Chapter 10. How to Use Terraform as a Team

As you've been reading this book and working through the code samples, you've most likely been working by yourself. In the real world, you'll most likely be working as part of a team, which introduces a number of new challenges. You may need to find a way to convince your team to use Terraform and other infrastructure-as-code (IaC) tools. You may need to deal with multiple people concurrently trying to understand, use, and modify the Terraform code you write. And you may need to figure out how to fit Terraform into the rest of your tech stack and make it a part of your company's workflow.

In this chapter, I'll dive into the key processes you need to put in place to make Terraform and IaC work for your team:

- Adopting infrastructure as code in your team
- A workflow for deploying application code
- A workflow for deploying infrastructure code
- Putting it all together

Let's go through these topics one at a time.

EXAMPLE CODE

As a reminder, you can find all of the code examples in the book on GitHub.

Adopting IaC in Your Team

If your team is used to managing all of your infrastructure by hand, switching to infrastructure as code requires more than just introducing a new tool or technology. It also requires changing the culture and processes of the team. Changing culture and process is a significant undertaking, especially at larger companies. Because every team's culture and process is a little different, there's no one-size-fits-all way to do it, but here are a few tips that will be useful in most situations:

- Convince your boss
- Work incrementally
- Give your team the time to learn

Convince Your Boss

I've seen this story play out many times at many companies: a developer discovers Terraform, becomes inspired by what it can do, shows up to work full of enthusiasm and excitement, shows Terraform to everyone… and the boss says "no." The developer, of course, becomes frustrated and discouraged. Why doesn't everyone else see the benefits of this? We could automate everything! We could avoid so many bugs! How else can we pay down all this tech debt? How can you all be so blind??

The problem is that although this developer sees all the benefits of adopting an IaC tool such as Terraform, they aren't seeing all the costs. Here are just a few of the costs of adopting IaC:

Skills gap

The move to IaC means that your Ops team will need to spend most of its time writing large amounts of code: Terraform modules, Go tests, Chef recipes, and so on. Whereas some Ops engineers are comfortable with coding all day and will love the change, others will find this a tough transition. Many Ops engineers and sysadmins are used to making changes manually, with perhaps an occasional short script here or there, and the move to doing software engineering nearly full time might require learning a number of new skills or hiring new people.

New tools

Software developers can become attached to the tools they use; some are nearly religious about it. Every time you introduce a new tool, some developers will be thrilled at the opportunity to learn something new, but others will prefer to stick to what they know and may resist having to invest lots of time and energy learning new languages and techniques.

Change in mindset

If your team members are used to managing infrastructure manually, they are used to making all of their changes *directly*: for example, by SSHing to a server and executing a few commands. The move to IaC requires a shift in mindset where you make all of your changes *indirectly*, first by editing code, then checking it in, and then letting some automated process apply the changes. This layer of indirection can be frustrating; for simple tasks, it'll feel slower than the direct option, especially when you're still learning a new IaC tool and are not efficient with it.

Opportunity cost

> If you choose to invest your time and resources in one project, you are implicitly choosing not to invest that time and resources in other projects. What projects will have to be put on hold so that you can migrate to IaC? How important are those projects?

Some developers on your team will look at this list and become excited. But many others will groan—including your boss. Learning new skills, mastering new tools, and adopting new mindsets may or may not be beneficial, but one thing is certain: it is not free. Adopting IaC is a significant investment, and as with any investment, you need to consider not only the potential upside but also the potential downsides.

Your boss in particular will be sensitive to the opportunity cost. One of the key responsibilities of any manager is to make sure the team is working on the highest-priority projects. When you show up and excitedly start talking about Terraform, what your boss might really be hearing is, "Oh no, this sounds like a massive undertaking. How much time is it going to take?" It's not that your boss is blind to what Terraform can do, but if you are spending time on that, you might not have time to deploy the new app the search team has been asking about for months, or to prepare for the Payment Card Industry (PCI) audit, or to dig into the outage from last week. So, if you want to convince your boss that your team should adopt IaC, your goal is not to prove that it has value but that it will bring more value to your team than anything else you could work on during that time.

One of the least effective ways to do this is to just list the features of your favorite IaC tool: for example, Terraform is declarative, it's popular, it's open source. This is one of many areas where developers would do well to learn from salespeople. Most salespeople know that focusing on features is typically an ineffective way to sell products. A slightly better technique is to focus on benefits: that is, instead of talking about what a product can do ("product X can do Y!"), you should talk about what the customer can do by using that product ("you can do Y by using product X!"). In other words, show the customer what new superpowers your product can give them.

For example, instead of telling your boss that Terraform is declarative, talk about how your infrastructure will be far easier to maintain. Instead of talking about the fact that Terraform is popular, talk about how you'll be able to leverage lots of existing modules and plugins to get things done faster. And instead of explaining to your boss that Terraform is open

source, help your boss see how much easier it will be to hire new developers for the team from a large, active open source community.

Focusing on benefits is a great start, but the best salespeople know an even more effective strategy: focus on the problems. If you watch a great salesperson talking to a customer, you'll notice that it's actually the customer that does most of the talking. The salesperson spends most of their time listening and looking for one specific thing: What is the key problem that customer is trying to solve? What's the biggest pain point? Instead of trying to sell some sort of features or benefits, the best salespeople try to solve their customer's problems. If that solution happens to include the product they are selling, all the better, but the real focus is on problem solving, not selling.

Talk to your boss and try to understand the most important problems they are working on that quarter or that year. You might find that those problems would not be solved by IaC. And that's OK! It might be slightly heretical for the author of a book on Terraform to say this, but not every team needs IaC. Adopting IaC has a relatively high cost, and although it will pay off in the long term for some scenarios, it won't for others; for example, if you're at a tiny startup with just one Ops person, or you're working on a prototype that might be thrown away in a few months, or you're just working on a side project for fun, managing infrastructure by hand is often the right choice. Sometimes, even if IaC would be a great fit for your team, it won't be the highest priority, and given limited resources, working on other projects might still be the right choice.

If you do find that one of the key problems your boss is focused on can be solved with IaC, then your goal is to show your boss what that world looks like. For example, perhaps the biggest issue your boss is focused on this quarter is improving uptime. You've had numerous outages the last few months, many hours of downtime, customers are complaining, and the CEO is breathing down your manager's neck, checking in daily to see how things are going. You dig in and find out that more than half of these outages were caused by a manual error during deployment: e.g., someone accidentally skipped an important step during the rollout process, or a server was misconfigured, or the infrastructure in staging didn't match what you had in production.

Now, when you talk to your boss, instead of talking about Terraform features or benefits, lead with the following: "I have an idea for how to reduce our outages by half." I guarantee this will get your boss's attention. Use this opportunity to paint a picture for your boss of a world in which your deployment process is fully automated, reliable, and repeatable so

that the manual errors that caused half of your previous outages are no longer possible. Not only that, but if deployment is automated, you can also add automated tests, reducing outages further and allowing the whole company to deploy twice as often. Let your boss dream of being the one to tell the CEO that they've managed to cut outages in half and double deployments. And then mention that, based on your research, you believe you can deliver this future world using Terraform.

There's no guarantee that your boss will say yes, but your odds are quite a bit higher with this approach. And your odds get even better if you work incrementally.

Work Incrementally

One of the most important lessons I've learned in my career is that most large software projects fail. Whereas roughly 3 out of 4 small IT projects (less than $1 million) are completed successfully, only 1 out of 10 large projects (greater than $10 million) are completed on time and on budget, and more than one-third of large projects are never completed at all.[1]

This is why I always get worried when I see a team try to not only adopt IaC but to do so all at once, across a huge amount of infrastructure, across every team, and often as part of an even bigger initiative. I can't help but shake my head when I see the CEO or CTO of a large company give marching orders that everything must be migrated to the cloud, the old datacenters must be shut down, and that everyone will "do DevOps" (whatever that means), all within six months. I'm not exaggerating when I say that I've seen this pattern several dozen times, and without exception, every single one of these initiatives has failed. Inevitably, two to three years later, every one of these companies is still working on the migration, the old datacenter is still running, and no one can tell whether they are really doing DevOps.

If you want to successfully adopt IaC, or if you want to succeed at any other type of migration project, the only sane way to do it is incrementally. The key to *incrementalism* is not just splitting up the work into a series of small steps but splitting up the work in such a way that every step brings its own value—even if the later steps never happen.

To understand why this is so important, consider the opposite, *false incrementalism*.[2] Suppose that you do a huge migration project, broken up into several small steps, but the project doesn't offer any real value until the very final step is completed. For example, the first step is to rewrite the frontend, but you don't launch it, because it relies on a new backend.

Then, you rewrite the backend, but you don't launch that either, because it doesn't work until data is migrated to a new data store. And then, finally, the last step is to do the data migration. Only after this last step do you finally launch everything and begin realizing any value from doing all this work. Waiting until the very end of a project to get any value is a big risk. If that project is canceled or put on hold or significantly changed partway through, you might get zero value out of it, despite a lot of investment.

In fact, this is exactly what happens with many large migration projects. The project is big to begin with, and like most software projects, it takes much longer than expected. During that time, market conditions change, or the original stakeholders lose patience (e.g., the CEO was OK with spending three months to clean up tech debt, but after 12 months, it's time to begin shipping new products), and the project ends up getting canceled before completion. With false incrementalism, this gives you the worst possible outcome: you've paid a huge cost and received absolutely nothing in return.

Therefore, incrementalism is essential. You want each part of the project to deliver some value so that even if the project doesn't finish, no matter what step you got to, it was still worth doing. The best way to accomplish this is to focus on solving one, small, concrete problem at a time. For example, instead of trying to do a "big bang" migration to the cloud, try to identify one, small, specific app or team that is struggling, and work to migrate just them. Or instead of trying to do a "big bang" move to "DevOps," try to identify a single, small, concrete problem (e.g., outages during deployment) and put in place a solution for that specific problem (e.g., automate the most problematic deployment with Terraform).

