```
* error loading the remote state: RequestError: send request
failed
Post https://s3.amazonaws.com/: dial tcp 11.22.33.44:443:
connect: connection refused

(...)

'terraform [apply]' failed with the error 'exit status code 1'
but this error was expected and warrants a retry. Further
details:
Intermittent error, possibly due to throttling?

(...)

Running command terraform with args [apply -input=false -
lock=false
-auto-approve]
```

End-to-End Tests

Now that you have unit tests and integration tests in place, the final type of tests that you might want to add are *end-to-end* tests. With the Ruby web server example, end-to-end tests might consist of deploying the web server and any data stores it depends on and testing it from the web browser using a tool such as Selenium. The end-to-end tests for Terraform infrastructure will look similar: deploy everything into an environment that mimics production, and test it from the end-user's perspective.

Although you could write your end-to-end tests using the exact same strategy as the integration tests—that is, create a few dozen test stages to run `terraform apply`, do some validations, and then run `terraform destroy`—this is rarely done in practice. The reason for this has to do with the *test pyramid*, which you can see in Figure 9-1.

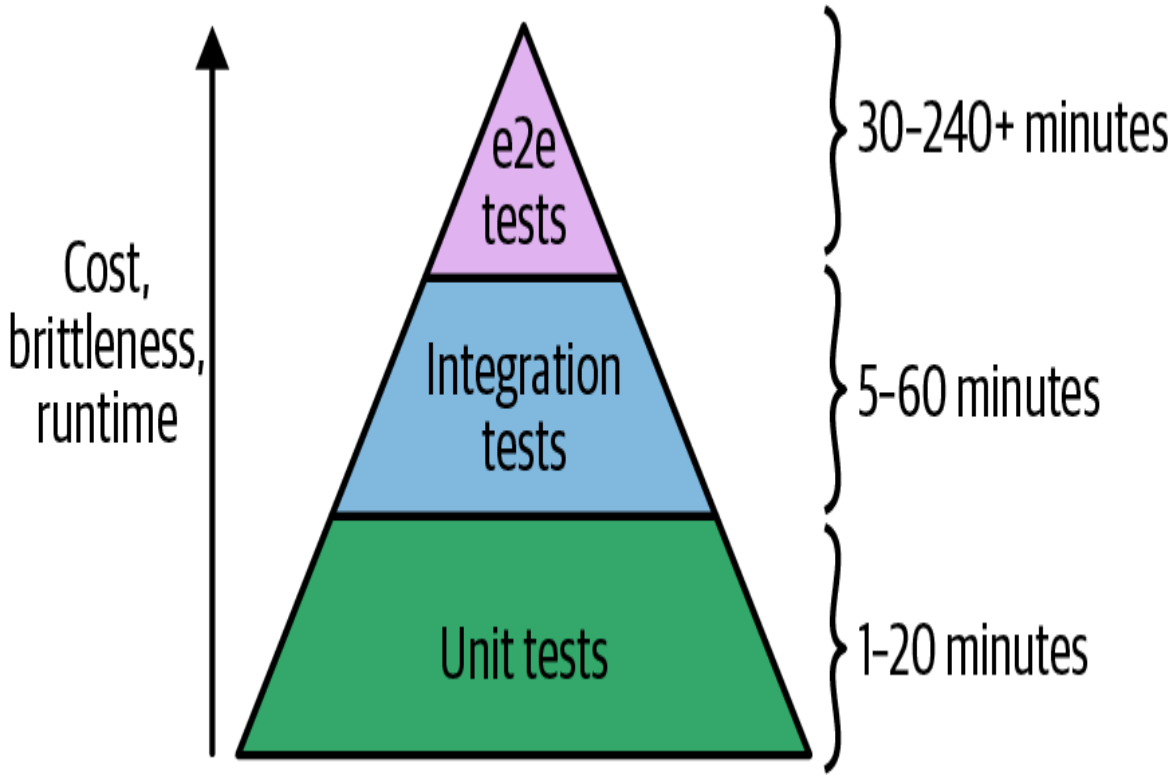

Figure 9-1. The test pyramid.

