- They only thoroughly check that servers work and not other types of infrastructure.
- These tests aren't checking functionality, so it's possible for all the checks to pass and the infrastructure still doesn't work!

Conclusion

Everything in the infrastructure world is continuously changing: Terraform, Packer, Docker, Kubernetes, AWS, Google Cloud, Azure, and so on are all moving targets. This means that infrastructure code rots very quickly. Or to put it another way:

Infrastructure code without automated tests is broken.

I mean this both as an aphorism and as a literal statement. Every single time I've gone to write infrastructure code, no matter how much effort I've put into keeping the code clean, testing it manually, and doing code reviews, as soon as I've taken the time to write automated tests, I've found numerous nontrivial bugs. Something magical happens when you take the time to automate the testing process and, almost without exception, it flushes out problems that you otherwise would've never found yourself (but your customers would've). And not only do you find these bugs when you first add automated tests, but if you run your tests after every commit, you'll keep finding bugs over time, especially as the DevOps world changes all around you.

The automated tests I've added to my infrastructure code have caught bugs not only in my own code but also in the tools I was using, including nontrivial bugs in Terraform, Packer, Elasticsearch, Kafka, AWS, and so on. Writing automated tests as shown in this chapter is *not* easy: it takes considerable effort to write these tests, it takes even more effort to maintain them and add enough retry logic to make them reliable, and it takes still more effort to keep your test environment clean to keep costs in check. But it's all worth it.

When I build a module to deploy a data store, for example, after every commit to that repo, my tests fire up a dozen copies of that data store in various configurations, write data, read data, and then tear everything back down. Each time those tests pass, that gives me huge confidence that my code still works. If nothing else, the automated tests let me sleep better. Those hours I spent dealing with retry logic and eventual consistency pay off in the hours I won't be spending at 3 a.m. dealing with an outage.

THIS BOOK HAS TESTS, TOO!

All of the code examples in this book have tests, too. You can find all of the code examples, and all of their corresponding tests, at GitHub.

Throughout this chapter, you saw the basic process of testing Terraform code, including the following key takeaways:

When testing Terraform code, you can't use localhost

> Therefore, you need to do all of your manual testing by deploying real resources into one or more isolated sandbox environments.

You cannot do pure unit testing for Terraform code

> Therefore, you have to do all of your automated testing by writing code that deploys real resources into one or more isolated sandbox environments.

Regularly clean up your sandbox environments

> Otherwise, the environments will become unmanageable, and costs will spiral out of control.

You must namespace all of your resources

> This ensures that multiple tests running in parallel do not conflict with one another.

Smaller modules are easier and faster to test

> This was one of the key takeaways in Chapter 8, and it's worth repeating in this chapter, too: smaller modules are easier to create, maintain, use, and test.

You also saw a number of different testing approaches throughout this chapter: unit testing, integration testing, end-to-end testing, static analysis, and so on. Table 9-4 shows the trade-offs between these different types of tests.

Table 9-4. A comparison of testing approaches (more black squares is better

	Static analysis	Plan testing	Server testing	Unit tests
Fast to run	■■■■■	■■■■□	■■■□□	■■□□□
Cheap to run	■■■■■	■■■■□	■■■□□	■■□□□
Stable and reliable	■■■■■	■■■■□	■■■□□	■■□□□
Easy to use	■■■■■	■■■■□	■■■□□	■■□□□
Check syntax	■■■■■	■■■■■	■■■■■	■■■■■
Check policies	■■□□□	■■■■□	■■■■□	■■■■■
Check servers work	□□□□□	□□□□□	■■■■■	■■■■■
Check other infrastructure works	□□□□□	□□□□□	■■□□□	■■■■□
Check all the infrastructure works together	□□□□□	□□□□□	□□□□□	■□□□□

So which testing approach should you use? The answer is: a mix of all of them! Each type of test has strengths and weaknesses, so you have to combine multiple types of tests to be confident your code works as expected. That doesn't mean that you use all the different types of tests in equal proportion: recall the test pyramid and how, in general, you'll typically want lots of unit tests, fewer integration tests, and only a small number of high-value end-to-end tests. Moreover, you don't have to add all the different types of tests at once. Instead, pick the ones that give you the best bang for your buck and add those first. Almost any testing is better than none, so if all you can add for now is static analysis, then use that as a starting point, and build on top of it incrementally.

Let's now move on to Chapter 10, where you'll see how to incorporate Terraform code and your automated test code into your team's workflow, including how to manage environments, how to configure a CI/CD pipeline, and more.

---

1 AWS doesn't charge anything extra for additional AWS accounts, and if you use AWS Organizations, you can create multiple "child" accounts that all roll up their billing to a single root account, as you saw in Chapter 7.

2 In limited cases, it is possible to override the endpoints Terraform uses to communicate with providers, such as overriding the endpoints Terraform uses to talk to AWS to instead talk to a mocking tool called LocalStack. This works for a small number of endpoints, but most Terraform code makes *hundreds* of different API calls to the underlying provider, and mocking out all of them is impractical. Moreover, even if you do mock them all out, it's not clear that the resulting unit test can give you much confidence that your code works correctly: e.g., if you create mock endpoints for ASGs and ALBs, your `terraform apply` might succeed, but does that tell you anything useful about whether your code would have actually deployed a working app on top of that infrastructure?