If you can get a quick win by fixing one real, concrete problem right away, and making one team successful, you'll begin to build momentum. That team can become your cheerleader and help convince other teams to migrate, too. Fixing the specific deployment issue can make the CEO happy and get you support to use IaC for more projects. This will allow you to go for another quick win, and another one after that. And if you can keep repeating this process—delivering value early and often—you'll be far more likely to succeed at the larger migration effort. But even if the larger migration doesn't work out, at least one team is more successful now and one deployment process works better, so it was still worth the investment.

Give Your Team the Time to Learn

I hope that, at this point, it's clear that adopting IaC can be a significant investment. It's not something that will happen overnight. It's not something that will happen magically, just because the manager gives you a nod. It will happen only through a deliberate effort of getting everyone on board, making learning resources (e.g., documentation, video tutorials, and, of course, this book!) available, and providing dedicated time for team members to ramp up.

If your team doesn't get the time and resources that it needs, then your IaC migration is unlikely to be successful. No matter how nice your code is, if your entire team isn't on board with it, here's how it will play out:

1. One developer on the team is passionate about IaC and spends a few months writing beautiful Terraform code and using it to deploy lots of infrastructure.
2. The developer is happy and productive, but unfortunately, the rest of the team did not get the time to learn and adopt Terraform.
3. Then, the inevitable happens: an outage. One of your team members needs to deal with it, and they have two options: either (A) fix the outage the way they've always done it, by making changes manually, which takes a few minutes, or (B) fix the outage by using Terraform, but they aren't familiar with it, so this could take hours or days. Your team members are probably reasonable, rational people and will almost always choose option A.
4. Now, as a result of the manual change, the Terraform code no longer matches what's actually deployed. Therefore, next time someone on your team tries to use Terraform, there's a chance that they will get a weird error. If they do, they will lose trust in the Terraform code and once again fall back to option A, making more manual changes. This makes the code even more out of sync with reality, so the odds of the next person getting a weird Terraform error are even higher, and you quickly get into a cycle in which team members make more and more manual changes.
5. In a remarkably short time, everyone is back to doing everything manually, the Terraform code is completely unusable, and the months spent writing it are a total waste.

This scenario isn't hypothetical but something I've seen happen at many different companies. They have large, expensive codebases full of beautiful Terraform code that are just gathering dust. To avoid this scenario, you need to not only convince your boss that you should use Terraform but also give everyone on the team the time they need to learn the tool

and internalize how to use it so that when the next outage happens, it's easier to fix it in code than it is to do it by hand.

One thing that can help teams adopt IaC faster is to have a well-defined process for using it. When you're learning or using IaC on a small team, running it ad hoc on a developer's computer is good enough. But as your company and IaC usage grows, you'll want to define a more systematic, repeatable, automated workflow for how deployments happen.

A Workflow for Deploying Application Code

In this section, I'll introduce a typical workflow for taking application code (e.g., a Ruby on Rails or Java/Spring app) from development all the way to production. This workflow is reasonably well understood in the DevOps industry, so you'll probably be familiar with parts of it. Later in this chapter, I'll talk about a workflow for taking infrastructure code (e.g., Terraform modules) from development to production. This workflow is not nearly as well known in the industry, so it will be helpful to compare that workflow side by side with the application workflow to understand how to translate each application code step to an analogous infrastructure code step.

Here's what the application code workflow looks like:

1. Use version control.
2. Run the code locally.
3. Make code changes.
4. Submit changes for review.
5. Run automated tests.
6. Merge and release.
7. Deploy.

Let's go through these steps one at a time.

Use Version Control

All of your code should be in version control. No exceptions. It was the #1 item on the classic Joel Test when Joel Spolsky created it more than 20 years ago, and the only things that have changed since then are that (a) with tools like GitHub, it's easier than ever to use version control and (b) you can represent more and more things as code. This includes documentation (e.g., a README written in Markdown), application configuration

(e.g., a config file written in YAML), specifications (e.g., test code written with RSpec), tests (e.g., automated tests written with JUnit), databases (e.g., schema migrations written in ActiveRecord), and of course, infrastructure.

As in the rest of this book, I'm going to assume that you're using Git for version control. For example, here is how you can check out the code repo for this book:

```
$ git clone https://github.com/brikis98/terraform-up-and-running-code.git
```

By default, this checks out the `main` branch of your repo, but you'll most likely do all of your work in a separate branch. Here's how you can create a branch called `example-feature` and switch to it by using the `git checkout` command:

```
$ cd terraform-up-and-running-code
$ git checkout -b example-feature
Switched to a new branch 'example-feature'
```

Run the Code Locally

Now that the code is on your computer, you can run it locally. You may recall the Ruby web server example from Chapter 9, which you can run as follows:

```
$ cd code/ruby/10-terraform/team
$ ruby web-server.rb

[2019-06-15 15:43:17] INFO  WEBrick 1.3.1
[2019-06-15 15:43:17] INFO  ruby 2.3.7 (2018-03-28) [universal.x86_64-darwi
[2019-06-15 15:43:17] INFO  WEBrick::HTTPServer#start: pid=28618 port=8000
```

Now you can manually test it with `curl`:

```
$ curl http://localhost:8000
Hello, World
```

Alternatively, you can run the automated tests:

```
$ ruby web-server-test.rb

(...)
```

```
Finished in 0.633175 seconds.
-----------------------------------------
8 tests, 24 assertions, 0 failures, 0 errors
100% passed
-----------------------------------------
```

The key thing to notice is that both manual and automated tests for application code can run completely locally on your own computer. You'll see later in this chapter that this is not true for the same part of the workflow for infrastructure changes.

Make Code Changes

Now that you can run the application code, you can begin making changes. This is an iterative process in which you make a change, rerun your manual or automated tests to see whether the change worked, make another change, rerun the tests, and so on.

For example, you can change the output of *web-server.rb* to "Hello, World v2," restart the server, and see the result:

```
$ curl http://localhost:8000
Hello, World v2
```

You might also update and rerun the automated tests. The idea in this part of the workflow is to optimize the feedback loop so that the time between making a change and seeing whether it worked is minimized.

As you work, you should regularly be committing your code, with clear commit messages explaining the changes you've made:

```
$ git commit -m "Updated Hello, World text"
```

Submit Changes for Review

Eventually, the code and tests will work the way you want them to, so it's time to submit your changes for a code review. You can do this with a separate code review tool (e.g., Phabricator or Review Board) or, if you're using GitHub, you can create a *pull request*. There are several different ways to create a pull request. One of the easiest is to `git push` your `example-feature` branch back to `origin` (that is, back to GitHub

itself), and GitHub will automatically print out a pull request URL in the log output:

```
$ git push origin example-feature

(...)

remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'example-feature' on GitHub by visiting:
remote:      https://github.com/<OWNER>/<REPO>/pull/new/example-feature
remote:
```

Open that URL in your browser, fill out the pull request title and description, and then click Create. Your team members will now be able to review the changes, as shown in Figure 10-1.

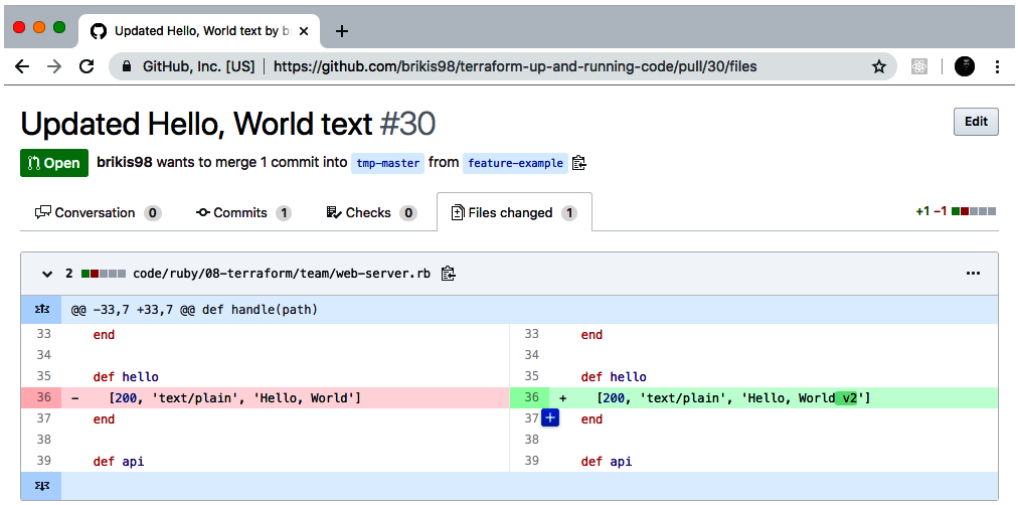

Figure 10-1. Your team members can review your code changes in a GitHub pull request.

Run Automated Tests

You should set up commit hooks to run automated tests for every commit you push to your version control system. The most common way to do this is to use a *continuous integration* (CI) server, such as Jenkins, CircleCI, or GitHub Actions. Most popular CI servers have integrations built in specifically for GitHub, so not only does every commit automatically run tests, but the output of those tests shows up in the pull request itself, as shown in Figure 10-2.

You can see in Figure 10-2 that CircleCI has run unit tests, integration tests, end-to-end tests, and some static analysis checks (in the form of security vulnerability scanning using a tool called `snyk`) against the code in the branch, and everything passed.

