes those API calls as efficiently as possible on your behalf, as shown in Figure 1-6.

Figure 1-6. Terraform is a binary that translates the contents of your configurations into API calls to cloud providers

When someone on your team needs to make changes to the infrastructure, instead of updating the infrastructure manually and directly on the servers, they make their changes in the Terraform configuration files, validate those changes through automated tests and code reviews, commit the updated code to version control, and then run the `terraform apply` command to have Terraform make the necessary API calls to deploy the changes.

TRANSPARENT PORTABILITY BETWEEN CLOUD PROVIDERS

Because Terraform supports many different cloud providers, a common question that arises is whether it supports *transparent portability* between them. For example, if you used Terraform to define a bunch of servers, databases, load balancers, and other infrastructure in AWS, could you instruct Terraform to deploy exactly the same infrastructure in another cloud provider, such as Azure or Google Cloud, in just a few clicks?

This question turns out to be a bit of a red herring. The reality is that you can't deploy "exactly the same infrastructure" in a different cloud provider because the cloud providers don't offer the same types of infrastructure! The servers, load balancers, and databases offered by AWS are very different from those in Azure and Google Cloud in terms of features, configuration, management, security, scalability, availability, observability, and so on. There is no easy way to "transparently" paper over these differences, especially as functionality in one cloud provider often doesn't exist at all in the others.

Terraform's approach is to allow you to write code that is specific to each provider, taking advantage of that provider's unique functionality, but to use the same language, toolset, and IaC practices under the hood for all providers.

How Terraform Compares to Other IaC Tools

Infrastructure as code is wonderful, but the process of picking an IaC tool is not. Many of the IaC tools overlap in what they do. Many of them are open source. Many of them offer commercial support. Unless you've used each one yourself, it's not clear what criteria you should use to pick one or the other.

What makes this even more difficult is that most of the comparisons you find between these tools do little more than list the general properties of each one and make it sound as if you could be equally successful with any of them. And although that's technically true, it's not helpful. It's a bit like telling a programming newbie that you could be equally successful building a website with PHP, C, or assembly—a statement that's technically true, but one that omits a huge amount of information that is essential for making a good decision.

In the following sections, I'm going to do a detailed comparison between the most popular configuration management and provisioning tools: Terraform, Chef, Puppet, Ansible, SaltStack, CloudFormation, and OpenStack Heat. My goal is to help you decide whether Terraform is a good choice by explaining why my company, Gruntwork, picked Terraform as our IaC tool of choice and, in some sense, why I wrote this book.[6] As with all technology decisions, it's a question of trade-offs and

priorities, and even though your particular priorities might be different than mine, my hope is that sharing this thought process will help you to make your own decision.

Here are the main trade-offs to consider:

- Configuration management versus provisioning
- Mutable infrastructure versus immutable infrastructure
- Procedural language versus declarative language
- Master versus masterless
- Agent versus agentless
- Large community versus small community
- Mature versus cutting-edge
- Using multiple tools together

Configuration Management Versus Provisioning

As you saw earlier, Chef, Puppet, Ansible, and SaltStack are all configuration management tools, whereas CloudFormation, Terraform, and OpenStack Heat are all provisioning tools. Although the distinction is not entirely clear cut, given that configuration management tools can typically do some degree of provisioning (e.g., you can deploy a server with Ansible) and provisioning tools can typically do some degree of configuration (e.g., you can run configuration scripts on each server you provision with Terraform), you typically want to pick the tool that's the best fit for your use case.[7]

In particular, if you use server templating tools such as Docker or Packer, the vast majority of your configuration management needs are already taken care of. Once you have an image created from a Dockerfile or Packer template, all that's left to do is provision the infrastructure for running those images. And when it comes to provisioning, a provisioning tool is going to be your best choice.

That said, if you're not using server templating tools, a good alternative is to use a configuration management and provisioning tool together. For

example, you might use Terraform to provision your servers and run Chef to configure each one.

