

Solution: If you use Terragrunt, you can run commands across multiple folders concurrently using the `run-all` command.

Copy/paste

The file layout described in this section has a lot of duplication. For example, the same `frontend-app` and `backend-app` live in both the `stage` and `prod` folders.

Solution: You won't actually need to copy and paste all of that code! In [Chapter 4](#), you'll see how to use Terraform modules to keep all of this code DRY.

Resource dependencies

Breaking the code into multiple folders makes it more difficult to use resource dependencies. If your app code was defined in the same Terraform configuration files as the database code, that app code could directly access attributes of the database using an attribute reference (e.g., access the database address via `aws_db_instance.foo.address`). But if the app code and database code live in different folders, as I've recommended, you can no longer do that.

Solution: One option is to use dependency blocks in Terragrunt, as you'll see in [Chapter 10](#). Another option is to use the `terraform_remote_state` data source, as described in the next section.

The `terraform_remote_state` Data Source

In [Chapter 2](#), you used data sources to fetch read-only information from AWS, such as the `aws_subnets` data source, which returns a list of subnets in your VPC. There is another data source that is particularly useful when working with state: `terraform_remote_state`. You can use

this data source to fetch the Terraform state file stored by another set of Terraform configurations.

Let's go through an example. Imagine that your web server cluster needs to communicate with a MySQL database. Running a database that is scalable, secure, durable, and highly available is a lot of work. Again, you can let AWS take care of it for you, this time by using Amazon's *Relational Database Service* (RDS), as shown in [Figure 3-9](#). RDS supports a variety of databases, including MySQL, PostgreSQL, SQL Server, and Oracle.

You might not want to define the MySQL database in the same set of configuration files as the web server cluster, because you'll be deploying updates to the web server cluster far more frequently and don't want to risk accidentally breaking the database each time you do so.

