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 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, I'll dive into the following topics:

- The rise of DevOps
- What is infrastructure as code?
- 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 had 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 Operations ("Ops"), dedicated to managing this hardware, and a separate team, typically called Developers ("Devs"), dedicated to writing the software.

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: since 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*, where each one has a subtly different configuration from all the others (a problem known as *configuration drift*). As a result, the

branch. Teams start blaming each other. Silos form. The company grinds to a halt.

Nowadays, a profound shift is taking place. Instead of managing their own data centers, many companies are moving to the cloud, taking advantage of services such as Amazon Web Services, Azure, and Google Cloud. 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 may 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 double the number of features it delivered per month, 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, and update your infrastructure. This represents an important shift in mindset where 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

- Configuration management tools
- Server templating tools
- Server 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.

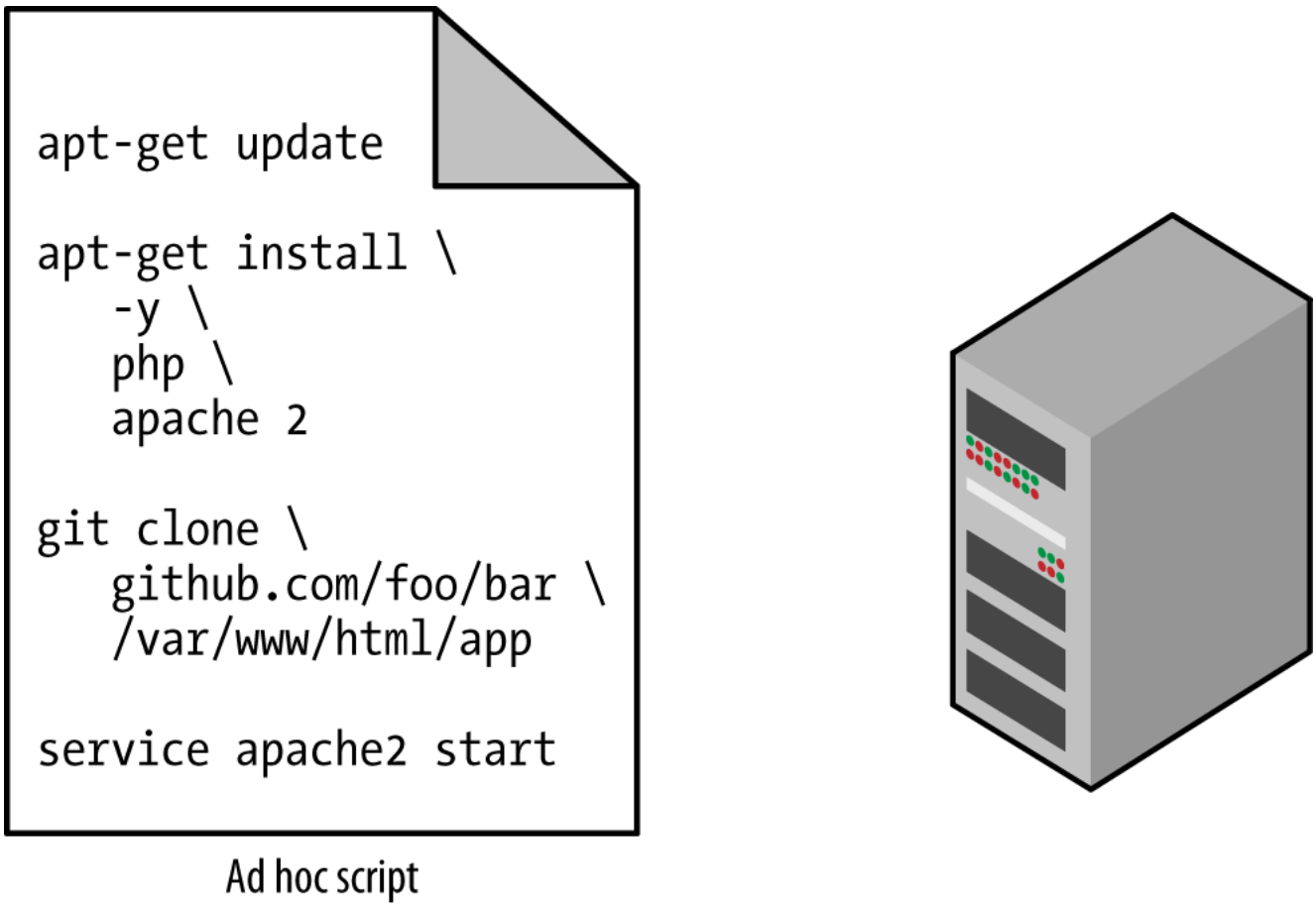


Figure 1-1. Running an ad hoc script on your server

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 the Apache web server:

```
# Install Apache
sudo apt-get install -y 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 have 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 his or her 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 someone else's repository of ad hoc 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 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:

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 harder. Every time you go to create a folder in your script, you need to remember to check if 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 have 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 only install Apache if it isn't installed already and will only try to start the Apache web server 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.

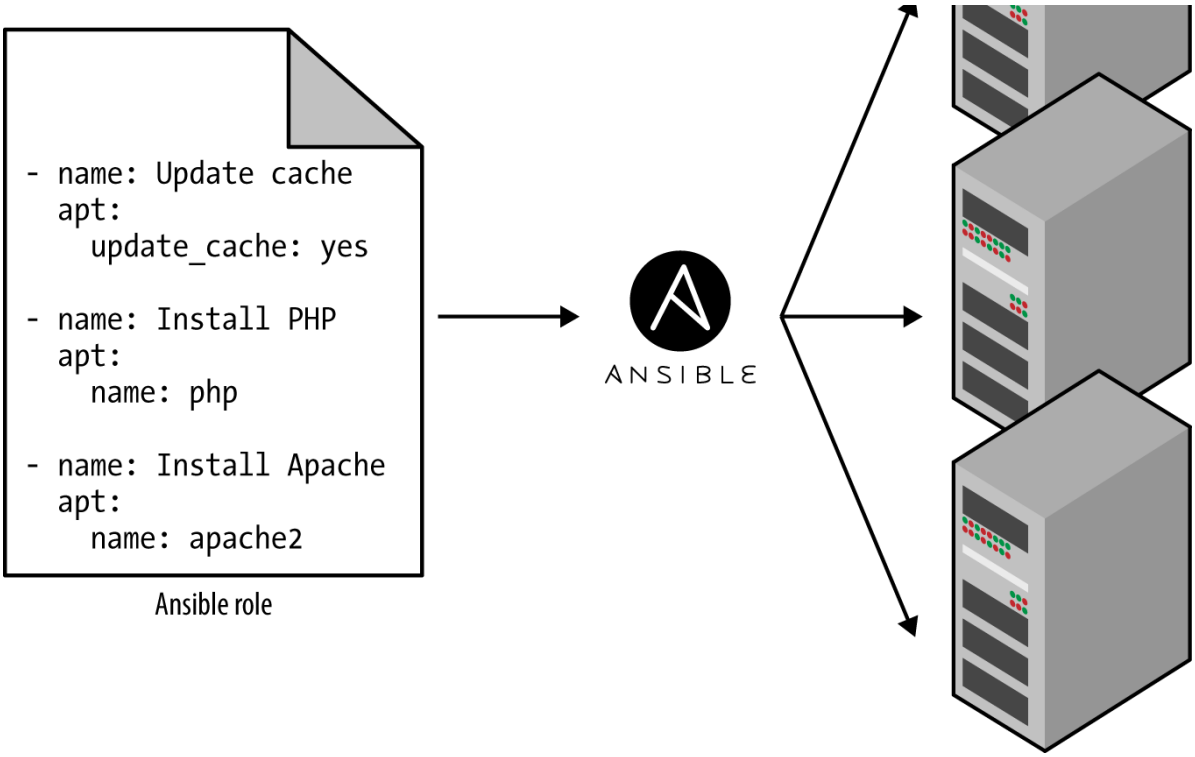


Figure 1-2. A configuration management tool like Ansible can execute your code across a large number of servers

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 will tell Ansible to configure all five servers in parallel. Alternatively, by setting a single parameter called `serial` in the playbook, you can do a rolling deployment, which updates the servers in batches.

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, 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.