Mutable Infrastructure Versus Immutable Infrastructure

Configuration management tools such as Chef, Puppet, Ansible, and SaltStack typically default to a mutable infrastructure paradigm. For example, if you instruct Chef to install a new version of OpenSSL, it will run the software update on your existing servers and the changes will happen in place. Over time, as you apply more and more updates, each server builds up a unique history of changes. As a result, each server becomes slightly different than all the others, leading to subtle configuration bugs that are difficult to diagnose and reproduce (this is the same configuration drift problem that happens when you manage servers manually, although it's much less problematic when using a configuration management tool). Even with automated tests these bugs are difficult to catch; a configuration management change might work just fine on a test server, but that same change might behave differently on a production server because the production server has accumulated months of changes that aren't reflected in the test environment.

If you're using a provisioning tool such as Terraform to deploy machine images created by Docker or Packer, most "changes" are actually deployments of a completely new server. For example, to deploy a new version of OpenSSL, you would use Packer to create a new image with the new version of OpenSSL, deploy that image across a set of new servers, and then terminate the old servers. Because every deployment uses immutable images on fresh servers, this approach reduces the likelihood of configuration drift bugs, makes it easier to know exactly what software is running on each server, and allows you to easily deploy any previous version of the software (any previous image) at any time. It also makes your automated testing more effective, because an immutable image that passes your tests in the test environment is likely to behave exactly the same way in the production environment.

Of course, it's possible to force configuration management tools to do immutable deployments, too, but it's not the idiomatic approach for those tools, whereas it's a natural way to use provisioning tools. It's also worth mentioning that the immutable approach has downsides of its own. For example, rebuilding an image from a server template and

redeploying all your servers for a trivial change can take a long time. Moreover, immutability lasts only until you actually run the image. After a server is up and running, it will begin making changes on the hard drive and experiencing some degree of configuration drift (although this is mitigated if you deploy frequently).

Procedural Language Versus Declarative Language

Chef and Ansible encourage a *procedural* style in which you write code that specifies, step by step, how to achieve some desired end state. Terraform, CloudFormation, SaltStack, Puppet, and Open Stack Heat all encourage a more *declarative* style in which you write code that specifies your desired end state, and the IaC tool itself is responsible for figuring out how to achieve that state.

To demonstrate the difference, let's go through an example. Imagine that you want to deploy 10 servers (*EC2 Instances* in AWS lingo) to run an AMI with ID `ami-0c55b159cbfafe1f0` (Ubuntu 18.04). Here is a simplified example of an Ansible template that does this using a procedural approach:

```
- ec2:
    count: 10
    image: ami-0c55b159cbfafe1f0
    instance_type: t2.micro
```

And here is a simplified example of a Terraform configuration that does the same thing using a declarative approach:

```
resource "aws_instance" "example" {
  count           = 10
  ami             = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

On the surface, these two approaches might look similar, and when you initially execute them with Ansible or Terraform, they will produce similar results. The interesting thing is what happens when you want to make a change.

For example, imagine traffic has gone up and you want to increase the number of servers to 15. With Ansible, the procedural code you wrote earlier is no longer useful; if you just updated the number of servers to 15 and reran that code, it would deploy 15 new servers, giving you 25

total! So instead, you need to be aware of what is already deployed and write a totally new procedural script to add the five new servers:

```
- ec2:
    count: 5
    image: ami-0c55b159cbfafe1f0
    instance_type: t2.micro
```

With declarative code, because all you do is declare the end state that you want, and Terraform figures out how to get to that end state, Terraform will also be aware of any state it created in the past. Therefore, to deploy five more servers, all you need to do is go back to the same Terraform configuration and update the count from 10 to 15:

```
resource "aws_instance" "example" {
  count           = 15
  ami             = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

