Configure your CI/CD pipeline to require that every deployment be approved by at least one person (other than the person who requested the deployment in the first place). During this approval step, the reviewer should be able to see both the code changes and the `plan` output, as one final check that things look OK before `apply` runs. This ensures that every deployment, code change, and `plan` output has had at least two sets of eyes on it.

Don't give the CI server permanent credentials

As you saw in Chapter 6, instead of manually managed, permanent credentials (e.g., AWS access keys copy/pasted into your CI server), you should prefer to use authentication mechanisms that use temporary credentials, such as IAM roles and OIDC.

Don't give the CI server admin credentials

Instead, isolate the admin credentials to a totally separate, isolated *worker*: e.g., a separate server, a separate container, etc. That worker should be extremely locked down, so no developers have access to it at all, and the only thing it allows is for the CI server to trigger that worker via an extremely limited remote API. For example, that worker's API may only allow you to run specific commands (e.g., `terraform plan` and `terraform apply`), in specific repos (e.g., your live repo), in specific branches (e.g., the `main` branch), and so on. This way, even if an attacker gets access to your CI server, they still won't have access to the admin credentials, and all they can do is request a deployment on some code that's already in your version control system, which isn't nearly as much of a catastrophe as leaking the admin credentials fully.[6]

Promote artifacts across environments

Just as with application artifacts, you'll want to promote your immutable, versioned infrastructure artifacts from environment to environment: for

example, promote `v0.0.6` from dev to stage to prod.[7] The rule here is also simple:

Always test Terraform changes in pre-prod before prod.

Because everything is automated with Terraform anyway, it doesn't cost you much extra effort to try a change in staging before production, but it will catch a huge number of errors. Testing in pre-prod is especially important because, as mentioned earlier in this chapter, Terraform does not roll back changes in case of errors. If you run `terraform apply` and something goes wrong, you must fix it yourself. This is easier and less stressful to do if you catch the error in a pre-prod environment rather than prod.

The process for promoting Terraform code across environments is similar to the process of promoting application artifacts, except there is an extra approval step, as mentioned in the previous section, where you run `terraform plan` and have someone manually review the output and approve the deployment. This step isn't usually necessary for application deployments, as most application deployments are similar and relatively low risk. However, every infrastructure deployment can be completely different, and mistakes can be very costly (e.g., deleting a database), so having one last chance to look at the `plan` output and review it is well worth the time.

Here's what the process looks like for promoting, for instance, `v0.0.6` of a Terraform module across the dev, stage, and prod environments:

1. Update the dev environment to `v0.0.6`, and run `terraform plan`.
2. Prompt someone to review and approve the plan; for example, send an automated message via Slack.
3. If the plan is approved, deploy `v0.0.6` to dev by running `terraform apply`.
4. Run your manual and automated tests in dev.

5. If `v0.0.6` works well in dev, repeat steps 1–4 to promote `v0.0.6` to staging.
6. If `v0.0.6` works well in staging, repeat steps 1–4 again to promote `v0.0.6` to production.

One important issue to deal with is all the code duplication between environments in the *live* repo. For example, consider the *live* repo shown in Figure 10-4.