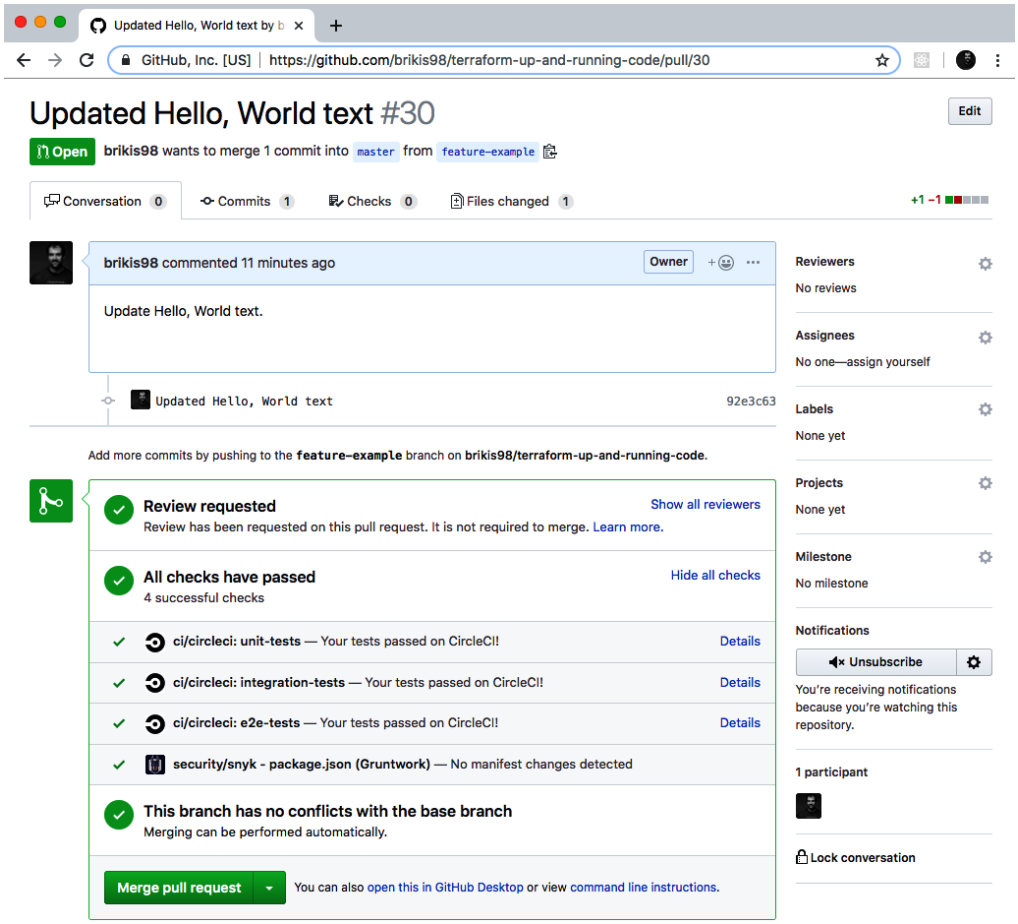

Figure 10-2. GitHub pull request showing automated test results from CircleCI.

Merge and Release

Your team members should review your code changes, looking for potential bugs, enforcing coding guidelines (more on this later in the chapter), checking that the existing tests passed, and ensuring that you've added tests for any new behavior. If everything looks good, your code can be merged into the `main` branch.

The next step is to release the code. If you're using immutable infrastructure practices (as discussed in "Server Templating Tools"), releasing application code means packaging that code into a new, immutable, versioned artifact. Depending on how you want to package and deploy your application, the artifact can be a new Docker image, a new virtual machine image (e.g., new AMI), a new *.jar* file, a new *.tar* file, etc. Whatever format you pick, make sure the artifact is immutable (i.e., you never change it) and that it has a unique version number (so you can distinguish this artifact from all of the others).

For example, if you are packaging your application using Docker, you can store the version number in a Docker tag. You could use the ID of the commit (the sha1 hash) as the tag so that you can map the Docker image you're deploying back to the exact code it contains:

```
$ commit_id=$(git rev-parse HEAD)
$ docker build -t brikis98/ruby-web-server:$commit_id .
```

The preceding code will build a new Docker image called `brikis98/ruby-web-server` and tag it with the ID of the most recent commit, which will look something like `92e3c6380ba6d1e8c9134452ab6e26154e6ad849`. Later on, if you're debugging an issue in a Docker image, you can see the exact code it contains by checking out the commit ID the Docker image has as a tag:

```
$ git checkout 92e3c6380ba6d1e8c9134452ab6e26154e6ad849
HEAD is now at 92e3c63 Updated Hello, World text
```

One downside to commit IDs is that they aren't very readable or memorable. An alternative is to create a Git tag:

```
$ git tag -a "v0.0.4" -m "Update Hello, World text"
$ git push --follow-tags
```

A tag is a pointer to a specific Git commit but with a friendlier name. You can use this Git tag on your Docker images:

```
$ git_tag=$(git describe --tags)
$ docker build -t brikis98/ruby-web-server:$git_tag .
```

Thus, when you're debugging, check out the code at a specific tag:

```
$ git checkout v0.0.4
Note: checking out 'v0.0.4'.
(...)
HEAD is now at 92e3c63 Updated Hello, World text
```

Deploy

Now that you have a versioned artifact, it's time to deploy it. There are many different ways to deploy application code, depending on the type of application, how you package it, how you want to run it, your architecture, what tools you're using, and so on. Here are a few of the key considerations:

- Deployment tooling
- Deployment strategies

- Deployment server
- Promotion across environments

Deployment tooling

There are many different tools that you can use to deploy your application, depending on how you package it and how you want to run it. Here are a few examples:

Terraform

As you've seen in this book, you can use Terraform to deploy certain types of applications. For example, in earlier chapters, you created a module called `asg-rolling-deploy` that could do a zero-downtime rolling deployment across an ASG. If you package your application as an AMI (e.g., using Packer), you could deploy new AMI versions with the `asg-rolling-deploy` module by updating the `ami` parameter in your Terraform code and running `terraform apply`.

Orchestration tools

There are a number of orchestration tools designed to deploy and manage applications, such as Kubernetes (arguably the most popular Docker orchestration tool), Amazon ECS, HashiCorp Nomad, and Apache Mesos. In Chapter 7, you saw an example of how to use Kubernetes to deploy Docker containers.

Scripts

Terraform and most orchestration tools support only a limited set of deployment strategies (discussed in the next section). If you have more complicated requirements, you may have to write custom scripts to implement these requirements.

Deployment strategies

There are a number of different strategies that you can use for application deployment, depending on your requirements. Suppose that you have five copies of the old version of your app running, and you want to roll out a new version. Here are a few of the most common strategies you can use:

Rolling deployment with replacement

Take down one of the old copies of the app, deploy a new copy to replace it, wait for the new copy to come up and pass health

checks, start sending the new copy live traffic, and then repeat the process until all of the old copies have been replaced. Rolling deployment with replacement ensures that you never have more than five copies of the app running, which can be useful if you have limited capacity (e.g., if each copy of the app runs on a physical server) or if you're dealing with a stateful system where each app has a unique identity (e.g., this is often the case with consensus systems, such as Apache ZooKeeper). Note that this deployment strategy can work with larger batch sizes (you can replace more than one copy of the app at a time if you can handle the load and won't lose data with fewer apps running) and that during deployment, you will have both the old and new versions of the app running at the same time.

Rolling deployment without replacement

Deploy one new copy of the app, wait for the new copy to come up and pass health checks, start sending the new copy live traffic, undeploy an old copy of the app, and then repeat the process until all the old copies have been replaced. Rolling deployment without replacement works only if you have flexible capacity (e.g., your apps run in the cloud, where you can spin up new virtual servers any time you want) and if your application can tolerate more than five copies of it running at the same time. The advantage is that you never have less than five copies of the app running, so you're not running at a reduced capacity during deployment. Note that this deployment strategy can also work with larger batch sizes (if you have the capacity for it, you can deploy five new copies all at once) and that during deployment, you will have both the old and new versions of the app running at the same time.

Blue-green deployment

Deploy five new copies of the app, wait for all of them to come up and pass health checks, shift all live traffic to the new copies at the same time, and then undeploy the old copies. Blue-green deployment works only if you have flexible capacity (e.g., your apps run in the cloud, where you can spin up new virtual servers any time you want) and if your application can tolerate more than five copies of it running at the same time. The advantage is that only one version of your app is visible to users at any given time and that you never have less than five copies of the app running, so you're not running at a reduced capacity during deployment.

Canary deployment

Deploy one new copy of the app, wait for it to come up and pass health checks, start sending live traffic to it, and then pause the deployment. During the pause, compare the new copy of the app, called the "canary," to one of the old copies, called the "control." You can compare the canary and control across a variety of dimensions: CPU usage, memory usage, latency, throughput, error rates in the logs, HTTP response codes, and so on. Ideally, there's no way to tell the two servers apart, which should give you confidence that the new code works just fine. In that case, you unpause the deployment and use one of the rolling deployment strategies to complete it. On the other hand, if you spot any differences, then that may be a sign of problems in the new code, and you can cancel the deployment and undeploy the canary before the problem becomes worse.

The name comes from the "canary in a coal mine" concept, where miners would take canary birds with them down into the tunnels, and if the tunnels filled with dangerous gases (e.g., carbon monoxide), those gases would affect the canary before the miners, thus providing an early warning to the miners that something was wrong and that they needed to exit immediately, before more damage was done. The canary deployment offers similar benefits, giving you a systematic way to test new code in production in a way that, if something goes wrong, you get a warning early on, when it has affected only a small portion of your users and you still have enough time to react and prevent further damage.

Canary deployments are often combined with *feature toggles*, in which you wrap all new features in an if-statement. By default, the if-statement defaults to false, so the new feature is toggled off when you initially deploy the code. Because all new functionality is off, when you deploy the canary server, it should behave identically to the control, and any differences can be automatically flagged as a problem and trigger a rollback. If there were no problems, later on you can enable the feature toggle for a portion of your users via an internal web interface. For example, you might initially enable the new feature only for employees; if that works well, you can enable it for 1% of users; if that's still working well, you can ramp it up to 10%; and so on. If at any point there's a problem, you can use the feature toggle to ramp the feature back down. This process allows you to separate *deployment* of new code from *release* of new features.

Deployment server

You should run the deployment from a CI server and not from a developer's computer. This has the following benefits:

Fully automated

To run deployments from a CI server, you'll be forced to fully automate all deployment steps. This ensures that your deployment process is captured as code, that you don't miss any steps accidentally due to manual error, and that the deployment is fast and repeatable.

