the place, cluttering up all of your environments, and costing you a lot of money.

To keep costs from spiraling out of control, *key testing takeaway #2* is: regularly clean up your sandbox environments.

At a minimum, you should create a culture in which developers clean up whatever they deploy when they are done testing by running `terraform destroy`. Depending on your deployment environment, you might also be able to find tools that you can run on a regular schedule (e.g., a cron job) to automatically clean up unused or old resources, such as cloud-nuke and aws-nuke.

For example, a common pattern is to run `cloud-nuke` as a cron job once per day in each sandbox environment to delete all resources that are more than 48 hours old, based on the assumption that any infrastructure a developer fired up for manual testing is no longer necessary after a couple of days:

```
$ cloud-nuke aws --older-than 48h
```

> **WARNING: LOTS OF CODING AHEAD**
>
> Writing automated tests for infrastructure code is not for the faint of heart. This automated testing section is arguably the most complicated part of the book and does not make for light reading. If you're just skimming, feel free to skip this part. On the other hand, if you really want to learn how to test your infrastructure code, roll up your sleeves and get ready to write some code! You don't need to run any of the Ruby code (it's just there to help build up your mental model), but you'll want to write and run as much Go code as you can.

Automated Tests

The idea with automated testing is to write test code that validates that your real code behaves the way it should. As you'll see in Chapter 10, you can set up a CI server to run these tests after every single commit and then

immediately revert or fix any commits that cause the tests to fail, thereby always keeping your code in a working state.

Broadly speaking, there are three types of automated tests:

Unit tests

> Unit tests verify the functionality of a single, small unit of code. The definition of *unit* varies, but in a general-purpose programming language, it's typically a single function or class. Usually, any external dependencies—for example, databases, web services, even the filesystem—are replaced with *test doubles* or *mocks* that allow you to finely control the behavior of those dependencies (e.g., by returning a hardcoded response from a database mock) to test that your code handles a variety of scenarios.

Integration tests

> Integration tests verify that multiple units work together correctly. In a general-purpose programming language, an integration test consists of code that validates that several functions or classes work together correctly. Integration tests typically use a mix of real dependencies and mocks: for example, if you're testing the part of your app that communicates with the database, you might want to test it with a real database, but mock out other dependencies, such as the app's authentication system.

End-to-end tests

> End-to-end tests involve running your entire architecture—for example, your apps, your data stores, your load balancers—and validating that your system works as a whole. Usually, these tests are done from the end-user's perspective, such as using Selenium to automate interacting with your product via a web browser. End-to-end tests typically use real systems everywhere, without any mocks, in an architecture that mirrors production (albeit with fewer/smaller servers to save money).