The idea of the test pyramid is that you should typically be aiming for a large number of unit tests (the bottom of the pyramid), a smaller number of integration tests (the middle of the pyramid), and an even smaller number of end-to-end tests (the top of the pyramid). This is because, as you go up the pyramid, the cost and complexity of writing the tests, the brittleness of the tests, and the runtime of the tests all increase.

That gives us *key testing takeaway #5*: smaller modules are easier and faster to test.

You saw in the previous sections that it required a fair amount of work with namespacing, dependency injection, retries, error handling, and test stages to test even a relatively simple `hello-world-app` module. With larger and more complicated infrastructure, this only becomes more difficult. Therefore, you want to do as much of your testing as low in the pyramid as you can because the bottom of the pyramid offers the fastest, most reliable feedback loop.

In fact, by the time you get to the top of the test pyramid, running tests to deploy a complicated architecture from scratch becomes untenable for two main reasons:

Too slow

Deploying your entire architecture from scratch and then undeploying it all again can take a very long time: on the order of several hours. Test suites that take that long provide relatively little value because the feedback loop is simply too slow. You'd probably run such a test suite only overnight, which means in the morning you'll get a report about a test failure, you'll investigate for a while, submit a fix, and then wait for the next day to see whether it worked. That limits you to roughly one bug fix attempt per day. In these sorts of situations, what actually happens is developers begin blaming others for test failures, convince management to deploy despite the test failures, and eventually ignore the test failures entirely.

Too brittle

As mentioned in the previous section, the infrastructure world is messy. As the amount of infrastructure you're deploying goes up, the odds of hitting an intermittent, flaky issue goes up as well. For example, suppose that a single resource (such as an EC2 Instance) has a one-in-a-thousand chance (0.1%) of failing due to an intermittent error (actual failure rates in the DevOps world are likely higher). This means that the probability that a test that deploys a single resource runs without any intermittent errors is 99.9%. So what about a test that deploys two resources? For that test to succeed, you need both resources to deploy without intermittent errors, and to calculate those odds, you multiply the probabilities: 99.9% × 99.9% = 99.8%. With three resources, the odds are 99.9% × 99.9% × 99.9% = 99.7%. With *N* resources, the formula is $99.9\%^{N}$.

So now let's consider different types of automated tests. If you had a unit test of a single module that deployed, say, 20 resources, the odds of success are $99.9\%^{20} = 98.0\%$. This means that 2 test runs out of 100

will fail; if you add a few retries, you can typically make these tests fairly reliable. Now, suppose that you had an integration test of 3 modules that deployed 60 resources. Now the odds of success are $99.9\%^{60} = 94.1\%$. Again, with enough retry logic, you can typically make these tests stable enough to be useful. So what happens if you want to write an end-to-end test that deploys your entire infrastructure, which consists of 30 modules, or about 600 resources? The odds of success are $99.9\%^{600} = 54.9\%$. This means that nearly half of your test runs will fail for transient reasons!

You'll be able to handle some of these errors with retries, but it quickly turns into a never-ending game of whack-a-mole. You add a retry for a TLS handshake timeout, only to be hit by an APT repo downtime in your Packer template; you add retries to the Packer build, only to have the build fail due to a Terraform eventual-consistency bug; just as you are applying the Band-Aid to that, the build fails due to a brief GitHub outage. And because end-to-end tests take so long, you get only one attempt, maybe two, per day to fix these issues.

In practice, very few companies with complicated infrastructure run end-to-end tests that deploy everything *from scratch*. Instead, the more common test strategy for end-to-end tests works as follows:

1. One time, you pay the cost of deploying a persistent, production-like environment called "test," and you leave that environment running.
2. Every time someone makes a change to your infrastructure code, the end-to-end test does the following:
    a. Applies the infrastructure change to the test environment.
    b. Runs validations against the test environment (e.g., uses Selenium to test your code from the end-user's perspective) to make sure everything is working.

By changing your end-to-end test strategy to applying only incremental changes, you're reducing the number of resources that are being deployed