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