```
"provisioners": [{
  "type": "shell",
  "inline": [
     "apt-get update",
     "apt-get install
-y php",
     "apt-get install
-y apache2",
  ]
}]
```

Packer Template

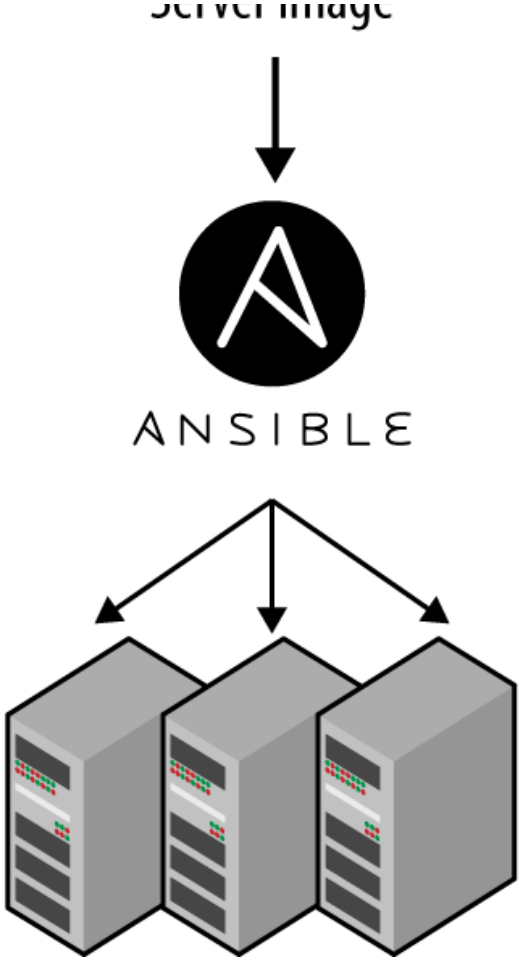


Figure 1-3. A server templating tool like Packer can be used to create a self-contained image of a server. You can then use other tools, such as Ansible, to install that image across all of your servers.

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* you run on top of the hypervisor can only see the virtualized hardware, so it's fully isolated from the host machine and any other VM Images, and will run exactly the same way in all environments (e.g., your computer, a QA server, a production server, etc). The drawback is that virtualizing all this hardware and running a totally separate operating system 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 operating system.[2] You run a *container engine*, such as Docker or CoreOS rkt, to create isolated processes, memory, mount points, and networking. The benefit

Container Images as code using tools such as Docker and CoreOs rkt.

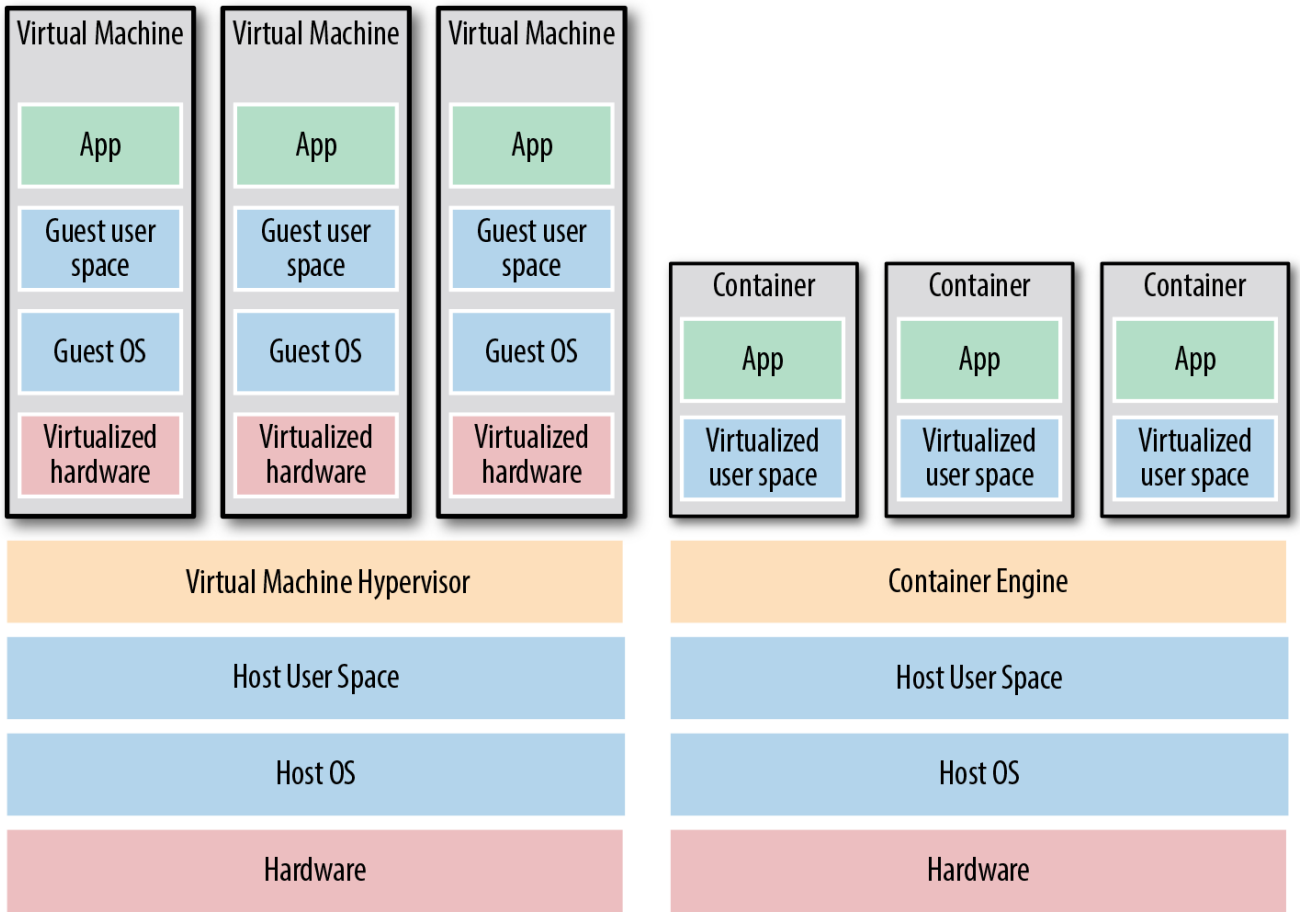


Figure 1-4. The two main types of images: VMs, on the left, and containers, on the right. VMs virtualize the hardware, whereas containers only virtualize user space.

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