If you applied this configuration, Terraform would realize it had already created 10 servers and therefore all it needs to do is create five new servers. In fact, before applying this configuration, you can use Terraform's `plan` command to preview what changes it would make:

```
$ terraform plan



# aws_instance.example[11] will be created

+ resource "aws_instance" "example" {

    + ami             = "ami-0c55b159cbfafe1f0"

    + instance_type   = "t2.micro"

    + (...)

  }



# aws_instance.example[12] will be created
```

```
+ resource "aws_instance" "example" {
    + ami           = "ami-0c55b159cbfafe1f0"
    + instance_type = "t2.micro"
    + (...)
  }

# aws_instance.example[13] will be created
+ resource "aws_instance" "example" {
    + ami           = "ami-0c55b159cbfafe1f0"
    + instance_type = "t2.micro"
    + (...)
  }

# aws_instance.example[14] will be created
+ resource "aws_instance" "example" {
    + ami           = "ami-0c55b159cbfafe1f0"
    + instance_type = "t2.micro"
    + (...)
  }

Plan: 5 to add, 0 to change, 0 to destroy.
```

Now what happens when you want to deploy a different version of the app, such as AMI ID `ami-02bcbb802e03574ba`? With the procedural approach, both of your previous Ansible templates are again not useful, so you need to write yet another template to track down the 10 servers you deployed previously (or was it 15 now?) and carefully update each one to the new version. With the declarative approach of Terraform, you go back to the exact same configuration file again and simply change the `ami` parameter to `ami-02bcbb802e03574ba`:

```
resource "aws_instance" "example" {
  count           = 15
  ami              = "ami-02bcbb802e03574ba"
  instance_type = "t2.micro"
}
```

Obviously, these examples are simplified. Ansible does allow you to use tags to search for existing EC2 Instances before deploying new ones (e.g., using the `instance_tags` and `count_tag` parameters), but having to manually figure out this sort of logic for every single resource you manage with Ansible, based on each resource's past history, can be surprisingly complicated—finding existing Instances not only by tag, but also image version, Availability Zone. This highlights two major problems with procedural IAC tools:

Procedural code does not fully capture the state of the infrastructure

Reading through the three preceding Ansible templates is not enough to know what's deployed. You'd also need to know the *order* in which those templates were applied. Had you applied them in a different order, you might have ended up with different infrastructure, and that's not something you can see in the codebase itself. In other words, to reason about an Ansible or Chef codebase, you need to know the full history of every change that has ever happened.

Procedural code limits reusability

The reusability of procedural code is inherently limited because you must manually take into account the current state of the infrastructure. Because that state is constantly changing, code you used a week ago might no longer be usable because it was designed to modify a state of your infrastructure that no longer exists. As a result, procedural codebases tend to grow large and complicated over time.

With Terraform's declarative approach, the code always represents the latest state of your infrastructure. At a glance, you can determine what's currently deployed and how it's configured, without having to worry about history or timing. This also makes it easy to create reusable code, since you don't need to manually account for the current state of the world. Instead, you just focus on describing your desired state, and Terraform figures out how to get from one state to the other automatically. As a result, Terraform codebases tend to stay small and easy to understand.

Of course, there are downsides to declarative languages, too. Without access to a full programming language, your expressive power is limited. For example, some types of infrastructure changes, such as a zero-downtime deployment, are difficult to express in purely declarative terms (but not impossible, as you'll see in Chapter 5). Similarly, with limited ability to do "logic" (e.g., `if`-statements, loops), creating generic, reusable code can be tricky. Fortunately, Terraform provides a number of powerful primitives—such as input variables, output variables, modules, `create_before_destroy`, `count`, ternary syntax, and built-in functions—that make it possible to create clean, configurable, modular code even in a declarative language. I'll revisit these topics in Chapters 4 and 5.

Master Versus Masterless

By default, Chef, Puppet, and SaltStack all require that you run a *master server* for storing the state of your infrastructure and distributing updates. Every time you want to update something in your infrastructure, you use a client (e.g., a command-line tool) to issue new commands to the master server, and the master server either pushes the updates out to all of the other servers, or those servers pull the latest updates down from the master server on a regular basis.