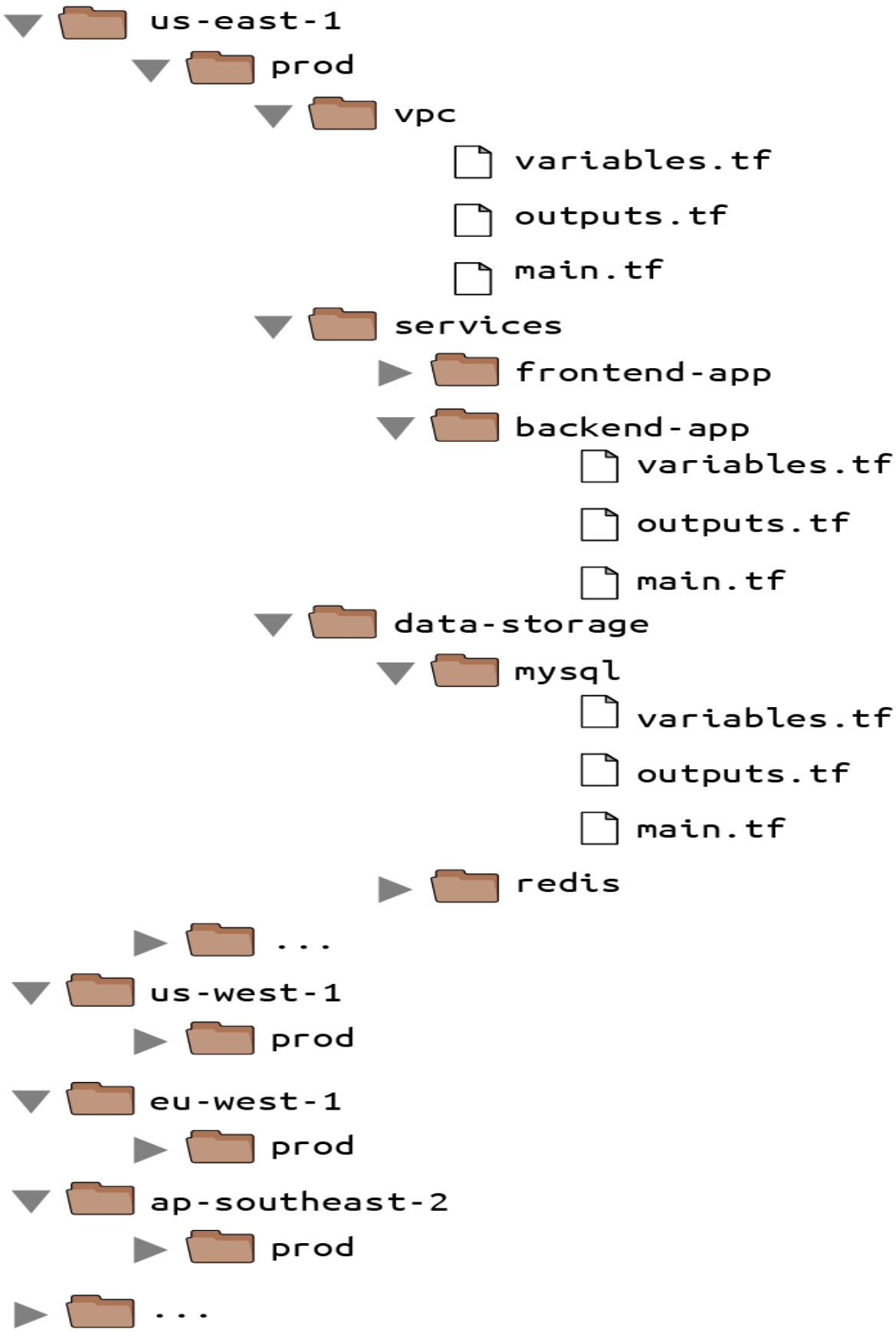
us-east-1
prod
vpc
variables.tf
outputs.tf
main.tf
services
frontend-app
backend-app
variables.tf
outputs.tf
main.tf
data-storage
mysql
variables.tf
outputs.tf
main.tf
redis
...
us-west-1
prod
eu-west-1
prod
ap-southeast-2
prod
...

Figure 10-4. File layout with a large number of copy/pasted environments and modules within each environment.

