Atlantis

The open source tool you saw earlier can not only add the `plan` output to your pull requests but also allows you to trigger a `terraform apply` when you add a special comment to your pull request. Although this provides a convenient web interface for Terraform deployments, be aware that it doesn't support versioning, which can make maintenance and debugging for larger projects more difficult.

Terraform Cloud and Terraform Enterprise

HashiCorp's paid products provide a web UI that you can use to run `terraform plan` and `terraform apply` as well as manage variables, secrets, and access permissions.

Terragrunt

This is an open source wrapper for Terraform that fills in some gaps in Terraform. You'll see how to use it a bit later in this chapter to deploy versioned Terraform code across multiple environments with minimal copying and pasting.

Scripts

As always, you can write scripts in a general-purpose programming language such as Python or Ruby or Bash to customize how you use Terraform.

Deployment strategies

For most types of infrastructure changes, Terraform doesn't offer any built-in deployment strategies: for example, there's no way to do a blue-green deployment for a VPC change, and there's no way to feature toggle a database change. You're essentially limited to `terraform apply`, which either works or it doesn't. A small subset of changes do support deployment strategies, such as the zero-downtime rolling deployment in the `asg-`

`rolling-deploy` module you built in previous chapters, but these are the exceptions and not the norm.

Due to these limitations, it's critical to take into account what happens when a deployment goes wrong. With an application deployment, many types of errors are caught by the deployment strategy; for example, if the app fails to pass health checks, the load balancer will never send it live traffic, so users won't be affected. Moreover, the rolling deployment or blue-green deployment strategy can automatically roll back to the previous version of the app in case of errors.

Terraform, on the other hand, *does not roll back automatically in case of errors*. In part, that's because there is no reasonable way to roll back many types of infrastructure changes: for example, if an app deployment failed, it's almost always safe to roll back to an older version of the app, but if the Terraform change you were deploying failed, and that change was to delete a database or terminate a server, you can't easily roll that back!

Therefore, you should expect errors to happen and ensure you have a first-class way to deal with them:

Retries

> Certain types of Terraform errors are transient and go away if you rerun `terraform apply`. The deployment tooling you use with Terraform should detect these known errors and automatically retry after a brief pause. Terragrunt has automatic retries on known errors as a built-in feature.

Terraform state errors

> Occasionally, Terraform will fail to save state after running `terraform apply`. For example, if you lose internet connectivity partway through an `apply`, not only will the `apply` fail, but Terraform won't be able to write the updated state file to your remote backend (e.g., to Amazon S3). In these cases, Terraform will save the state file on disk in a file called *errored.tfstate*. Make sure that your CI server does not delete these files (e.g., as part of cleaning up the workspace