A master server offers a few advantages. First, it's a single, central place where you can see and manage the status of your infrastructure. Many configuration management tools even provide a web interface (e.g., the Chef Console, Puppet Enterprise Console) for the master server to make it easier to see what's going on. Second, some master servers can run continuously in the background, and enforce your configuration. That

way, if someone makes a manual change on a server, the master server can revert that change to prevent configuration drift.

However, having to run a master server has some serious drawbacks:

Extra infrastructure

: You need to deploy an extra server, or even a cluster of extra servers (for high availability and scalability), just to run the master.

Maintenance

: You need to maintain, upgrade, back up, monitor, and scale the master server(s).

Security

: You need to provide a way for the client to communicate to the master server(s) and a way for the master server(s) to communicate with all the other servers, which typically means opening extra ports and configuring extra authentication systems, all of which increases your surface area to attackers.

Chef, Puppet, and SaltStack do have varying levels of support for masterless modes where you run just their agent software on each of your servers, typically on a periodic schedule (e.g., a cron job that runs every five minutes), and use that to pull down the latest updates from version control (rather than from a master server). This significantly reduces the number of moving parts, but, as I discuss in the next section, this still leaves a number of unanswered questions, especially about how to provision the servers and install the agent software on them in the first place.

Ansible, CloudFormation, Heat, and Terraform are all masterless by default. Or, to be more accurate, some of them might rely on a master server, but it's already part of the infrastructure you're using and not an extra piece that you need to manage. For example, Terraform communicates with cloud providers using the cloud provider's APIs, so in some sense, the API servers are master servers, except that they don't require any extra infrastructure or any extra authentication mechanisms (i.e., just use your API keys). Ansible works by connecting directly to each server over SSH, so again, you don't need to run any extra infrastructure or manage extra authentication mechanisms (i.e., just use your SSH keys).

Agent Versus Agentless

Chef, Puppet, and SaltStack all require you to install *agent software* (e.g., Chef Client, Puppet Agent, Salt Minion) on each server that you want to configure. The agent typically runs in the background on each server and is responsible for installing the latest configuration management updates.

This has a few drawbacks:

Bootstrapping

> How do you provision your servers and install the agent software on them in the first place? Some configuration management tools kick the can down the road, assuming that some external process will take care of this for them (e.g., you first use Terraform to deploy a bunch of servers with an AMI that has the agent already installed); other configuration management tools have a special bootstrapping process in which you run one-off commands to provision the servers using the cloud provider APIs and install the agent software on those servers over SSH.

Maintenance

> You need to carefully update the agent software on a periodic basis, being careful to keep it synchronized with the master server if there is one. You also need to monitor the agent software and restart it if it crashes.

Security

> If the agent software pulls down configuration from a master server (or some other server if you're not using a master), you need to open outbound ports on every server. If the master server pushes configuration to the agent, you need to open inbound ports on every server. In either case, you must figure out how to authenticate the agent to the server to which it's communicating. All of this increases your surface area to attackers.

Once again, Chef, Puppet, and SaltStack do have varying levels of support for agentless modes (e.g., salt-ssh), but these feel like they were tacked on as an afterthought and don't support the full feature set of the configuration management tool. That's why in the wild, the default or

idiomatic configuration for Chef, Puppet, and SaltStack almost always includes an agent and usually a master, too, as shown in Figure 1-7.

Figure 1-7. The typical architecture for Chef, Puppet, and SaltStack involves many moving parts. For example, the default setup for Chef is to run the Chef client on your computer, which talks to a Chef master server, which deploys changes by communicates with Chef clients running on all your other servers.

All of these extra moving parts introduce a large number of new failure modes into your infrastructure. Each time you get a bug report at 3 a.m., you'll need to figure out whether it's a bug in your application code, or your IaC code, or the configuration management client, or the master server(s), or the way the client communicates the master server(s), or the way other servers communicates the master server(s), or…

Ansible, CloudFormation, Heat, and Terraform do not require you to install any extra agents. Or, to be more accurate, some of them require agents, but these are typically already installed as part of the infrastructure you're using. For example, AWS, Azure, Google Cloud, and all of the other cloud providers take care of installing, managing, and authenticating agent software on each of their physical servers. As a user of Terraform, you don't need to worry about any of that: you just issue commands and the cloud provider's agents execute them for you on all of your servers, as shown in Figure 1-8. With Ansible, your servers need to run the SSH Daemon, which is common to run on most servers anyway.

