directions and you can’t find the time to focus on it, it might take you significantly longer.

In this chapter, I’ll go over why it takes so long to build production-grade infrastructure, what production grade really means, and what patterns work best for creating reusable, production-grade modules:

- Why it takes so long to build production-grade infrastructure
- The production-grade infrastructure checklist
- Production-grade infrastructure modules
    - Small modules
    - Composable modules
    - Testable modules
    - Versioned modules
    - Beyond Terraform modules

EXAMPLE CODE

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

Why It Takes So Long to Build Production-Grade Infrastructure

Time estimates for software projects are notoriously inaccurate. Time estimates for DevOps projects, doubly so. That quick tweak that you thought would take five minutes takes up the entire day; the minor new feature that you estimated at a day of work takes two weeks; the app that you thought would be in production in two weeks is still not quite there six months later. Infrastructure and DevOps projects, perhaps more than any other type of software, are the ultimate examples of Hofstadter’s Law:[1]

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

I think there are three major reasons for this. The first reason is that DevOps, as an industry, is still in the Stone Age. I don't mean that as an insult but rather in the sense that the industry is still in its infancy. The terms "cloud computing," "infrastructure as code," and "DevOps" only appeared in the mid- to late-2000s, and tools like Terraform, Docker, Packer, and Kubernetes were all initially released in the mid- to late-2010s. All of these tools and techniques are relatively new, and all of them are changing rapidly. This means that they are not particularly mature and few people have deep experience with them, so it's no surprise that projects take longer than expected.

The second reason is that DevOps seems to be particularly susceptible to *yak shaving*. If you haven't heard of "yak shaving" before, I assure you, this is a term that you will grow to love (and hate). The best definition I've seen of this term comes from a blog post by Seth Godin:[2]

"I want to wax the car today."

"Oops, the hose is still broken from the winter. I'll need to buy a new one at Home Depot."

"But Home Depot is on the other side of the Tappan Zee bridge and getting there without my EZPass is miserable because of the tolls."

"But, wait! I could borrow my neighbor's EZPass…"

"Bob won't lend me his EZPass until I return the mooshi pillow my son borrowed, though."

"And we haven't returned it because some of the stuffing fell out and we need to get some yak hair to restuff it."

And the next thing you know, you're at the zoo, shaving a yak, all so you can wax your car.

Yak shaving consists of all the tiny, seemingly unrelated tasks you must do before you can do the task you originally wanted to do. If you develop software, and especially if you work in the DevOps industry, you've probably seen this sort of thing a thousand times. You go to deploy a fix for a small typo, only to uncover a bug in your app configuration. You try to deploy a fix for the app configuration, but that's blocked by a TLS certificate issue. After spending hours on Stack Overflow, you try to roll out a fix for the TLS issue, but that fails due to a problem with your deployment system. You spend hours digging into that problem and find out it's due to an out-of-date Linux version. The next thing you know, you're updating the operating system on your entire fleet of servers, all so you can deploy a "quick" one-character typo fix.

DevOps seems to be especially prone to these sorts of yak-shaving incidents. In part, this is a consequence of the immaturity of DevOps technologies and modern system design, which often involves lots of tight coupling and duplication in the infrastructure. Every change you make in the DevOps world is a bit like trying to pull out one wire from a box of tangled wires—it just tends to pull up everything else in the box with it. In part, this is because the term "DevOps" covers an astonishingly broad set of topics: everything from build to deployment to security and so on.

This brings us to the third reason why DevOps work takes so long. The first two reasons—DevOps is in the Stone Age and yak shaving—can be classified as accidental complexity. *Accidental complexity* refers to the problems imposed by the particular tools and processes you've chosen, as opposed to *essential complexity*, which refers to the problems inherent in whatever it is that you're working on.[3] For example, if you're using C++ to write stock-trading algorithms, dealing with memory allocation bugs is accidental complexity: had you picked a different programming language with automatic memory management, you wouldn't have this as a problem at all. On the other hand, figuring out an algorithm that can beat the market is essential complexity: you'd have to solve this problem no matter what programming language you picked.