This *live* repo has a large number of regions, and within each region, a large number of modules, most of which are copied and pasted. Sure, each module has a *main.tf* that references a module in your *modules* repo, so it's not as much copying and pasting as it could be, but even if all you're doing is instantiating a single module, there is still a large amount of boilerplate that needs to be duplicated between each environment:

- The `provider` configuration
- The `backend` configuration
- The input variables to pass to the module
- The output variables to proxy from the module

This can add up to dozens or hundreds of lines of mostly identical code in each module, copied and pasted into each environment. To make this code more DRY, and to make it easier to promote Terraform code across environments, you can use the open source tool I've mentioned earlier called Terragrunt. Terragrunt is a thin wrapper for Terraform, which means that you run all of the standard `terraform` commands, except you use `terragrunt` as the binary:

```
$ terragrunt plan
$ terragrunt apply
$ terragrunt output
```

Terragrunt will run Terraform with the command you specify, but based on configuration you specify in a *terragrunt.hcl* file, you can get some extra behavior. In particular, Terragrunt allows you to define all of your Terraform code exactly once in the *modules* repo, whereas in the *live* repo, you will have solely *terragrunt.hcl* files that provide a DRY way to configure and deploy each module in each environment. This will result in a *live* repo with far fewer files and lines of code, as shown in Figure 10-5.

To get started, install Terragrunt by following the instructions on the Terragrunt website. Next, add a `provider` configuration to *modules/data-stores/mysql/main.tf* and *modules/services/hello-world-app/main.tf*:

```
provider "aws" {
  region = "us-east-2"
}
```

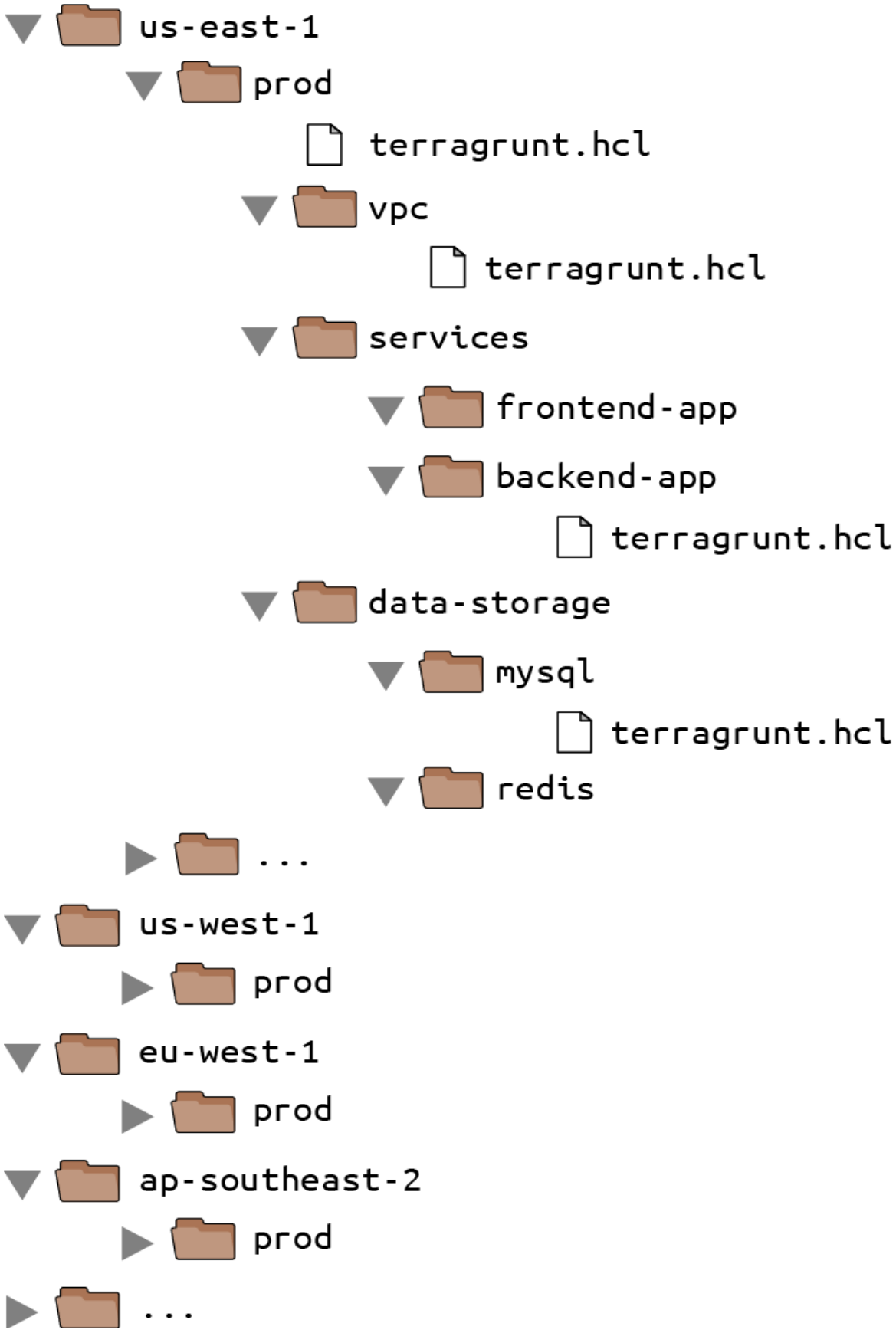
us-east-1
prod
terragrunt.hcl
vpc
terragrunt.hcl
services
frontend-app
backend-app
terragrunt.hcl
data-storage
mysql
terragrunt.hcl
redis
...
us-west-1
prod
eu-west-1
prod
ap-southeast-2
prod
...

