- How many copies (aka *replicas*) of the Pod to run in your cluster. The preceding code configures three replicas. Kubernetes automatically figures out where in your cluster to deploy each Pod, using a scheduling algorithm to pick the optimal servers in terms of high availability (e.g., try to run each Pod on a separate server so a single server crash doesn't take down your app), resources (e.g., pick servers that have available the ports, CPU, memory, and other resources required by your containers), performance (e.g., try to pick servers with the least load and fewest containers on them), and so on. Kubernetes also constantly monitors the cluster to ensure that there are always three replicas running, automatically replacing any Pods that crash or stop responding.
- How to deploy updates. When deploying a new version of the Docker container, the preceding code rolls out three new replicas, waits for them to be healthy, and then undeploys the three old replicas.

That's a lot of power in just a few lines of YAML! You run `kubectl apply -f example-app.yml` to instruct Kubernetes to deploy your app. You can then make changes to the YAML file and run `kubectl apply` again to roll out the updates. You can also manage both the Kubernetes cluster and the apps within it using Terraform; you'll see an example of this in Chapter 7.

Provisioning Tools

Whereas configuration management, server templating, and orchestration tools define the code that runs on each server, *provisioning tools* such as Terraform, CloudFormation, OpenStack Heat, and Pulumi are responsible for creating the servers themselves. In fact, you can use provisioning tools to create not only servers but also databases, caches, load balancers, queues, monitoring, subnet configurations, firewall settings, routing rules, Secure Sockets Layer (SSL) certificates, and almost every other aspect of your infrastructure, as shown in Figure 1-5.

For example, the following code deploys a web server using Terraform:

```
resource "aws_instance" "app" {
  instance_type     = "t2.micro"
  availability_zone = "us-east-2a"
  ami               = "ami-0fb653ca2d3203ac1"

  user_data = <<-EOF
              #!/bin/bash
              sudo service apache2 start
              EOF
}
```

Don't worry if you're not yet familiar with some of the syntax. For now, just focus on two parameters:

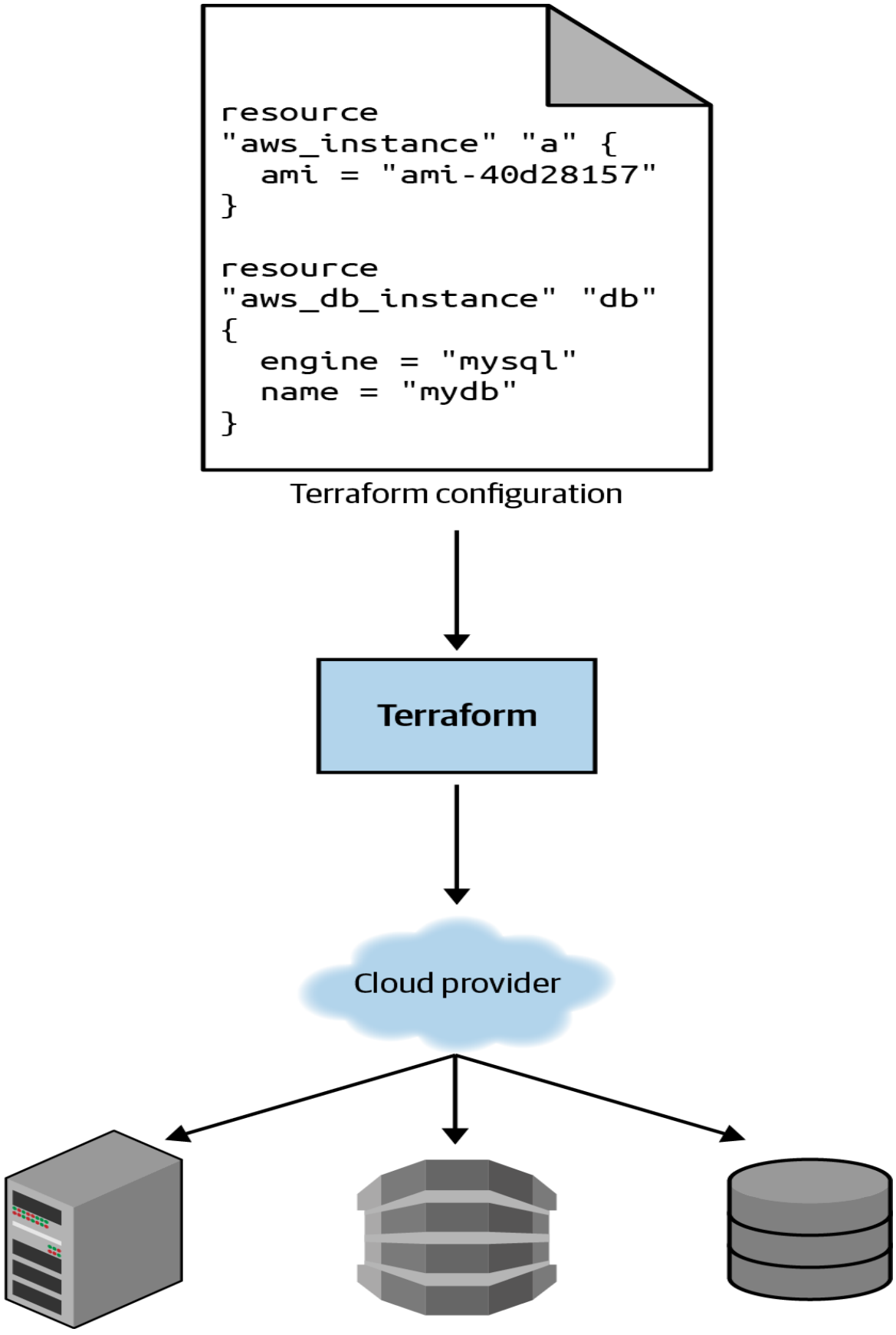
`ami`

: This parameter specifies the ID of an AMI to deploy on the server. You could set this parameter to the ID of an AMI built from the *web-server.json* Packer template in the previous section, which has PHP, Apache, and the application source code.

`user_data`

: This is a Bash script that executes when the web server is booting. The preceding code uses this script to boot up Apache.

In other words, this code shows you provisioning and server templating working together, which is a common pattern in immutable infrastructure.


resource
"aws_instance" "a" {
ami = "ami-40d28157"
}
resource
"aws_db_instance" "db"
{
engine = "mysql"
name = "mydb"
}
Terraform configuration
Terraform
Cloud provider