Consistent environment

If developers run deployments from their own computers, you'll run into bugs due to differences in how their computer is configured: for example, different operating systems, different dependency versions (different versions of Terraform), different configurations, and differences in what's actually being deployed (e.g., the developer accidentally deploys a change that wasn't committed to version control). You can eliminate all of these issues by deploying everything from the same CI server.

Better permissions management

Instead of giving every developer permissions to deploy, you can give solely the CI server those permissions (especially for the production environment). It's a lot easier to enforce good security practices for a single server than it is to do for numerous developers with production access.

Promotion across environments

If you're using immutable infrastructure practices, the way to roll out new changes is to promote the exact same versioned artifact from one environment to another. For example, if you have dev, staging, and production environments, to roll out `v0.0.4` of your app, you would do the following:

1. Deploy `v0.0.4` of the app to dev.
2. Run your manual and automated tests in dev.
3. If `v0.0.4` works well in dev, repeat steps 1 and 2 to deploy `v0.0.4` to staging (this is known as *promoting* the artifact).
4. If `v0.0.4` works well in staging, repeat steps 1 and 2 again to promote `v0.0.4` to prod.

Because you're running the exact same artifact everywhere, there's a good chance that if it works in one environment, it will work in another. And if you do hit any issues, you can roll back anytime by deploying an older artifact version.

A Workflow for Deploying Infrastructure Code

Now that you've seen the workflow for deploying application code, it's time to dive into the workflow for deploying infrastructure code. In this section, when I say "infrastructure code," I mean code written with any IaC tool (including, of course, Terraform) that you can use to deploy arbitrary infrastructure changes beyond a single application: for example, deploying databases, load balancers, network configurations, DNS settings, and so on.

Here's what the infrastructure code workflow looks like:

1. Use version control
2. Run the code locally
3. Make code changes
4. Submit changes for review
5. Run automated tests
6. Merge and release
7. Deploy

On the surface, it looks identical to the application workflow, but under the hood, there are important differences. Deploying infrastructure code changes is more complicated, and the techniques are not as well understood, so being able to relate each step back to the analogous step from the application code workflow should make it easier to follow along. Let's dive in.

Use Version Control

Just as with your application code, all of your infrastructure code should be in version control. This means that you'll use `git clone` to check out your code, just as before. However, version control for infrastructure code has a few extra requirements:

- *Live* repo and *modules* repo
- Golden Rule of Terraform
- The trouble with branches

Live repo and modules repo

As discussed in Chapter 4, you will typically want at least two separate version control repositories for your Terraform code: one repo for modules and one repo for live infrastructure. The repository for modules is where you create your reusable, versioned modules, such as all the modules you built in the previous chapters of this book (`cluster/asg-rolling-deploy` , `data-stores/mysql` , `networking/alb` , and `services/hello-world-app`). The repository for live infrastructure defines the live infrastructure you've deployed in each environment (dev, stage, prod, etc.).

One pattern that works well is to have one infrastructure team in your company that specializes in creating reusable, robust, production-grade modules. This team can create remarkable leverage for your company by building a library of modules that implement the ideas from Chapter 8; that is, each module has a composable API, is thoroughly documented (including executable documentation in the *examples* folder), has a comprehensive suite of automated tests, is versioned, and implements all of your company's requirements from the production-grade infrastructure checklist (i.e., security, compliance, scalability, high availability, monitoring, and so on).

If you build such a library (or you buy one off the shelf[3]), all the other teams at your company will be able to consume these modules, a bit like a service catalog, to deploy and manage their own infrastructure, without (a) each team having to spend months assembling that infrastructure from scratch or (b) the Ops team becoming a bottleneck because it must deploy and manage the infrastructure for every team. Instead, the Ops team can spend most of its time writing infrastructure code, and all of the other teams will be able to work independently, using these modules to get themselves up and running. And because every team is using the same canonical modules under the hood, as the company grows and requirements change, the Ops team can push out new versions of the modules to all teams, ensuring everything stays consistent and maintainable.

Or it will be maintainable, as long as you follow the Golden Rule of Terraform.

The Golden Rule of Terraform

Here's a quick way to check the health of your Terraform code: go into your *live* repository, pick several folders at random, and run `terraform plan` in each one. If the output is always "no changes," that's great, be-

cause it means that your infrastructure code matches what's actually deployed. If the output sometimes shows a small diff, and you hear the occasional excuse from your team members ("Oh, right, I tweaked that one thing by hand and forgot to update the code"), your code doesn't match reality, and you might soon be in trouble. If `terraform plan` fails completely with weird errors, or every `plan` shows a gigantic diff, your Terraform code has no relation at all to reality and is likely useless.

The gold standard, or what you're really aiming for, is what I call *The Golden Rule of Terraform*:

> *The main branch of the live repository should be a 1:1 representation of what's actually deployed in production.*

Let's break this sentence down, starting at the end and working our way back:

"...what's actually deployed"

The only way to ensure that the Terraform code in the *live* repository is an up-to-date representation of what's actually deployed is to *never make out-of-band changes*. After you begin using Terraform, do not make changes via a web UI, or manual API calls, or any other mechanism. As you saw in Chapter 5, out-of-band changes not only lead to complicated bugs, but they also void many of the benefits you get from using IaC in the first place.

"...a 1:1 representation..."

If I browse your *live* repository, I should be able to see, from a quick scan, what resources have been deployed in what environments. That is, every resource should have a 1:1 match with some line of code checked into the *live* repo. This seems obvious at first glance, but it's surprisingly easy to get it wrong. One way to get it wrong, as I just mentioned, is to make out-of-band changes so that the code is there, but the live infrastructure is different. A more subtle way to get it wrong is to use Terraform workspaces to manage environments so that the live infrastructure is there, but the code isn't. That is, if you use workspaces, your *live* repo will have only one copy of the code, even though you may have 3 or 30 environments deployed with it. From merely looking at the code, there will be no way to know what's actually deployed, which will lead to mistakes and make maintenance complicated. Therefore, as described in "Isolation via Workspaces", instead of using workspaces to manage environments, you want each environment defined in a separate folder, using separate files, so that you can see exactly

what environments have been deployed just by browsing the *live* repository. Later in this chapter, you'll see how to do this with minimal copying and pasting.

"The main branch..."

You should have to look at only a single branch to understand what's actually deployed in production. Typically, that branch will be `main`. This means that all changes that affect the production environment should go directly into `main` (you can create a separate branch but only to create a pull request with the intention of merging that branch into `main`), and you should run `terraform apply` only for the production environment against the `main` branch. In the next section, I'll explain why.

The trouble with branches

In Chapter 3, you saw that you can use the locking mechanisms built into Terraform backends to ensure that if two team members are running `terraform apply` at the same time on the same set of Terraform configurations, their changes do not overwrite each other. Unfortunately, this only solves part of the problem. Even though Terraform backends provide locking for Terraform state, they cannot help you with locking at the level of the Terraform code itself. In particular, if two team members are deploying the same code to the same environment but from different branches, you'll run into conflicts that locking can't prevent.

For example, suppose that one of your team members, Anna, makes some changes to the Terraform configurations for an app called "foo" that consists of a single EC2 Instance:

```
resource "aws_instance" "foo" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
}
```

The app is getting a lot of traffic, so Anna decides to change the `instance_type` from `t2.micro` to `t2.medium`:

```
resource "aws_instance" "foo" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.medium"
}
```

Here's what Anna sees when she runs `terraform plan`:

```
$ terraform plan

(...)

Terraform will perform the following actions:

  # aws_instance.foo will be updated in-place
  ~ resource "aws_instance" "foo" {
        ami                           = "ami-0fb653ca2d3203ac1"
        id                            = "i-096430d595c80cb53"
        instance_state                = "running"
      ~ instance_type                 = "t2.micro" -> "t2.medium"
        (...)
    }

Plan: 0 to add, 1 to change, 0 to destroy.
```

Those changes look good, so she deploys them to staging.

In the meantime, Bill comes along and also starts making changes to the Terraform configurations for the same app but on a different branch. All Bill wants to do is to add a tag to the app:

```
resource "aws_instance" "foo" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  tags = {
    Name = "foo"
  }
}
```

Note that Anna's changes are already deployed in staging, but because they are on a different branch, Bill's code still has the `instance_type` set to the old value of `t2.micro`. Here's what Bill sees when he runs the `plan` command (the following log output is truncated for readability):

```
$ terraform plan

(...)

Terraform will perform the following actions:

  # aws_instance.foo will be updated in-place
  ~ resource "aws_instance" "foo" {
        ami                           = "ami-0fb653ca2d3203ac1"
        id                            = "i-096430d595c80cb53"
```

```
          instance_state                = "running"
        ~ instance_type                 = "t2.medium" -> "t2.micro"
        + tags                          = {
            + "Name" = "foo"
          }
          (...)
      }

Plan: 0 to add, 1 to change, 0 to destroy.
```

Uh oh, he's about to undo Anna's `instance_type` change! If Anna is still testing in staging, she'll be very confused when the server suddenly redeploys and starts behaving differently. The good news is that if Bill diligently reads the `plan` output, he can spot the error before it affects Anna. Nevertheless, the point of the example is to highlight what happens when you deploy changes to a shared environment from different branches.

The locking from Terraform backends doesn't help here, because the conflict has nothing to do with concurrent modifications to the state file; Bill and Anna might be applying their changes weeks apart, and the problem would be the same. The underlying cause is that branching and Terraform are a bad combination. Terraform is implicitly a mapping from Terraform code to infrastructure deployed in the real world. Because there's only one real world, it doesn't make much sense to have multiple branches of your Terraform code. So for any shared environment (e.g., stage, prod), always deploy from a single branch.

