This is a Terragrunt feature that can be used to automatically read the output variables of another Terragrunt module, so you can pass them as input variables to the current module, as follows:

```
mysql_config = dependency.mysql.outputs
```

In other words, `dependency` blocks are an alternative to using `terraform_remote_state` data sources to pass data between modules. While `terraform_remote_state` data sources have the advantage of being native to Terraform, the drawback is that they make your modules more tightly coupled together, as each module needs to know how other modules store state. Using Terragrunt `dependency` blocks allows your modules to expose generic inputs like `mysql_config` and `vpc_id`, instead of using data sources, which makes the modules less tightly coupled and easier to test and reuse.

Once you've got `hello-world-app` working in staging, create analogous *terragrunt.hcl* files in *live/prod* and promote the exact same `v0.0.7` artifact to production by running `terragrunt apply` in each module.

Putting It All Together

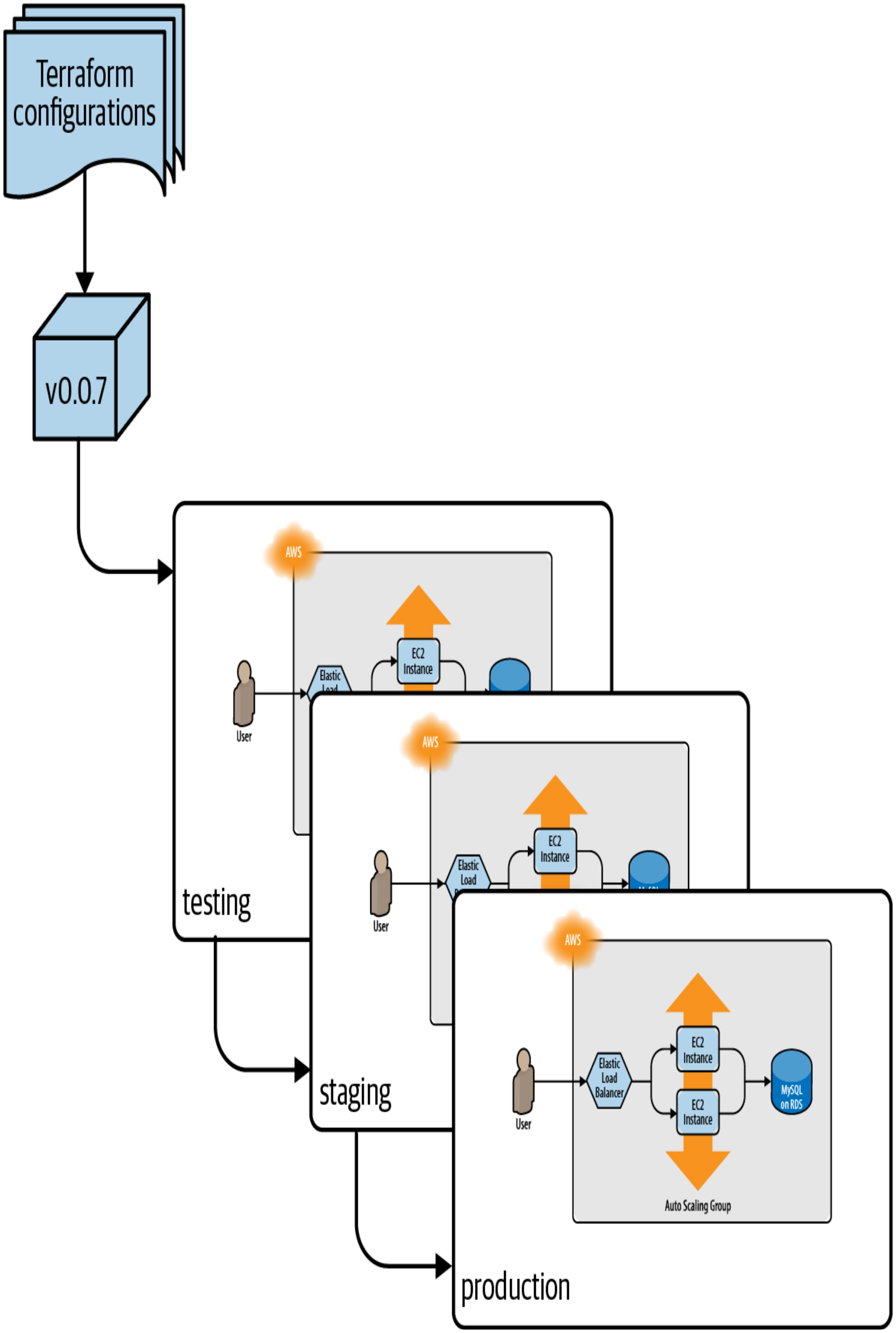
You've now seen how to take both application code and infrastructure code from development all the way through to production. Table 10-1 shows an overview of the two workflows side by side.

Table 10-1. Application and infrastructure code workflows

	Application code	Infrastructure code
Use version control	• `git clone` • One repo per app • Use branches	• `git clone` • *live* and *modules* repos • Don't use branches
Run the code locally	• Run on localhost • `ruby web-server.rb` • `ruby web-server-test.rb`	• Run in a sandbox environment • `terraform apply` • `go test`
Make code changes	• Change the code • `ruby web-server.rb` • `ruby web-server-test.rb`	• Change the code • `terraform apply` • `go test` • Use test stages
Submit changes for review	• Submit a pull request • Enforce coding guidelines	• Submit a pull request • Enforce coding guidelines
Run automated tests	• Tests run on CI server • Unit tests • Integration tests • End-to-end tests • Static analysis	• Tests run on CI server • Unit tests • Integration tests • End-to-end tests • Static analysis • `terraform plan`

	Application code	Infrastructure code
Merge and release	• `git tag` • Create versioned, immutable artifact	• `git tag` • Use repo with tag as versioned, immutable artifact
Deploy	• Deploy with Terraform, orchestration tool (e.g., Kubernetes, Mesos), scripts • Many deployment strategies: rolling deployment, blue-green, canary • Run deployment on a CI server • Give CI server limited permissions • Promote immutable, versioned artifacts across environments • Once a pull request is merged, deploy automatically	• Deploy with Terraform, Atlantis, Terraform Cloud, Terraform Enterprise, Terragrunt, scripts • Limited deployment strategies (make sure to handle errors: retries, `errored.tfstate`!) • Run deployment on a CI server • Give CI server temporary credentials solely to invoke a separate, locked-down worker that has admin permissions • Promote immutable, versioned artifacts across environments • Once a pull request is merged, go through an approval workflow where someone checks the `plan` output one last time, and then deploy automatically

If you follow this process, you will be able to run application and infrastructure code in dev, test it, review it, package it into versioned, immutable artifacts, and promote those artifacts from environment to environment, as shown in Figure 10-6.


Terraform configurations
v0.0.7
testing
staging
production
AWS
User
Elastic Load Balancer
EC2 Instance
EC2 Instance
MySQL on RDS
Auto Scaling Group