Figure 1-8. Terraform uses a masterless, agent-only architecture. All you need to run is the Terraform client and it takes care of the rest by using the APIs of cloud providers, such as AWS.

Large Community Versus Small Community

Whenever you pick a technology, you are also picking a community. In many cases, the ecosystem around the project can have a bigger impact on your experience than the inherent quality of the technology itself. The community determines how many people contribute to the project, how many plug-ins, integrations, and extensions are available, how easy it is to find help online (e.g., blog posts, questions on StackOverflow), and how easy it is to hire someone to help you (e.g., an employee, consultant, or support company).

It's difficult to do an accurate comparison between communities, but you can spot some trends by searching online. Table 1-1 shows a comparison of popular IaC tools, with data I gathered during May 2019, including whether the IaC tool is open source or closed source, what cloud providers it supports, the total number of contributors and stars on GitHub, how many commits and active issues there were over a one-month period from mid-April to mid-May, how many open source libraries are available for the tool, the number of questions listed for that tool on StackOverflow, and the number of jobs that mention the tool on Indeed.com.[8]

	Source	Cloud	Contributors	Stars	Commits (30 days)	Bugs (30 days)
Chef	Open	All	562	5,794	435	86
Puppet	Open	All	515	5,299	94	314[c]
Ansible	Open	All	4,386	37,161	506	523
SaltStack	Open	All	2,237	9,901	608	441
CloudFormation	Closed	AWS	?	?	?	?
Heat	Open	All	361	349	12	600[i]
Terraform	Open	All	1,261	16,837	173	204

[a] This is the number of cookbooks in the Chef Supermarket.
[b] To avoid false positives for the term "chef," I searched for "chef devops."
[c] Based on the Puppet Labs JIRA account.
[d] This is the number of modules in Puppet Forge.
[e] To avoid false positives for the term "puppet," I searched for "puppet devops."
[f] This is the number of reusable roles in Ansible Galaxy.
[g] This is the number of formulas in the Salt Stack Formulas GitHub account.
[h] This is the number of templates in the awslabs GitHub account.
[i] Based on the OpenStack bug tracker.
[j] I could not find any collections of community Heat templates.
[k] To avoid false positives for the term "heat," I searched for "openstack."
[l] This is the number of modules in the Terraform Registry.

Table 1-1. A comparison of IaC communities

Obviously, this is not a perfect apples-to-apples comparison. For example, some of the tools have more than one repository, and some use other methods for bug tracking and questions; searching for jobs with common words like “chef” or “puppet” is tricky; Terraform split the provider code out into separate repos in 2017, so measuring activity on solely the core repo dramatically understates activity (by at least 10 times); and so on.

That said, a few trends are obvious. First, all of the IaC tools in this comparison are open source and work with many cloud providers, except for CloudFormation, which is closed source, and works only with AWS. Second, Ansible leads the pack in terms of popularity, with Salt and Terraform not too far behind.

Another interesting trend to note is how these numbers have changed since the first edition of the book. Table 1-2 shows the percent change in each of the numbers from the values I gathered back in September 2016.

	Source	Cloud	Contributors	Stars	Commits (30 days)	Issues (3 days)
Chef	Open	All	+18%	+31%	+139%	+48%
Puppet	Open	All	+19%	+27%	+19%	+42%
Ansible	Open	All	+195%	+97%	+49%	+66%
SaltStack	Open	All	+40%	+44%	+79%	+27%
CloudFormation	Closed	AWS	?	?	?	?
Heat	Open	All	+28%	+23%	-85%	+1,566
Terraform	Open	All	+93%	+194%	-61%	-58%

Table 1-2. How the IaC communities have changed between Sept

Again, the data here is not perfect, but it’s good enough to spot a clear trend: Terraform and Ansible are experiencing explosive growth. The increase in the number of contributors, stars, open source libraries,

StackOverflow posts, and jobs is through the roof.[9] Both of these tools have large, active communities today, and judging by these trends, it's likely that they will become even larger in the future.