Run the Code Locally

Now that you've got the code checked out onto your computer, the next step is to run it. The gotcha with Terraform is that, unlike application code, you don't have "localhost"; for example, you can't deploy an AWS ASG onto your own laptop. As discussed in "Manual Testing Basics", the only way to manually test Terraform code is to run it in a sandbox environment, such as an AWS account dedicated for developers (or better yet, one AWS account for each developer).

Once you have a sandbox environment, to test manually, you run `terraform apply`:

```
$ terraform apply

(...)
```

```
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:

alb_dns_name = "hello-world-stage-477699288.us-east-2.elb.amazonaws.com"
```

And you verify the deployed infrastructure works by using tools such as `curl`:

```
$ curl hello-world-stage-477699288.us-east-2.elb.amazonaws.com
Hello, World
```

To run automated tests written in Go, you use `go test` in a sandbox account dedicated to testing:

```
$ go test -v -timeout 30m

(...)

PASS
ok      terraform-up-and-running        229.492s
```

Make Code Changes

Now that you can run your Terraform code, you can iteratively begin to make changes, just as with application code. Every time you make a change, you can rerun `terraform apply` to deploy those changes and rerun `curl` to see whether those changes worked:

```
$ curl hello-world-stage-477699288.us-east-2.elb.amazonaws.com
Hello, World v2
```

Or you can rerun `go test` to make sure the tests are still passing:

```
$ go test -v -timeout 30m

(...)

PASS
ok      terraform-up-and-running        229.492s
```

The only difference from application code is that infrastructure code tests typically take longer, so you'll want to put more thought into how you can shorten the test cycle so that you can get feedback on your changes as

quickly as possible. In "Test stages", you saw that you can use these test stages to rerun only specific stages of a test suite, dramatically shortening the feedback loop.

As you make changes, be sure to regularly commit your work:

```
$ git commit -m "Updated Hello, World text"
```

Submit Changes for Review

After your code is working the way you expect, you can create a pull request to get your code reviewed, just as you would with application code. Your team will review your code changes, looking for bugs as well as enforcing *coding guidelines*. Whenever you're writing code as a team, regardless of what type of code you're writing, you should define guidelines for everyone to follow. One of my favorite definitions of "clean code" comes from an interview I did with Nick Dellamaggiore for my earlier book, *Hello, Startup*:

> *If I look at a single file and it's written by 10 different engineers, it should be almost indistinguishable which part was written by which person. To me, that is clean code.*
>
> *The way you do that is through code reviews and publishing your style guide, your patterns, and your language idioms. Once you learn them, everybody is way more productive because you all know how to write code the same way. At that point, it's more about what you're writing and not how you write it.*
>
> —Nick Dellamaggiore, Infrastructure Lead at Coursera

The Terraform coding guidelines that make sense for each team will be different, so here, I'll list a few of the common ones that are useful for most teams:

- Documentation
- Automated tests
- File layout
- Style guide

Documentation

In some sense, Terraform code is, in and of itself, a form of documentation. It describes in a simple language exactly what infrastructure you deployed and how that infrastructure is configured. However, there is no such thing as self-documenting code. Although well-written code can tell you *what* it does, no programming language that I'm aware of (including Terraform) can tell you *why* it does it.

This is why all software, including IaC, needs documentation beyond the code itself. There are several types of documentation that you can consider and have your team members require as part of code reviews:

Written documentation

Most Terraform modules should have a README that explains what the module does, why it exists, how to use it, and how to modify it. In fact, you may want to write the README first, before any of the actual Terraform code, because that will force you to consider *what* you're building and *why* you're building it before you dive into the code and get lost in the details of *how* to build it.[4] Spending 20 minutes writing a README can often save you hours of writing code that solves the wrong problem. Beyond the basic README, you might also want to have tutorials, API documentation, wiki pages, and design documents that go deeper into how the code works and why it was built this way.

Code documentation

Within the code itself, you can use comments as a form of documentation. Terraform treats any text that begins with a hash (`#`) as a comment. Don't use comments to explain what the code does; the code should do that itself. Only include comments to offer information that can't be expressed in code, such as how the code is meant to be used or why the code uses a particular design choice. Terraform also allows every input and output variable to declare a `description` parameter, which is a great place to describe how that variable should be used.

Example code

As discussed in Chapter 8, every Terraform module should include example code that shows how that module is meant to be used. This is a great way to highlight the intended usage patterns and give your users a way to try your module without having to write any code, and it's the main way to add automated tests for the module.

Automated tests

All of Chapter 9 focuses on testing Terraform code, so I won't repeat any of that here, other than to say that infrastructure code without tests is broken. Therefore, one of the most important comments you can make in any code review is "How did you test this?"

File layout

Your team should define conventions for where Terraform code is stored and the file layout you use. Because the file layout for Terraform also determines the way Terraform state is stored, you should be especially mindful of how file layout affects your ability to provide isolation guarantees, such as ensuring that changes in a staging environment cannot accidentally cause problems in production. In a code review, you might want to enforce the file layout described in "Isolation via File Layout", which provides isolation between different environments (e.g., stage and prod) and different components (e.g., a network topology for the entire environment and a single app within that environment).

Style guide

Every team should enforce a set of conventions about code style, including the use of whitespace, newlines, indentation, curly braces, variable naming, and so on. Although programmers love to debate spaces versus tabs and where the curly brace should go, the more important thing is that you are consistent throughout your codebase.

Terraform has a built-in `fmt` command that can reformat code to a consistent style automatically:

```
$ terraform fmt
```

I recommend running this command as part of a commit hook to ensure that all code committed to version control uses a consistent style.

Run Automated Tests

Just as with application code, your infrastructure code should have commit hooks that kick off automated tests in a CI server after every commit and show the results of those tests in the pull request. You already saw how to write unit tests, integration tests, and end-to-end tests for your Terraform code in Chapter 9. There's one other critical type of test you should run: `terraform plan`. The rule here is simple:

Always run `plan` before `apply`.

Terraform shows the `plan` output automatically when you run `apply`, so what this rule really means is that you should always pause and read the `plan` output! You'd be amazed at the type of errors you can catch by taking 30 seconds to scan the "diff" you get as an output. A great way to encourage this behavior is by integrating `plan` into your code review flow. For example, Atlantis is an open source tool that automatically runs `terraform plan` on commits and adds the `plan` output to pull requests as a comment, as shown in Figure 10-3.

Figure 10-3. Atlantis can automatically add the output of the `terraform plan` command as a comment on your pull requests.

Terraform Cloud and Terraform Enterprise, HashiCorp's paid tools, both support running `plan` automatically on pull requests as well.

Merge and Release

After your team members have had a chance to review the code changes and `plan` output and all the tests have passed, you can merge your changes into the `main` branch and release the code. Similar to application code, you can use Git tags to create a versioned release:

```
$ git tag -a "v0.0.6" -m "Updated hello-world-example text"
$ git push --follow-tags
```

Whereas with application code, you often have a separate artifact to deploy, such as a Docker image or VM image, since Terraform natively supports downloading code from Git, the repository at a specific tag *is* the immutable, versioned artifact you will be deploying.

Deploy

Now that you have an immutable, versioned artifact, it's time to deploy it. Here are a few of the key considerations for deploying Terraform code:

- Deployment tooling
- Deployment strategies
- Deployment server
- Promote artifacts across environments

Deployment tooling

When deploying Terraform code, Terraform itself is the main tool that you use. However, there are a few other tools that you might find useful:

Atlantis

: The open source tool you saw earlier can not only add the `plan` output to your pull requests but also allows you to trigger a `terraform apply` when you add a special comment to your pull request. Although this provides a convenient web interface for Terraform deployments, be aware that it doesn't support versioning, which can make maintenance and debugging for larger projects more difficult.

Terraform Cloud and Terraform Enterprise

: HashiCorp's paid products provide a web UI that you can use to run `terraform plan` and `terraform apply` as well as manage variables, secrets, and access permissions.

Terragrunt

: This is an open source wrapper for Terraform that fills in some gaps in Terraform. You'll see how to use it a bit later in this chapter to deploy versioned Terraform code across multiple environments with minimal copying and pasting.

Scripts

As always, you can write scripts in a general-purpose programming language such as Python or Ruby or Bash to customize how you use Terraform.

Deployment strategies

For most types of infrastructure changes, Terraform doesn't offer any built-in deployment strategies: for example, there's no way to do a blue-green deployment for a VPC change, and there's no way to feature toggle a database change. You're essentially limited to `terraform apply`, which either works or it doesn't. A small subset of changes do support deployment strategies, such as the zero-downtime rolling deployment in the `asg-rolling-deploy` module you built in previous chapters, but these are the exceptions and not the norm.

Due to these limitations, it's critical to take into account what happens when a deployment goes wrong. With an application deployment, many types of errors are caught by the deployment strategy; for example, if the app fails to pass health checks, the load balancer will never send it live traffic, so users won't be affected. Moreover, the rolling deployment or blue-green deployment strategy can automatically roll back to the previous version of the app in case of errors.

Terraform, on the other hand, *does not roll back automatically in case of errors*. In part, that's because there is no reasonable way to roll back many types of infrastructure changes: for example, if an app deployment failed, it's almost always safe to roll back to an older version of the app, but if the Terraform change you were deploying failed, and that change was to delete a database or terminate a server, you can't easily roll that back!

Therefore, you should expect errors to happen and ensure you have a first-class way to deal with them:

Retries

Certain types of Terraform errors are transient and go away if you rerun `terraform apply`. The deployment tooling you use with Terraform should detect these known errors and automatically retry after a brief pause. Terragrunt has automatic retries on known errors as a built-in feature.