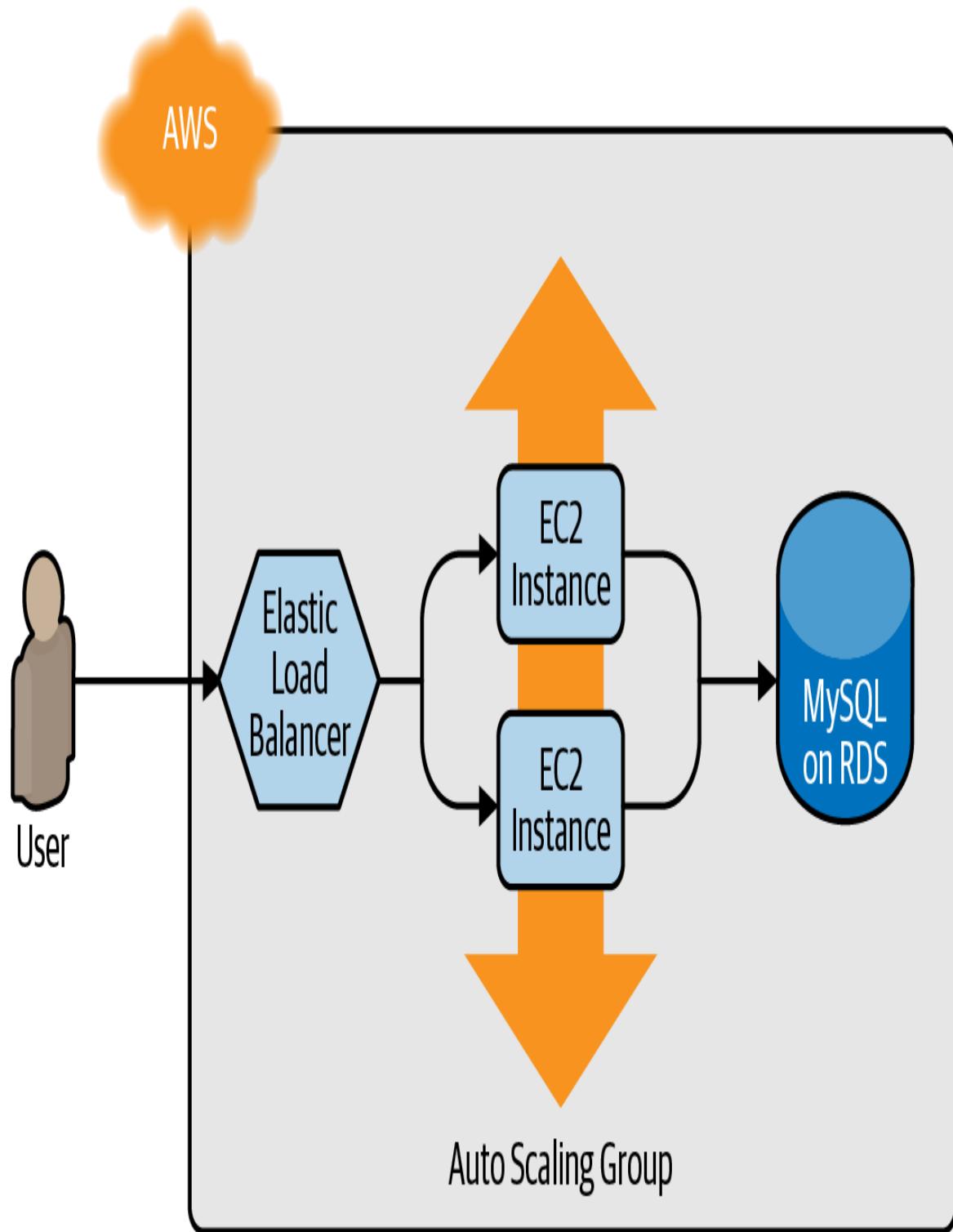


Figure 3-9. The web server cluster communicates with MySQL, which is deployed on top of Amazon RDS.

Therefore, your first step should be to create a new folder at `stage/data-stores/mysql` and create the basic Terraform files (`main.tf`, `variables.tf`,

outputs.tf) within it, as shown in [Figure 3-10](#).