```
{
  "builders": [{
    "ami_name": "packer-example",
    "instance_type": "t2.micro",
    "region": "us-east-1",
    "type": "amazon-ebs",
    "source_ami": "ami-40d28157",
    "ssh_username": "ubuntu"
  }],
  "provisioners": [{
    "type": "shell",
    "inline": [
      "sudo apt-get update",
      "sudo apt-get install -y php",
```

This Packer template configures the same Apache web server 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 (e.g., 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 once 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, so 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, where variables are immutable, so once 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. Since 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 (e.g., deploy a new version of your code), you create a new image from your server template and you deploy it on a new server. Since servers never change, it's a lot easier to reason about what's deployed.

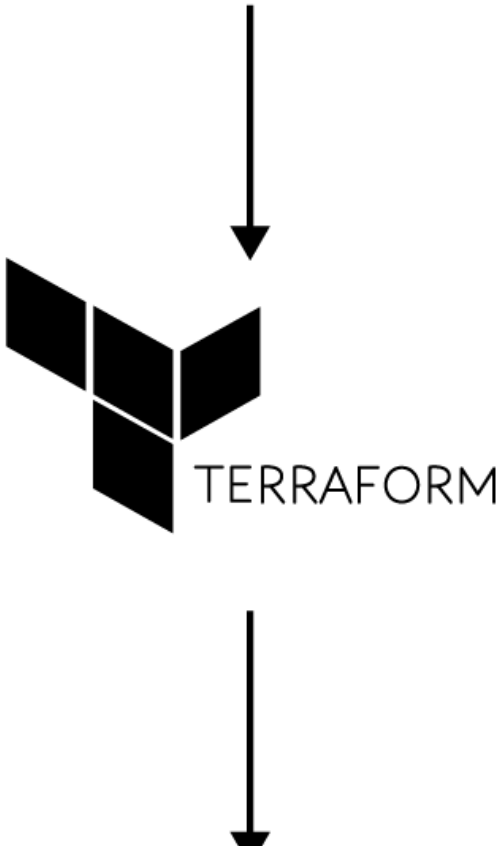
Server Provisioning Tools

Whereas configuration management and server templating tools define the code that runs on each server, *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, SSL certificates, and almost every other aspect of your infrastructure, as shown in Figure 1-5.