Terraform state errors

Occasionally, Terraform will fail to save state after running `terraform apply`. For example, if you lose internet connectivity partway through an `apply`, not only will the `apply` fail, but Terraform won't be able to write the updated state file to your remote backend (e.g., to Amazon S3). In these cases, Terraform will save the state file on disk in a file called *errored.tfstate*. Make sure that your CI server does not delete these files (e.g., as part of cleaning up the workspace after a build)! If you can still access this file after a failed deployment, as soon as internet connectivity is restored, you can push this file to your remote backend (e.g., to S3) using the `state push` command so that the state information isn't lost:

```
$ terraform state push errored.tfstate
```

Errors releasing locks

Occasionally, Terraform will fail to release a lock. For example, if your CI server crashes in the middle of a `terraform apply`, the state will remain permanently locked. Anyone else who tries to run `apply` on the same module will get an error message saying the state is locked and showing the ID of the lock. If you're absolutely sure this is an accidentally leftover lock, you can forcibly release it using the `force-unlock` command, passing it the ID of the lock from that error message:

```
$ terraform force-unlock <LOCK_ID>
```

Deployment server

Just as with your application code, all of your infrastructure code changes should be applied from a CI server and not from a developer's computer. You can run `terraform` from Jenkins, CircleCI, GitHub Actions, Terraform Cloud, Terraform Enterprise, Atlantis, or any other reasonably secure automated platform. This gives you the same benefits as with application code: it forces you to fully automate your deployment process, it ensures deployment always happens from a consistent environment, and it gives you better control over who has permissions to access production environments.

That said, permissions to deploy infrastructure code are quite a bit trickier than for application code. With application code, you can usually give your CI server a minimal, fixed set of permissions to deploy your apps; for example, to deploy to an ASG, the CI server typically needs only a few

specific `ec2` and `autoscaling` permissions. However, to be able to deploy arbitrary infrastructure code changes (e.g., your Terraform code might try to deploy a database or a VPC or an entirely new AWS account), the CI server needs arbitrary permissions—that is, admin permissions. And that's a problem.

The reason it's a problem is that CI servers are (a) notoriously hard to secure,[5] (b) accessible to all the developers at your company, and (c) used to execute arbitrary code. Adding permanent admin permissions to this mix is just asking for trouble! You'd effectively be giving every single person on your team admin permissions and turning your CI server into a very high-value target for attackers.

There are a few things you can do to minimize this risk:

Lock the CI server down

Make it accessible solely over HTTPs, require all users to be authenticated, and follow server-hardening practices (e.g., lock down the firewall, install fail2ban, enable audit logging, etc.).

Don't expose your CI server on the public internet

That is, run the CI server in private subnets, without any public IP, so that it's accessible only over a VPN connection. That way, only users with valid network access (e.g., via a VPN certificate) can access your CI server at all. Note that this does have a drawback: webhooks from external systems won't work. For example, GitHub won't automatically be able to trigger builds in your CI server; instead, you'll need to configure your CI server to poll your version control system for updates. This is a small price to pay for a significantly more secure CI server.

Enforce an approval workflow

Configure your CI/CD pipeline to require that every deployment be approved by at least one person (other than the person who requested the deployment in the first place). During this approval step, the reviewer should be able to see both the code changes and the `plan` output, as one final check that things look OK before `apply` runs. This ensures that every deployment, code change, and `plan` output has had at least two sets of eyes on it.

Don't give the CI server permanent credentials

As you saw in Chapter 6, instead of manually managed, permanent credentials (e.g., AWS access keys copy/pasted into your CI server),

you should prefer to use authentication mechanisms that use temporary credentials, such as IAM roles and OIDC.

Don't give the CI server admin credentials

Instead, isolate the admin credentials to a totally separate, isolated *worker*: e.g., a separate server, a separate container, etc. That worker should be extremely locked down, so no developers have access to it at all, and the only thing it allows is for the CI server to trigger that worker via an extremely limited remote API. For example, that worker's API may only allow you to run specific commands (e.g., `terraform plan` and `terraform apply`), in specific repos (e.g., your live repo), in specific branches (e.g., the `main` branch), and so on. This way, even if an attacker gets access to your CI server, they still won't have access to the admin credentials, and all they can do is request a deployment on some code that's already in your version control system, which isn't nearly as much of a catastrophe as leaking the admin credentials fully.[6]

Promote artifacts across environments

Just as with application artifacts, you'll want to promote your immutable, versioned infrastructure artifacts from environment to environment: for example, promote `v0.0.6` from dev to stage to prod.[7] The rule here is also simple:

Always test Terraform changes in pre-prod before prod.

Because everything is automated with Terraform anyway, it doesn't cost you much extra effort to try a change in staging before production, but it will catch a huge number of errors. Testing in pre-prod is especially important because, as mentioned earlier in this chapter, Terraform does not roll back changes in case of errors. If you run `terraform apply` and something goes wrong, you must fix it yourself. This is easier and less stressful to do if you catch the error in a pre-prod environment rather than prod.

The process for promoting Terraform code across environments is similar to the process of promoting application artifacts, except there is an extra approval step, as mentioned in the previous section, where you run `terraform plan` and have someone manually review the output and approve the deployment. This step isn't usually necessary for application deployments, as most application deployments are similar and relatively low risk. However, every infrastructure deployment can be completely different, and mistakes can be very costly (e.g., deleting a database), so

having one last chance to look at the `plan` output and review it is well worth the time.

Here's what the process looks like for promoting, for instance, `v0.0.6` of a Terraform module across the dev, stage, and prod environments:

1. Update the dev environment to `v0.0.6`, and run `terraform plan`.
2. Prompt someone to review and approve the plan; for example, send an automated message via Slack.
3. If the plan is approved, deploy `v0.0.6` to dev by running `terraform apply`.
4. Run your manual and automated tests in dev.
5. If `v0.0.6` works well in dev, repeat steps 1–4 to promote `v0.0.6` to staging.
6. If `v0.0.6` works well in staging, repeat steps 1–4 again to promote `v0.0.6` to production.

One important issue to deal with is all the code duplication between environments in the *live* repo. For example, consider the *live* repo shown in Figure 10-4.

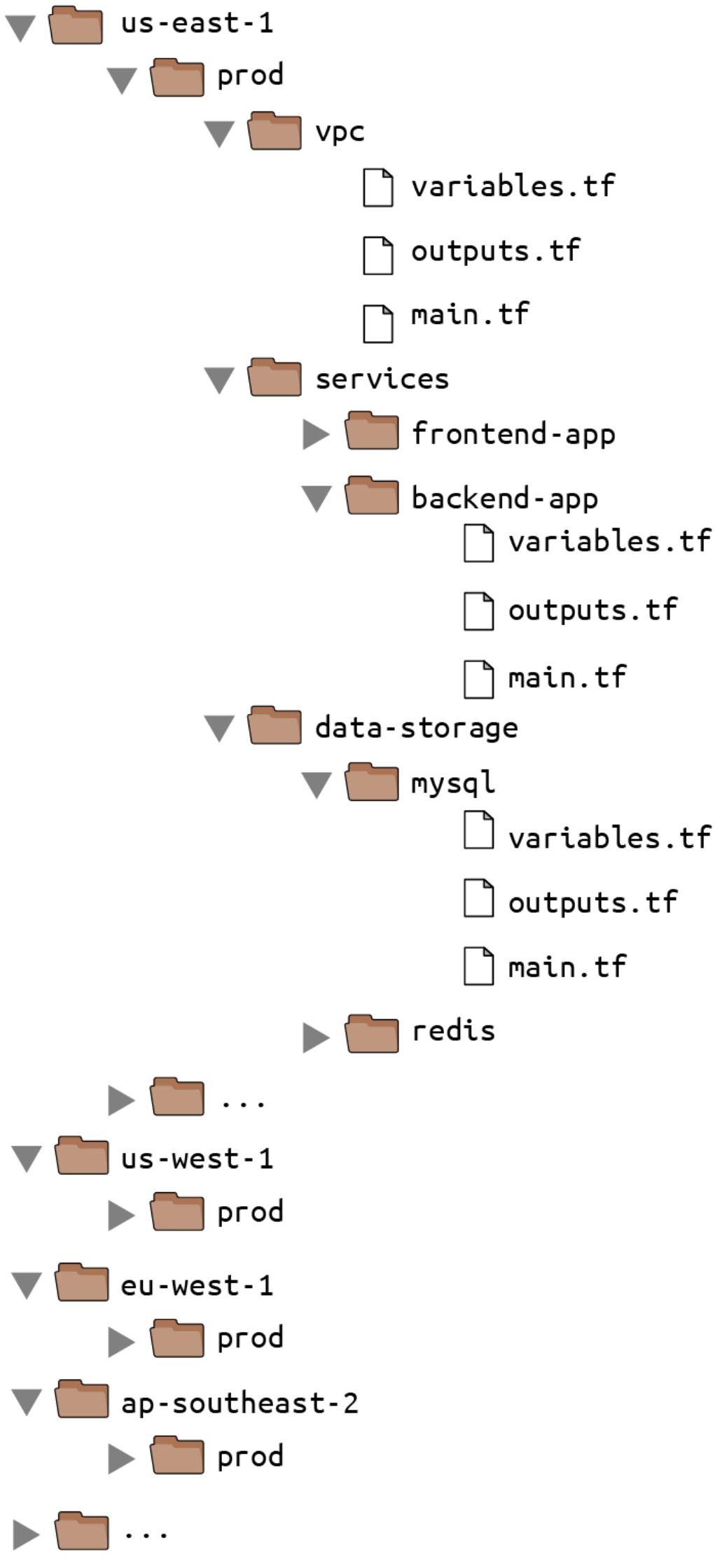
us-east-1
prod
vpc
variables.tf
outputs.tf
main.tf
services
frontend-app
backend-app
variables.tf
outputs.tf
main.tf
data-storage
mysql
variables.tf
outputs.tf
main.tf
redis
...
us-west-1
prod
eu-west-1
prod
ap-southeast-2
prod
...

Figure 10-4. File layout with a large number of copy/pasted environments and modules within each environment.