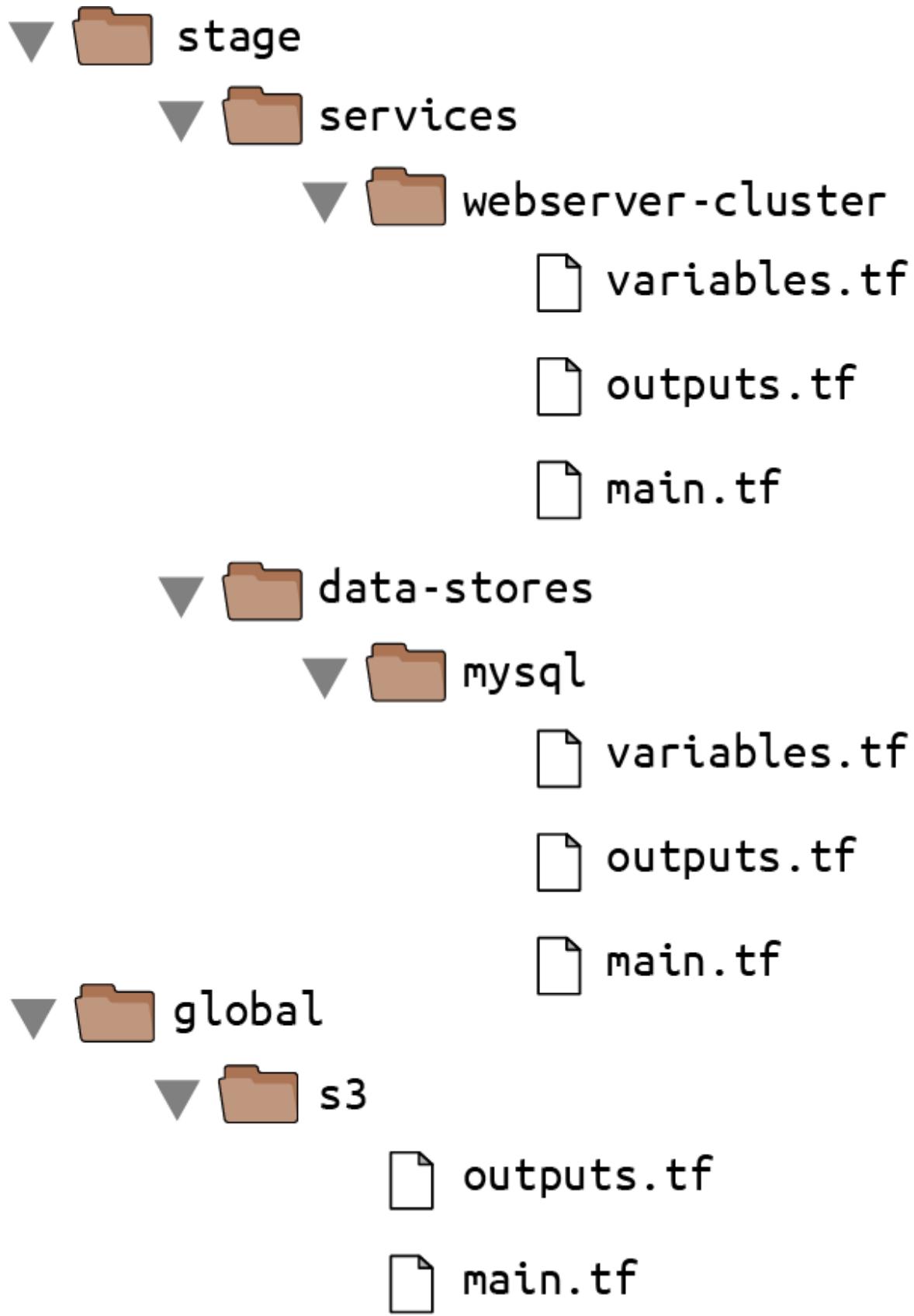


Figure 3-10. Create the database code in the stage/data-stores folder.

Next, create the database resources in *stage/data-stores/mysql/main.tf*:

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

resource "aws_db_instance" "example" {
  identifier_prefix      = "terraform-up-and-running"
  engine                 = "mysql"
  allocated_storage       = 10
  instance_class          = "db.t2.micro"
  skip_final_snapshot     = true
  db_name                = "example_database"

  # How should we set the username and password?
  username = "????"
  password = "????"
}
```

At the top of the file, you see the typical `provider` block, but just below that is a new resource: `aws_db_instance`. This resource creates a database in RDS with the following settings:

- MySQL as the database engine.
- 10 GB of storage.
- A `db.t2.micro` Instance, which has one virtual CPU, 1 GB of memory, and is part of the AWS Free Tier.
- The final snapshot is disabled, as this code is just for learning and testing (if you don't disable the snapshot, or don't provide a name for the snapshot via the `final_snapshot_identifier` parameter, `destroy` will fail).

Note that two of the parameters that you must pass to the `aws_db_instance` resource are the master username and master password. Because these are secrets, you should not put them directly into your code in plain text! In [Chapter 6](#), I'll discuss a variety of options for

how to securely handle secrets with Terraform. For now, let's use an option that avoids storing any secrets in plain text and is easy to use: you store your secrets, such as database passwords, outside of Terraform (e.g., in a password manager such as 1Password, LastPass, or macOS Keychain), and you pass those secrets into Terraform via environment variables.

To do that, declare variables called `db_username` and `db_password` in `stage/data-stores/mysql/variables.tf`:

```
variable "db_username" {
  description = "The username for the database"
  type        = string
  sensitive   = true
}

variable "db_password" {
  description = "The password for the database"
  type        = string
  sensitive   = true
}
```

First, note that these variables are marked with `sensitive = true` to indicate they contain secrets. This ensures Terraform won't log the values when you run `plan` or `apply`. Second, note that these variables do not have a `default`. This is intentional. You should not store your database credentials or any sensitive information in plain text. Instead, you'll set these variables using environment variables.

Before doing that, let's finish the code. First, pass the two new input variables through to the `aws_db_instance` resource:

```
resource "aws_db_instance" "example" {
  identifier_prefix      = "terraform-up-and-running"
  engine                 = "mysql"
  allocated_storage       = 10
  instance_class          = "db.t2.micro"
  skip_final_snapshot     = true
  db_name                = "example_database"

  username = var.db_username
```

```
    password = var.db_password
}
```

Next, configure this module to store its state in the S3 bucket you created earlier at the path *stage/data-stores/mysql/terraform.tfstate*:

```
terraform {
  backend "s3" {
    # Replace this with your bucket name!
    bucket      = "terraform-up-and-running-state"
    key         = "stage/data-stores/mysql/terraform.tfstate"
    region      = "us-east-2"

    # Replace this with your DynamoDB table name!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt       = true
  }
}
```

Finally, add two output variables in *stage/data-stores/mysql/outputs.tf* to return the database's address and port:

```
output "address" {
  value      = aws_db_instance.example.address
  description = "Connect to the database at this endpoint"
}

output "port" {
  value      = aws_db_instance.example.port
  description = "The port the database is listening on"
}
```

You're now ready to pass in the database username and password using environment variables. As a reminder, for each input variable `foo` defined in your Terraform configurations, you can provide Terraform the value of this variable using the environment variable `TF_VAR_foo`. For the `db_username` and `db_password` input variables, here is how you can set the `TF_VAR_db_username` and `TF_VAR_db_password` environment variables on Linux/Unix/macOS systems:

```
$ export TF_VAR_db_username="(YOUR_DB_USERNAME)"
$ export TF_VAR_db_password="(YOUR_DB_PASSWORD)"
```

And here is how you do it on Windows systems:

```
$ set TF_VAR_db_username="(YOUR_DB_USERNAME)"
$ set TF_VAR_db_password="(YOUR_DB_PASSWORD)"
```

Run `terraform init` and `terraform apply` to create the database. Note that Amazon RDS can take roughly 10 minutes to provision even a small database, so be patient. After `apply` completes, you should see the outputs in the terminal:

```
$ terraform apply
(...)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Outputs:

```
address = "terraform-up-and-running.cowu6mts6srx.us-east-
2.rds.amazonaws.com"
port = 3306
```

These outputs are now also stored in the Terraform state for the database, which is in your S3 bucket at the path `stage/data-stores/mysql/terraform.tfstate`.

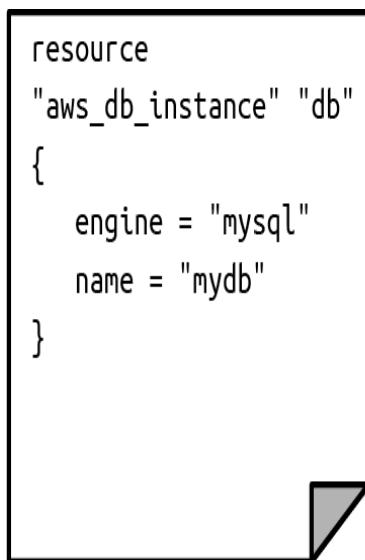
If you go back to your web server cluster code, you can get the web server to read those outputs from the database's state file by adding the `terraform_remote_state` data source in `stage/services/webserver-cluster/main.tf`:

```
data "terraform_remote_state" "db" {
  backend = "s3"

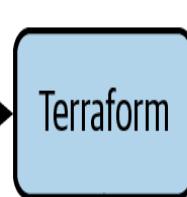
  config = {
    bucket = "(YOUR_BUCKET_NAME)"
    key    = "stage/data-stores/mysql/terraform.tfstate"
    region = "us-east-2"
```

```
    }  
}
```

This `terraform_remote_state` data source configures the web server cluster code to read the state file from the same S3 bucket and folder where the database stores its state, as shown in [Figure 3-11](#).

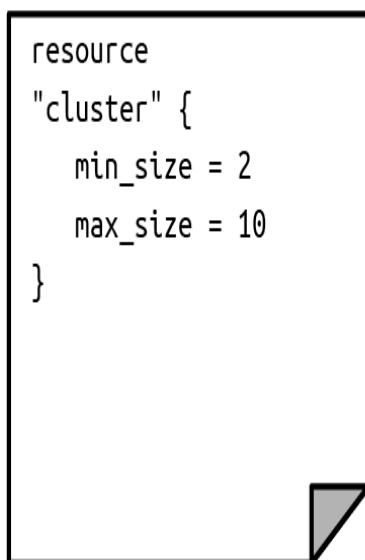
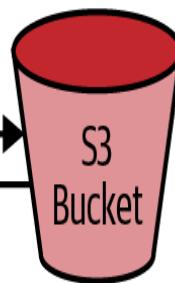