```
resource
"aws_instance" "a" {
  ami = "ami-40d28157"
}

resource
"aws_db_instance" "db"
{
  engine = "mysql"
  name = "mydb"
}
```

Terraform configuration



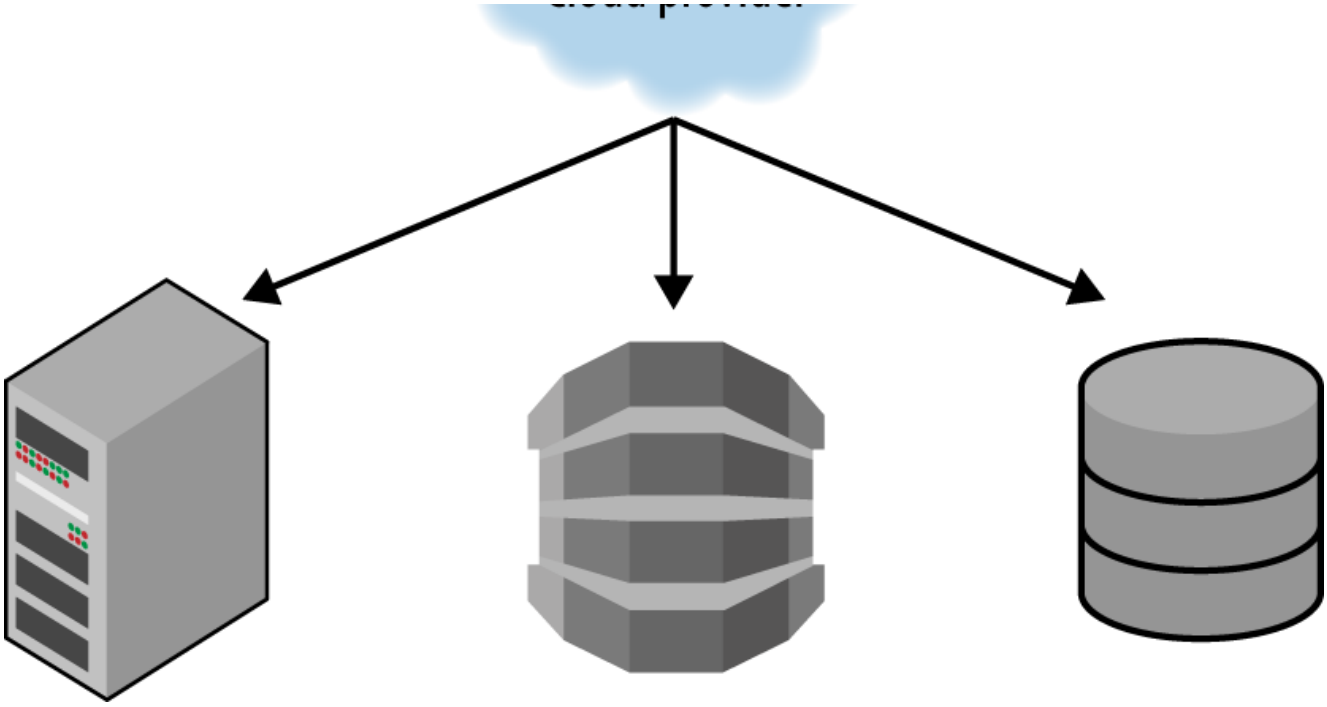


Figure 1-5. Server provisioning tools can be used with your cloud provider to create servers, databases, load balancers, and all other parts of your infrastructure.

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

```
resource "aws_instance" "app" {
  instance_type     = "t2.micro"
  availability_zone = "us-east-1a"
  ami               = "ami-40d28157"

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

Don't worry if some of the syntax isn't familiar to you yet. 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`

Benefits of Infrastructure as Code

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

The answer is that code is powerful. In exchange for the up-front 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:

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, then 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'll be significantly faster, since a computer can carry out the deployment steps far faster than a person; and safer, since 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 the entire history of your infrastructure is now captured in the commit log. This becomes a powerful tool for debugging issues, as 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 may be to resolve the problem by simply reverting back to a previous, known-good version of your IAC code.

Validation

> 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, as 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 right 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 Amazon Web Services (AWS), Azure, Google Cloud, DigitalOcean, OpenStack, etc. That means 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 wish to create. These configurations are the "code" in "infrastructure as code." Here's an example Terraform configuration:

```
resource "aws_instance" "example" {
  ami           = "ami-40d28157"
  instance_type = "t2.micro"
}

resource "dnsimple_record" "example" {
  domain = "example.com"
  name   = "test"
  value  = "${aws_instance.example.public_ip}"
  type   = "A"
}
```

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.