This *live* repo has a large number of regions, and within each region, a large number of modules, most of which are copied and pasted. Sure, each module has a *main.tf* that references a module in your *modules* repo, so it's not as much copying and pasting as it could be, but even if all you're doing is instantiating a single module, there is still a large amount of boilerplate that needs to be duplicated between each environment:

- The `provider` configuration
- The `backend` configuration
- The input variables to pass to the module
- The output variables to proxy from the module

This can add up to dozens or hundreds of lines of mostly identical code in each module, copied and pasted into each environment. To make this code more DRY, and to make it easier to promote Terraform code across environments, you can use the open source tool I've mentioned earlier called Terragrunt. Terragrunt is a thin wrapper for Terraform, which means that you run all of the standard `terraform` commands, except you use `terragrunt` as the binary:

```
$ terragrunt plan
$ terragrunt apply
$ terragrunt output
```

Terragrunt will run Terraform with the command you specify, but based on configuration you specify in a *terragrunt.hcl* file, you can get some extra behavior. In particular, Terragrunt allows you to define all of your Terraform code exactly once in the *modules* repo, whereas in the *live* repo, you will have solely *terragrunt.hcl* files that provide a DRY way to configure and deploy each module in each environment. This will result in a *live* repo with far fewer files and lines of code, as shown in Figure 10-5.

To get started, install Terragrunt by following the instructions on the Terragrunt website. Next, add a `provider` configuration to *modules/data-stores/mysql/main.tf* and *modules/services/hello-world-app/main.tf*:

```
provider "aws" {
  region = "us-east-2"
}
```

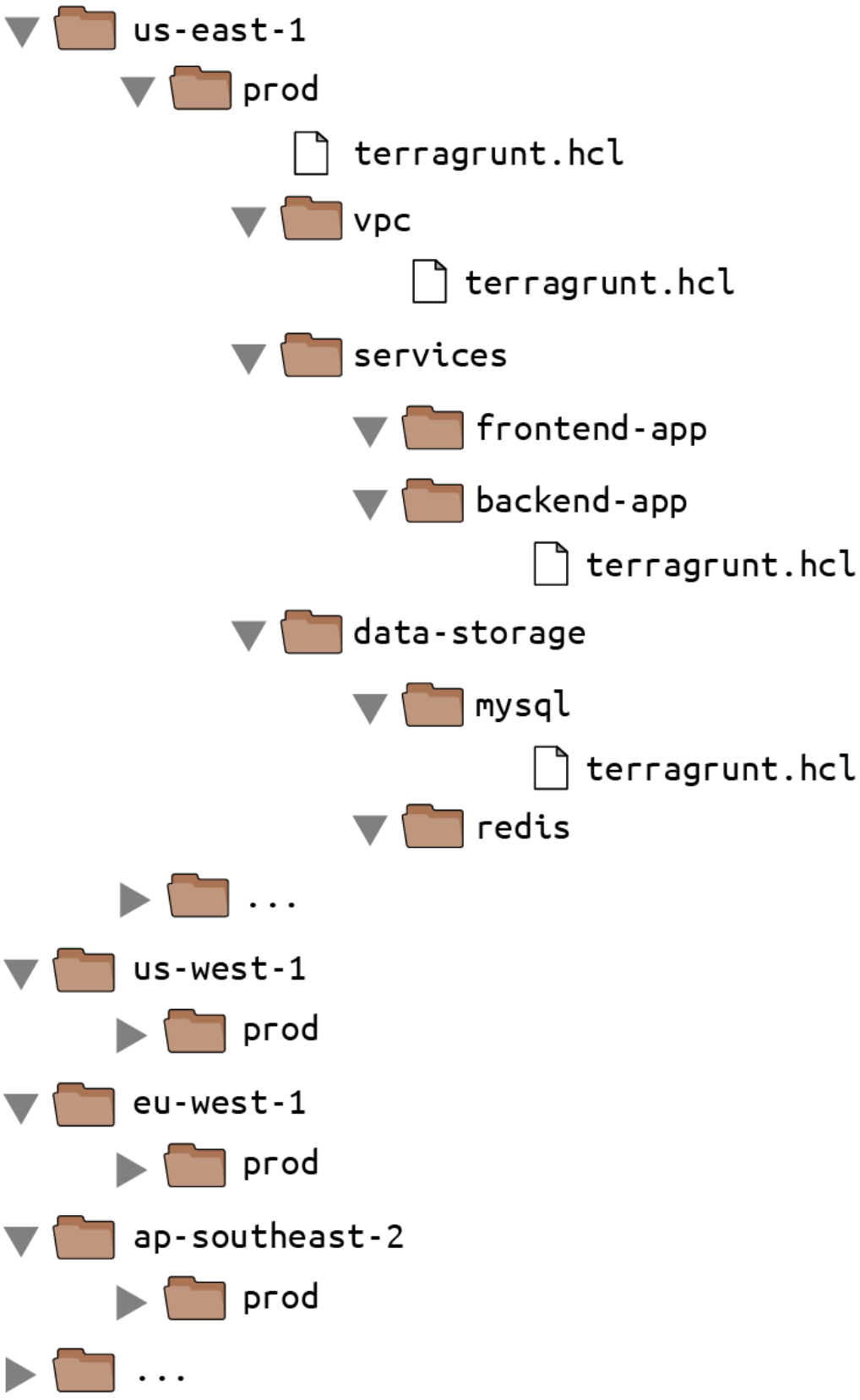

Figure 10-5. Use Terragrunt in your live repos to reduce the amount of code duplication.

Commit these changes and release a new version of your *modules* repo:

```
$ git add modules/data-stores/mysql/main.tf
$ git add modules/services/hello-world-app/main.tf
$ git commit -m "Update mysql and hello-world-app for Terragrunt"
$ git tag -a "v0.0.7" -m "Update Hello, World text"
$ git push --follow-tags
```

Now, head over to the *live* repo, and delete all the *.tf* files. You're going to replace all that copied and pasted Terraform code with a single *terragrunt.hcl* file for each module. For example, here's *terragrunt.hcl* for *live/stage/data-stores/mysql/terragrunt.hcl*:

```
terraform {
  source = "github.com/<OWNER>/modules//data-stores/mysql?ref=v0.0.7"
}

inputs = {
  db_name = "example_stage"

  # Set the username using the TF_VAR_db_username environment variable
  # Set the password using the TF_VAR_db_password environment variable
}
```

As you can see, *terragrunt.hcl* files use the same HashiCorp Configuration Language (HCL) syntax as Terraform itself. When you run `terragrunt apply` and it finds the `source` parameter in a *terragrunt.hcl* file, Terragrunt will do the following:

1. Check out the URL specified in `source` to a temporary folder. This supports the same URL syntax as the `source` parameter of Terraform modules, so you can use local file paths, Git URLs, versioned Git URLs (with a `ref` parameter, as in the preceding example), and so on.
2. Run `terraform apply` in the temporary folder, passing it the input variables that you've specified in the `inputs = { ... }` block.

The benefit of this approach is that the code in the *live* repo is reduced to just a single *terragrunt.hcl* file per module, which contains only a pointer to the module to use (at a specific version), plus the input variables to set for that specific environment. That's about as DRY as you can get.

Terragrunt also helps you keep your `backend` configuration DRY. Instead of having to define the `bucket`, `key`, `dynamodb_table`, and so on in every single module, you can define it in a single *terragrunt.hcl* file per environment. For example, create the following in *live/stage/terragrunt.hcl*:

```
remote_state {
  backend = "s3"

  generate = {
    path      = "backend.tf"
    if_exists = "overwrite"
```

```
  }

  config = {
    bucket         = "<YOUR BUCKET>"
    key            = "${path_relative_to_include()}/terraform.tfstate"
    region         = "us-east-2"
    encrypt        = true
    dynamodb_table = "<YOUR_TABLE>"
  }
}
```

From this one `remote_state` block, Terragrunt can generate the `backend` configuration dynamically for each of your modules, writing the configuration in `config` to the file specified via the `generate` param. Note that the `key` value in `config` uses a Terragrunt built-in function called `path_relative_to_include()`, which will return the relative path between this root *terragrunt.hcl* file and any child module that includes it. For example, to include this root file in *live/stage/data-stores/mysql/terragrunt.hcl*, add an `include` block:

```
terraform {
  source = "github.com/<OWNER>/modules//data-stores/mysql?ref=v0.0.7"
}

include {
  path = find_in_parent_folders()
}

inputs = {
  db_name = "example_stage"

  # Set the username using the TF_VAR_db_username environment variable
  # Set the password using the TF_VAR_db_password environment variable
}
```

The `include` block finds the root *terragrunt.hcl* using the Terragrunt built-in function `find_in_parent_folders()`, automatically inheriting all the settings from that parent file, including the `remote_state` configuration. The result is that this `mysql` module will use all the same `backend` settings as the root file, and the `key` value will automatically resolve to *data-stores/mysql/terraform.tfstate*. This means that your Terraform state will be stored in the same folder structure as your *live* repo, which will make it easy to know which module produced which state files.