Mature Versus Cutting Edge

Another key factor to consider when picking any technology is maturity.

Table 1-3 shows the initial release dates and current version number (as of May 2019) for each of the IaC tools.

	Initial release	Current version
Puppet	2005	6.0.9
Chef	2009	12.19.31
CloudFormation	2011	???
SaltStack	2011	2019.2.0
Ansible	2012	2.5.5
Heat	2012	12.0.0
Terraform	2014	0.12.0

Table 1-3. A comparison of IaC maturity as of May 2

Again, this is not an apples-to-apples comparison, because different tools have different versioning schemes, but some trends are clear. Terraform is, by far, the youngest IaC tool in this comparison. It's still pre-1.0.0, so there is no guarantee of a stable or backward compatible API, and bugs are relatively common (although most of them are minor). This is Terraform's biggest weakness: although it has become extremely popular in a short time, the price you pay for using this new, cutting-edge tool is that it is not as mature as some of the other IaC options.

Using Multiple Tools Together

Although I've been comparing IaC tools this entire chapter, the reality is that you will likely need to use multiple tools to build your infrastructure. Each of the tools you've seen has strengths and weaknesses, so it's your job to pick the right tool for the right job.

Following are three common combinations I've seen work well at a number of companies.

PROVISIONING PLUS CONFIGURATION MANAGEMENT

Example: Terraform and Ansible. You use Terraform to deploy all the underlying infrastructure, including the network topology (i.e., virtual private clouds [VPCs], subnets, route tables), data stores (e.g., MySQL, Redis), load balancers, and servers. You then use Ansible to deploy your apps on top of those servers, as depicted in Figure 1-9.

Figure 1-9. Using Terraform and Ansible together

This is an easy approach to get started with, because there is no extra infrastructure to run (Terraform and Ansible are both client-only applications) and there are many ways to get Ansible and Terraform to work together (e.g., Terraform adds special tags to your servers and Ansible uses those tags to find the server and configure them). The major downside is that using Ansible typically means that you're writing a lot of procedural code, with mutable servers, so as your codebase, infrastructure, and team grow, maintenance can become more difficult.

PROVISIONING PLUS SERVER TEMPLATING

Example: Terraform and Packer. You use Packer to package your apps as VM images. You then use Terraform to deploy (a) servers with these VM images and (b) the rest of your infrastructure, including the network topology (i.e., VPCs, subnets, route tables), data stores (e.g., MySQL, Redis), and load balancers, as illustrated in Figure 1-10.

Figure 1-10. Using Terraform and Packer together

This is also an easy approach to get started with, because there is no extra infrastructure to run (Terraform and Packer are both client-only

applications), and you'll get plenty of practice deploying VM images using Terraform later in this book. Moreover, this is an immutable infrastructure approach, which will make maintenance easier. However, there are two major drawbacks. First, VMs can take a long time to build and deploy, which will slow down your iteration speed. Second, as you'll see in later chapters, the deployment strategies you can implement with Terraform are limited (e.g., you can't implement blue-green deployment natively in Terraform), so you either end up writing lots of complicated deployment scripts, or you turn to orchestration tools, as described next.

PROVISIONING PLUS SERVER TEMPLATING PLUS ORCHESTRATION

Example: Terraform, Packer, Docker, and Kubernetes. You use Packer to create a VM image that has Docker and Kubernetes installed. You then use Terraform to deploy (a) a cluster of servers, each of which runs this VM image, and (b) the rest of your infrastructure, including the network topology (i.e., VPCs, subnets, route tables), data stores (e.g., MySQL, Redis), and load balancers. Finally, when the cluster of servers boots up, it forms a Kubernetes cluster that you use to run and manage your Dockerized applications, as shown in Figure 1-11.