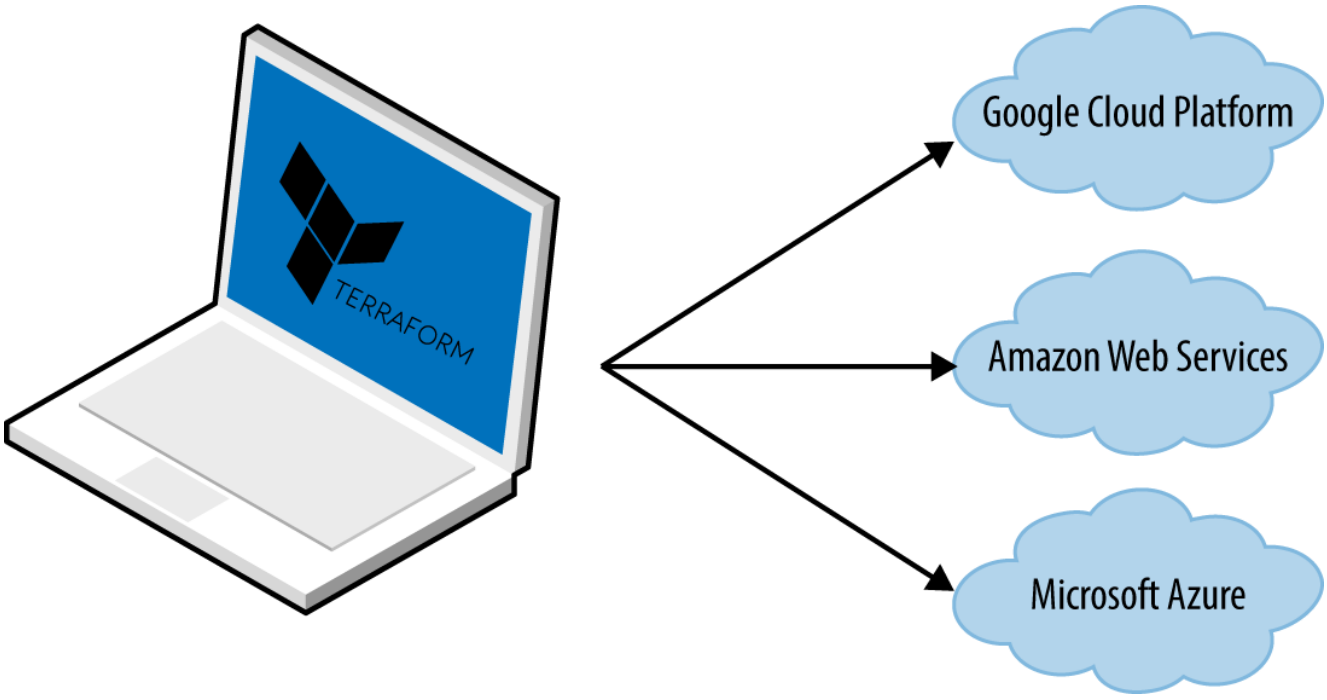


Figure 1-6. Terraform is a binary that translates the contents of your configurations into API calls to cloud providers

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.

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 than those in Azure and Google Cloud in terms of features, configuration, management, security, scalability, availability, observability, and so on. There is no 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 infrastructure as code practices under the hood for all providers.

How Terraform Compares to Other Infrastructure as Code 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 harder 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 like you could be equally successful with any of them. And while 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 if 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 while your particular priorities may be different than mine, I hope that sharing this thought process will help you 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

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, as 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 server 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 tell Chef to install a new version of OpenSSL, it'll 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 hard to catch, as a configuration management change may work just fine on a test server, but that same change may 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, then 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 undeploy the old servers. Since 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, as 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 only lasts until you actually run the image. Once a server is up and running, it'll start 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

Instances" in AWS lingo) to run an AMI with ID `ami-40d28157` (Ubuntu 16.04). Here is a simplified example of an Ansible template that does this using a procedural approach:

```
- ec2:
    count: 10
    image: ami-40d28157
    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-40d28157"
  instance_type = "t2.micro"
}
```

Now at the surface, these two approaches may 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 have to be aware of what is already deployed and write a totally new procedural script to add the 5 new servers:

```
- ec2:
    count: 5
    image: ami-40d28157
    instance_type: t2.micro
```

With declarative code, since all you do is declare the end state 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 5 more servers, all you have 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-40d28157"
  instance_type = "t2.micro"
}
```

If you applied this configuration, Terraform would realize it had already created 10 servers and therefore that all it needed to do was create 5 new servers. In fact, before applying this configuration, you can use Terraform's `plan` command to preview what changes it would make:

```
    ami:                          ami-40d28157
    instance_type:                "t2.micro"
+ aws_instance.example.13
    ami:                          "ami-40d28157"
    instance_type:                "t2.micro"
+ aws_instance.example.14
    ami:                          "ami-40d28157"
    instance_type:                "t2.micro"
+ aws_instance.example.15
    ami:                          "ami-40d28157"
    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-408c7f28`? With the procedural approach, both of your previous Ansible templates are again not useful, so you have 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 once again and simply change the `ami` parameter to `ami-408c7f28`:

```
resource "aws_instance" "example" {
  count         = 15
  ami           = "ami-408c7f28"
  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 (e.g., finding existing instances not only by tag, but also image version, availability zone, etc.). This highlights two major problems with procedural IAC tools:

1. *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 have 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 code base itself. In other words, to reason about an Ansible or Chef codebase, you have to know the full history of every change that has ever happened.

2. *Procedural code limits reusability*. The reusability of procedural code is inherently limited because you have to manually take into account the current state of the infrastructure. Since that state is constantly changing, code you used a week ago may 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 tell 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, as you don't have to manually account for the current state

Similarly, without the 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 interpolation functions—that make it possible to create clean, configurable, modular code even in a declarative language. I'll come back to these topics in Chapter 4 and Chapter 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 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 have to deploy an extra server, or even a cluster of extra servers (for high availability and scalability), just to run the master.

Maintenance

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

Security

: You have 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 just run their agent software on each of your servers, typically on a periodic schedule (e.g., a cron job that runs every 5 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 discussed 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 may rely on a master server, but it's already part of the infrastructure you're using and not an extra piece you have to manage. For example, Terraform communicates with cloud providers using the cloud provider's APIs, so in some

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 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 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 where 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 have to carefully update the agent software on a periodic basis, being careful to keep it in sync with the master server if there is one. You also have 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), then you have to open outbound ports on every server. If the master server pushes configuration to the agent, then you have to open inbound ports on every server. In either case, you have to figure out how to authenticate the agent to the server it's talking to. 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 always 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.

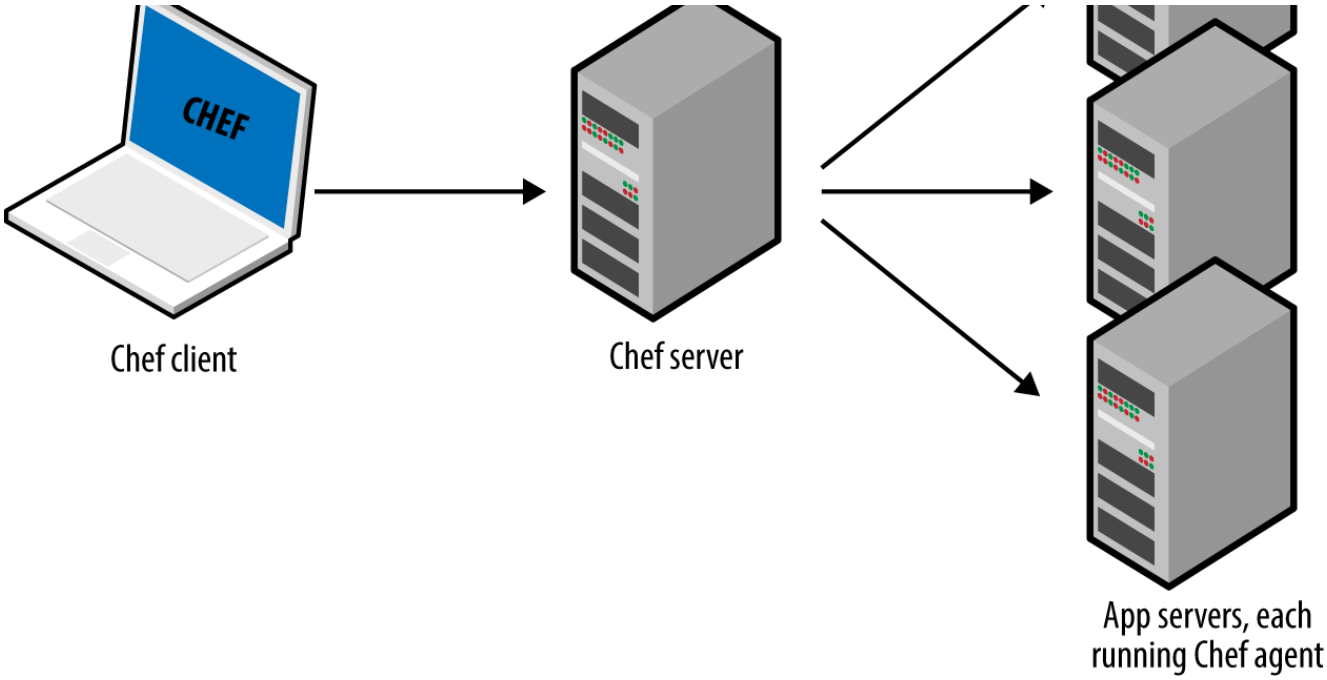


Figure 1-7. The typical architecture for Chef, Puppet, and SaltStack involves many moving parts. For example, the default setup for Chef is to run the Chef client on your computer, which talks to a Chef master server, which deploys changes by talking to Chef agents running on all your other servers.