To deploy this module, run `terragrunt apply`:

```
$ terragrunt apply --terragrunt-log-level debug
DEBU[0001] Reading Terragrunt config file at terragrunt.hcl
DEBU[0001] Included config live/stage/terragrunt.hcl
DEBU[0001] Downloading Terraform configurations into .terragrunt-cache
DEBU[0001] Generated file backend.tf
DEBU[0013] Running command: terraform init

(...)

Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.

(...)

DEBU[0024] Running command: terraform apply

(...)

Terraform will perform the following actions:

(...)

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

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

  Enter a value: yes

(...)

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

Normally, Terragrunt only shows the log output from Terraform itself, but as I included `--terragrunt-log-level debug`, the preceding output shows what Terragrunt does under the hood:

1. Read the *terragrunt.hcl* file in the *mysql* folder where you ran `apply`.
2. Pull in all the settings from the included root *terragrunt.hcl* file.
3. Download the Terraform code specified in the `source` URL into the *.terragrunt-cache* scratch folder.
4. Generate a *backend.tf* file with your `backend` configuration.
5. Detect that `init` has not been run and run it automatically (Terragrunt will even create your S3 bucket and DynamoDB table automat-

ically if they don't already exist).
6. Run `apply` to deploy changes.

Not bad for a couple of tiny *terragrunt.hcl* files!

You can now deploy the `hello-world-app` module in staging by adding *live/stage/services/hello-world-app/terragrunt.hcl* and running `terragrunt apply`:

```
terraform {
  source = "github.com/<OWNER>/modules//services/hello-world-app?ref=v0.0.7
}

include {
  path = find_in_parent_folders()
}

dependency "mysql" {
  config_path = "../../data-stores/mysql"
}

inputs = {
  environment = "stage"
  ami         = "ami-0fb653ca2d3203ac1"

  min_size = 2
  max_size = 2

  enable_autoscaling = false

  mysql_config = dependency.mysql.outputs
}
```

This *terragrunt.hcl* file uses the `source` URL and `inputs` just as you saw before and uses `include` to pull in the settings from the root *terragrunt.hcl* file, so it will inherit the same `backend` settings, except for the `key`, which will be automatically set to *services/hello-world-app/terraform.tfstate*, just as you'd expect. The one new thing in this *terragrunt.hcl* file is the `dependency` block:

```
dependency "mysql" {
  config_path = "../../data-stores/mysql"
}
```

This is a Terragrunt feature that can be used to automatically read the output variables of another Terragrunt module, so you can pass them as input variables to the current module, as follows:

```
    mysql_config = dependency.mysql.outputs
```

In other words, `dependency` blocks are an alternative to using `terraform_remote_state` data sources to pass data between modules. While `terraform_remote_state` data sources have the advantage of being native to Terraform, the drawback is that they make your modules more tightly coupled together, as each module needs to know how other modules store state. Using Terragrunt `dependency` blocks allows your modules to expose generic inputs like `mysql_config` and `vpc_id`, instead of using data sources, which makes the modules less tightly coupled and easier to test and reuse.

Once you've got `hello-world-app` working in staging, create analogous *terragrunt.hcl* files in *live/prod* and promote the exact same `v0.0.7` artifact to production by running `terragrunt apply` in each module.

Putting It All Together

You've now seen how to take both application code and infrastructure code from development all the way through to production. Table 10-1 shows an overview of the two workflows side by side.

Table 10-1. Application and infrastructure code workflows

	Application code	Infrastructure code
Use version control	• `git clone` • One repo per app • Use branches	• `git clone` • *live* and *modules* repos • Don't use branches
Run the code locally	• Run on localhost • `ruby web-server.rb` • `ruby web-server-test.rb`	• Run in a sandbox environment • `terraform apply` • `go test`
Make code changes	• Change the code • `ruby web-server.rb` • `ruby web-server-test.rb`	• Change the code • `terraform apply` • `go test` • Use test stages
Submit changes for review	• Submit a pull request • Enforce coding guidelines	• Submit a pull request • Enforce coding guidelines
Run automated tests	• Tests run on CI server • Unit tests • Integration tests • End-to-end tests • Static analysis	• Tests run on CI server • Unit tests • Integration tests • End-to-end tests • Static analysis • `terraform plan`
Merge and release	• `git tag` • Create versioned, immutable	• `git tag` • Use repo with tag as versioned, immutable

	Application code	Infrastructure code
	artifact	artifact
Deploy	• Deploy with Terraform, orchestration tool (e.g., Kubernetes, Mesos), scripts • Many deployment strategies: rolling deployment, blue-green, canary • Run deployment on a CI server • Give CI server limited permissions • Promote immutable, versioned artifacts across environments • Once a pull request is merged, deploy automatically	• Deploy with Terraform, Atlantis, Terraform Cloud, Terraform Enterprise, Terragrunt, scripts • Limited deployment strategies (make sure to handle errors: retries, `errored.tfstate`!) • Run deployment on a CI server • Give CI server temporary credentials solely to invoke a separate, locked-down worker that has admin permissions • Promote immutable, versioned artifacts across environments • Once a pull request is merged, go through an approval workflow where someone checks the `plan` output one last time, and then deploy automatically

If you follow this process, you will be able to run application and infrastructure code in dev, test it, review it, package it into versioned, immutable artifacts, and promote those artifacts from environment to environment, as shown in Figure 10-6.

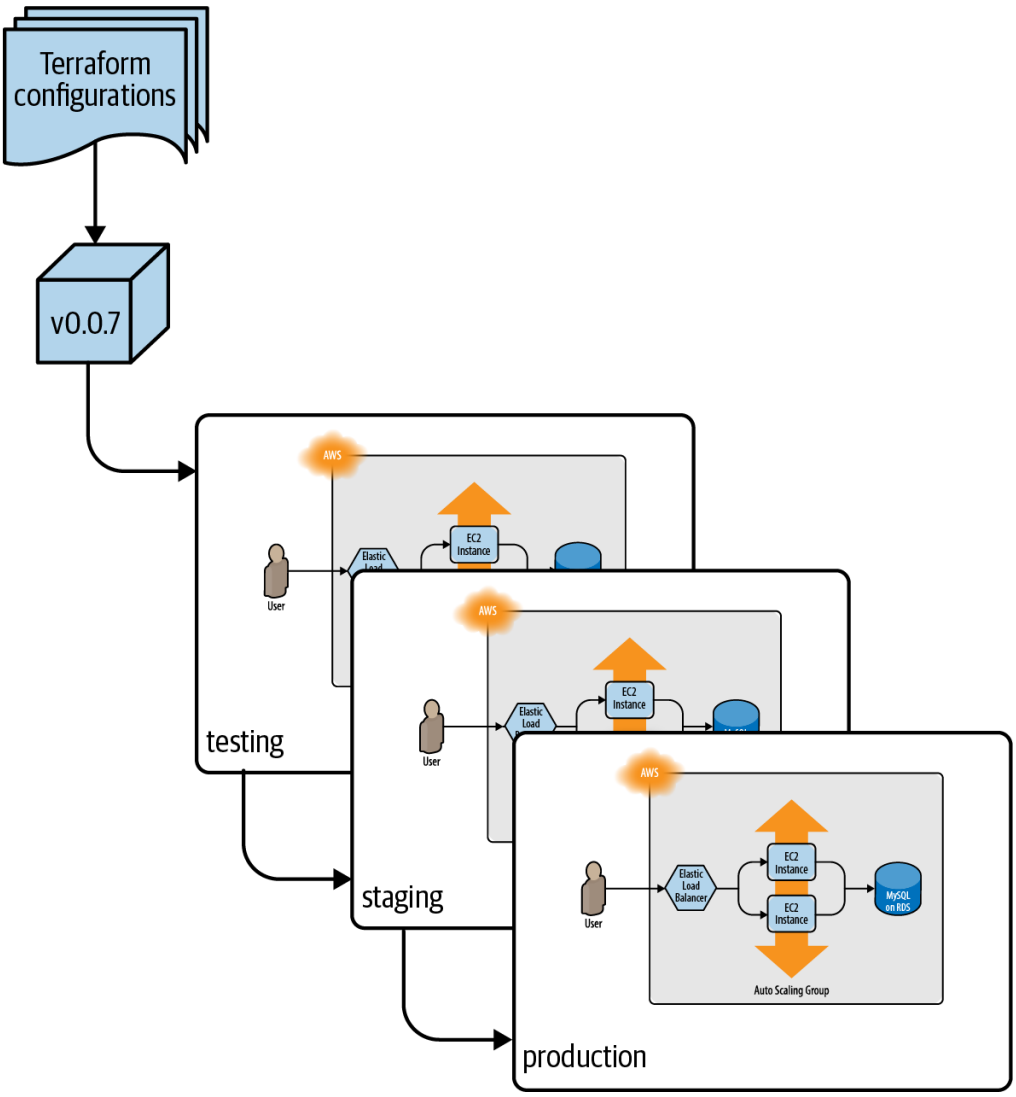

Figure 10-6. Promoting an immutable, versioned artifact of Terraform code from environment to environment.

Conclusion

If you've made it to this point in the book, you now know just about everything you need to use Terraform in the real world, including how to write Terraform code; how to manage Terraform state; how to create reusable modules with Terraform; how to do loops, if-statements, and deployments; how to manage secrets; how to work with multiple regions, accounts, and clouds; how to write production-grade Terraform code; how to test your Terraform code; and how to use Terraform as a team. You've worked through examples of deploying and managing servers, clusters of servers, load balancers, databases, scheduled actions, CloudWatch alarms, IAM users, reusable modules, zero-downtime deployment, AWS Secrets Manager, Kubernetes clusters, automated tests, and more. Phew! Just don't forget to run `terraform destroy` in each module when you're all done.

The power of Terraform, and more generally, IaC, is that you can manage all the operational concerns around an application using the same coding

principles as the application itself. This allows you to apply the full power of software engineering to your infrastructure, including modules, code reviews, version control, and automated testing.

If you use Terraform correctly, your team will be able to deploy faster and respond to changes more quickly. Hopefully, deployments will become routine and boring—and in the world of operations, boring is a very good thing. And if you really do your job right, rather than spending all your time managing infrastructure by hand, your team will be able to spend more and more time improving that infrastructure, allowing you to go even faster.

This is the end of the book but just the beginning of your journey with Terraform. To learn more about Terraform, IaC, and DevOps, head over to Appendix A for a list of recommended reading. And if you've got feedback or questions, I'd love to hear from you at jim@ybrikman.com. Thank you for reading!

1 The Standish Group, "CHAOS Manifesto 2013: Think Big, Act Small," 2013, *https://oreil.ly/ydaWQ*.

2 Dan Milstein, "How to Survive a Ground-Up Rewrite Without Losing Your Sanity," OnStartups.com, April 8, 2013, *https://oreil.ly/nOGrU*.

3 See the Gruntwork Infrastructure as Code Library.

4 Writing the README first is called Readme-Driven Development.

5 See 10 real-world stories of how we've compromised CI/CD pipelines for some eye-opening examples.

6 Check out Gruntwork Pipelines for a real-world example of this worker pattern.

7 Credit for how to promote Terraform code across environments goes to Kief Morris: Using Pipelines to Manage Environments with Infrastructure as Code.