Figure 1-11. Using Terraform, Packer, Docker, and Kubernetes together

The advantage of this approach is that Docker images build fairly quickly, you can run and test them on your local computer, and you can take advantage of all of the built-in functionality of Kubernetes, including various deployment strategies, auto healing, auto scaling, and so on. The drawback is the added complexity, both in terms of extra infrastructure to run (Kubernetes clusters are difficult and expensive to deploy and operate, though most major cloud providers now provide managed Kubernetes services, which can offload some of this work), and in terms of several extra layers of abstraction (Kubernetes, Docker, Packer) to learn, manage, and debug.

Conclusion

Putting it all together, Table 1-4 shows how the most popular IaC tools stack up. Note that this table shows the *default* or *most common* way the various IaC tools are used, though as discussed earlier in this chapter, these IaC tools are flexible enough to be used in other configurations, too (e.g., you can use Chef without a master, you can use Salt to do immutable infrastructure).

	Source	Cloud	Type	Infrastructure	Language
Chef	Open	All	Config Mgmt	Mutable	Procedural
Puppet	Open	All	Config Mgmt	Mutable	Declarative
Ansible	Open	All	Config Mgmt	Mutable	Procedural
SaltStack	Open	All	Config Mgmt	Mutable	Declarative
CloudFormation	Closed	AWS	Provisioning	Immutable	Declarative
Heat	Open	All	Provisioning	Immutable	Declarative
Terraform	Open	All	Provisioning	Immutable	Declarative

Table 1-4. A comparison of the most common way to use the

At Gruntwork, what we wanted was an open source, cloud-agnostic provisioning tool that supported immutable infrastructure, a declarative language, a masterless and agentless architecture, and had a large community and a mature codebase. Table 1-4 shows that Terraform, although not perfect, comes the closest to meeting all of our criteria.

Does Terraform fit your criteria, too? If so, head over to Chapter 2 to learn how to use it.

1 From *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations* (IT Revolution Press, 2016) by Gene Kim, Jez Humble, Patrick Debois, and John Willis.

2 On most modern operating systems, code runs in one of two "spaces": *kernel space* and *user space*. Code running in kernel space has direct, unrestricted access to all of the hardware. There are no security restrictions (i.e., you can execute any CPU instruction, access any part of the hard drive, write to any address in memory) or safety restrictions (e.g., a crash in kernel space will typically crash the entire computer), so kernel space is generally reserved for the lowest-level, most trusted functions of the OS (typically called the *kernel*). Code running in user space does not have any direct access to the hardware and must use APIs exposed by the OS kernel, instead. These APIs can enforce security restrictions (e.g., user permissions) and safety (e.g., a crash in a user space app typically affects only that app), so just about all application code runs in user space.

3 As a general rule, containers provide isolation that's good enough to run your own code, but if you need to run third-party code (e.g., you're building your own cloud provider) that might actively be performing malicious actions, you'll want the increased isolation guarantees of a VM.

4 As an alternative to Bash, Packer also allows you to configure your images using configuration management tools such as Ansible or Chef.

5 Check out the Gruntwork Infrastructure as Code Library for an example.

6 Docker, Packer, and Kubernetes are not part of the comparison, because they can be used with any of the configuration management or provisioning tools.

7 The distinction between configuration management and provisioning is less clear cut these days, because some of the major configuration management tools have gradually improved their support for provisioning, such as Chef Provisioning and the Puppet AWS Module.

8 Most of this data, including the number of contributors, stars, changes, and issues, comes from the open source repositories and bug trackers (mostly GitHub) for each tool. Because CloudFormation is closed source, some of this information is not available.

9 The decline in Terraform's commits and issues is solely due to the fact that I'm only measuring the core Terraform repo, whereas in 2017, all the provider code was extracted into separate repos, so the vast amount of activity across the more than 100 provider repos is not being counted.