Figure 10-5. Use Terragrunt in your live repos to reduce the amount of code duplication.

Commit these changes and release a new version of your *modules* repo:

```
$ git add modules/data-stores/mysql/main.tf
$ git add modules/services/hello-world-app/main.tf
$ git commit -m "Update mysql and hello-world-app for Terragrunt"
$ git tag -a "v0.0.7" -m "Update Hello, World text"
$ git push --follow-tags
```

Now, head over to the *live* repo, and delete all the *.tf* files. You're going to replace all that copied and pasted Terraform code with a single *terragrunt.hcl* file for each module. For example, here's *terragrunt.hcl* for *live/stage/data-stores/mysql/terragrunt.hcl*:

```
terraform {
  source = "github.com/<OWNER>/modules//data-stores/mysql?
ref=v0.0.7"
}

inputs = {
  db_name = "example_stage"

  # Set the username using the TF_VAR_db_username environment
variable
  # Set the password using the TF_VAR_db_password environment
variable
}
```

As you can see, *terragrunt.hcl* files use the same HashiCorp Configuration Language (HCL) syntax as Terraform itself. When you run `terragrunt apply` and it finds the `source` parameter in a *terragrunt.hcl* file, Terragrunt will do the following:

1. Check out the URL specified in `source` to a temporary folder. This supports the same URL syntax as the `source` parameter of Terraform modules, so you can use local file paths, Git URLs, versioned Git URLs (with a `ref` parameter, as in the preceding example), and so on.

2. Run `terraform apply` in the temporary folder, passing it the input variables that you've specified in the `inputs = { ... }` block.

The benefit of this approach is that the code in the *live* repo is reduced to just a single *terragrunt.hcl* file per module, which contains only a pointer to the module to use (at a specific version), plus the input variables to set for that specific environment. That's about as DRY as you can get.