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 have to figure out if 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 talks to the master server(s), or the way other servers talk to 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 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 have 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.

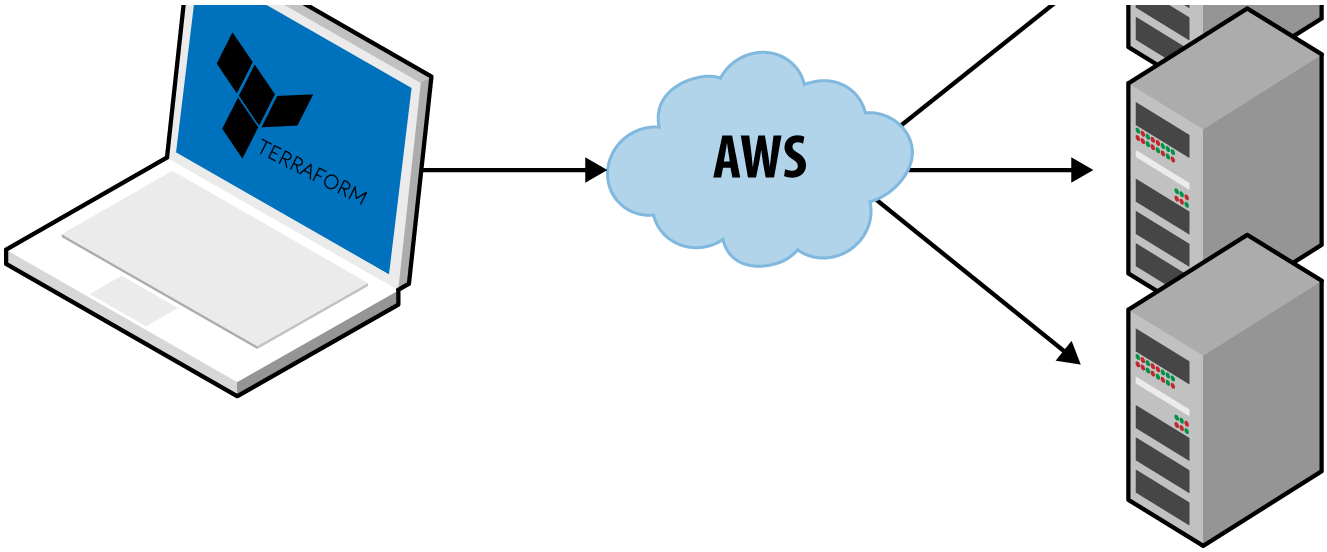


Figure 1-8. Terraform uses a masterless, agent-only architecture. All you need to run is the Terraform client and it takes care of the rest by using the APIs of cloud providers, such as AWS.

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 hard 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 from September 2016, including whether they are open source or closed source, what cloud providers they support, the total number of contributors and stars on GitHub, how many active changes and issues there were in the month of September, 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]

Chef	Open	All	477	4,439	182	58	3,052[a]	4,187	5,631[b]
Puppet	Open	All	432	4,158	79	130[c]	4,435[d]	2,639	5,213[e]
Ansible	Open	All	1,488	18,895	340	315	8,044[f]	3,633	3,901
SaltStack	Open	All	1,596	6,897	689	347	240[g]	614	454
CloudFormation	Closed	AWS	?	?	?	?	240[h]	613	665
Heat	Open	All	283	283	83	36[i]	0[j]	52	72[k]
Terraform	Open	All	653	5,732	440	480	40[l]	131	392

[a] This is the number of cookbooks in the Chef Supermarket.

[b] To avoid false positives for the term “chef”, I searched for “chef engineer”.

[c] Based on the Puppet Labs JIRA account.

[d] This is the number of modules in the Puppet Forge.

[e] To avoid false positives for the term “puppet”, I searched for “puppet engineer”.

[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-community-modules repo.

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; and so on.

Another key factor to consider when picking any technology is maturity. Table 1-2 shows the initial release dates and current version number (as of January 2017) for of each of the IAC tools.

Table 1-2. A comparison of IAC maturity

	Initial release	Current version
Puppet	2005	4.8.1
Chef	2009	12.17.44
CloudFormation	2011	?
SaltStack	2011	2016.11.1
Ansible	2012	v2.1.3.0-1
Heat	2012	7.0.1
Terraform	2014	0.8.2

Again, this is not an apples-to-apples comparison, since 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 gotten 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.

Conclusion

Putting it all together, Table 1-3 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., Chef can be used without a master, Salt can be used to do immutable infrastructure).

Chef	Open	All	Config Mgmt	Mutable	Procedural	Yes	Yes	Large	High
Puppet	Open	All	Config Mgmt	Mutable	Declarative	Yes	Yes	Large	High
Ansible	Open	All	Config Mgmt	Mutable	Procedural	No	No	Large	Medium
SaltStack	Open	All	Config Mgmt	Mutable	Declarative	Yes	Yes	Medium	Medium
CloudFormation	Closed	AWS	Provisioning	Immutable	Declarative	No	No	Small	Medium
Heat	Open	All	Provisioning	Immutable	Declarative	No	No	Small	Low
Terraform	Open	All	Provisioning	Immutable	Declarative	No	No	Medium	Low

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 1-3 shows that Terraform, while not perfect, comes the closest to meeting all of our criteria.

Does Terraform fit your criteria, too? If so, then 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) 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 operating system (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 operating system kernel instead. These APIs can enforce security restrictions (e.g., user permissions) and safety (e.g., a crash in a user space app typically only affects that app), so just about all application code runs in user space.

[5] Check out the Gruntwork Infrastructure Packages for an example.

[6] Docker and Packer 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 has become even less clear cut in recent months, as some of the major configuration management tools have started to add better 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 for each tool. Since CloudFormation is closed source, some of this information is not available.