Database
Terraform configurations



Write state

Read state



Web server
Terraform configurations

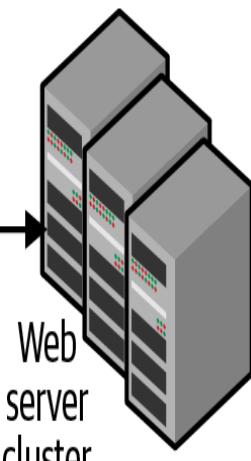
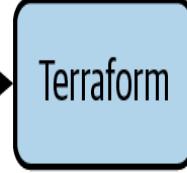


Figure 3-11. The database writes its state to an S3 bucket (top), and the web server cluster reads that state from the same bucket (bottom).

It's important to understand that, like all Terraform data sources, the data returned by `terraform_remote_state` is read-only. Nothing you do in your web server cluster Terraform code can modify that state, so you can pull in the database's state data with no risk of causing any problems in the database itself.

All of the database's output variables are stored in the state file, and you can read them from the `terraform_remote_state` data source using an attribute reference of the form:

```
data.terraform_remote_state.<NAME>.outputs.<ATTRIBUTE>
```

For example, here is how you can update the User Data of the web server cluster Instances to pull the database address and port out of the `terraform_remote_state` data source and expose that information in the HTTP response:

```
user_data = <<EOF
#!/bin/bash
echo "Hello, World" >> index.xhtml
echo "${data.terraform_remote_state.db.outputs.address}" >>
index.xhtml
echo "${data.terraform_remote_state.db.outputs.port}" >>
index.xhtml
nohup busybox httpd -f -p ${var.server_port} &
EOF
```

As the User Data script is growing longer, defining it inline is becoming messier and messier. In general, embedding one programming language (Bash) inside another (Terraform) makes it more difficult to maintain each one, so let's pause here for a moment to externalize the Bash script. To do that, you can use the `templatefile` built-in function.

Terraform includes a number of *built-in functions* that you can execute using an expression of the form:

```
function_name(...)
```

For example, consider the `format` function:

```
format(<FMT>, <ARGS>, ...)
```

This function formats the arguments in `ARGS` according to the `sprintf` syntax in the string `FMT`.⁶ A great way to experiment with built-in functions is to run the `terraform console` command to get an interactive console where you can try out Terraform syntax, query the state of your infrastructure, and see the results instantly:

```
$ terraform console  
> format("%.3f", 3.14159265359)  
3.142
```

Note that the Terraform console is read-only, so you don't need to worry about accidentally changing infrastructure or state.

There are a number of other built-in functions that you can use to manipulate strings, numbers, lists, and maps.⁷ One of them is the `templatefile` function:

```
templatefile(<PATH>, <VARS>)
```

This function reads the file at `PATH`, renders it as a template, and returns the result as a string. When I say “renders it as a template,” what I mean is that the file at `PATH` can use the string interpolation syntax in Terraform (`$(...)`), and Terraform will render the contents of that file, filling variable references from `VARS`.

To see this in action, put the contents of the User Data script into the file `stage/services/webserver-cluster/user-data.sh` as follows:

```
#!/bin/bash  
  
cat > index.xhtml <<EOF
```

```

<h1>Hello, World</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF

nohup busybox httpd -f -p ${server_port} &

```

Note that this Bash script has a few changes from the original:

- It looks up variables using Terraform's standard interpolation syntax, except the only variables it has access to are those you pass in via the second parameter to `templatefile` (as you'll see shortly), so you don't need any prefix to access them: for example, you should use `${server_port}` and not `${var.server_port}`.
- The script now includes some HTML syntax (e.g., `<h1>`) to make the output a bit more readable in a web browser.

The final step is to update the `user_data` parameter of the `aws_launch_configuration` resource to call the `templatefile` function and pass in the variables it needs as a map:

```

resource "aws_launch_configuration" "example" {
  image_id      = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
  security_groups = [aws_security_group.instance.id]

  # Render the User Data script as a template
  user_data = templatefile("user-data.sh", {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  })

  # Required when using a launch configuration with an auto
  # scaling group.
  lifecycle {
    create_before_destroy = true
  }
}

```

Ah, that's much cleaner than writing Bash scripts inline!