Terragrunt also helps you keep your `backend` configuration DRY. Instead of having to define the `bucket`, `key`, `dynamodb_table`, and so on in every single module, you can define it in a single *terragrunt.hcl* file per environment. For example, create the following in *live/stage/terragrunt.hcl*:

```
remote_state {
  backend = "s3"

  generate = {
    path      = "backend.tf"
    if_exists = "overwrite"
  }

  config = {
    bucket         = "<YOUR BUCKET>"
    key            =
"${path_relative_to_include()}/terraform.tfstate"
    region         = "us-east-2"
    encrypt        = true
    dynamodb_table = "<YOUR_TABLE>"
  }
}
```

From this one `remote_state` block, Terragrunt can generate the `backend` configuration dynamically for each of your modules, writing the configuration in `config` to the file specified via the `generate` param. Note that the `key` value in `config` uses a Terragrunt built-in function called `path_relative_to_include()`, which will return the relative path between this root *terragrunt.hcl* file and any child module that includes it. For example, to include this root file in *live/stage/data-stores/mysql/terragrunt.hcl*, add an `include` block:

```
terraform {
  source = "github.com/<OWNER>/modules//data-stores/mysql?
ref=v0.0.7"
}

include {
  path = find_in_parent_folders()
}

inputs = {
  db_name = "example_stage"

  # Set the username using the TF_VAR_db_username environment
variable
  # Set the password using the TF_VAR_db_password environment
variable
}
```

The `include` block finds the root *terragrunt.hcl* using the Terragrunt built-in function `find_in_parent_folders()`, automatically inheriting all the settings from that parent file, including the `remote_state` configuration. The result is that this `mysql` module will use all the same `backend` settings as the root file, and the `key` value will automatically resolve to *data-stores/mysql/terraform.tfstate*. This means that your Terraform state will be stored in the same folder structure as your *live* repo, which will make it easy to know which module produced which state files.

To deploy this module, run `terragrunt apply`:

```
$ terragrunt apply --terragrunt-log-level debug
DEBU[0001] Reading Terragrunt config file at terragrunt.hcl
DEBU[0001] Included config live/stage/terragrunt.hcl
DEBU[0001] Downloading Terraform configurations into .terragrunt-
cache
DEBU[0001] Generated file backend.tf
DEBU[0013] Running command: terraform init

(...)

Initializing the backend...

Successfully configured the backend "s3"! Terraform will
```

```
automatically
use this backend unless the backend configuration changes.

(...)

DEBU[0024] Running command: terraform apply

(...)

Terraform will perform the following actions:

(...)

Plan: 5 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

(...)

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

Normally, Terragrunt only shows the log output from Terraform itself, but as I included `--terragrunt-log-level debug`, the preceding output shows what Terragrunt does under the hood:

1. Read the *terragrunt.hcl* file in the *mysql* folder where you ran `apply`.
2. Pull in all the settings from the included root *terragrunt.hcl* file.
3. Download the Terraform code specified in the `source` URL into the *.terragrunt-cache* scratch folder.
4. Generate a *backend.tf* file with your `backend` configuration.
5. Detect that `init` has not been run and run it automatically (Terragrunt will even create your S3 bucket and DynamoDB table automatically if they don't already exist).
6. Run `apply` to deploy changes.

Not bad for a couple of tiny *terragrunt.hcl* files!

You can now deploy the `hello-world-app` module in staging by adding *live/stage/services/hello-world-app/terragrunt.hcl* and running `terragrunt apply`:

```
terraform {
  source = "github.com/<OWNER>/modules//services/hello-world-app?
ref=v0.0.7"
}

include {
  path = find_in_parent_folders()
}

dependency "mysql" {
  config_path = "../../data-stores/mysql"
}

inputs = {
  environment = "stage"
  ami         = "ami-0fb653ca2d3203ac1"

  min_size = 2
  max_size = 2

  enable_autoscaling = false

  mysql_config = dependency.mysql.outputs
}
```

This *terragrunt.hcl* file uses the `source` URL and `inputs` just as you saw before and uses `include` to pull in the settings from the root *terragrunt.hcl* file, so it will inherit the same `backend` settings, except for the `key`, which will be automatically set to *services/hello-world-app/terraform.tfstate*, just as you'd expect. The one new thing in this *terragrunt.hcl* file is the `dependency` block:

```
dependency "mysql" {
  config_path = "../../data-stores/mysql"
}
```