> There is one other very important, and often overlooked, reason for why you should use IaC: happiness. Deploying code and managing infrastructure manually is repetitive and tedious. Developers and sysadmins resent this type of work, since it involves no creativity, no challenge, and no recognition. You could deploy code perfectly for months, and no one will take notice—until that one day when you mess it up. That creates a stressful and unpleasant environment. IaC offers a better alternative that allows computers to do what they do best (automation) and developers to do what they do best (coding).

Now that you have a sense of why IaC is important, the next question is whether Terraform is the best IaC tool for you. To answer that, I'm first going to go through a very quick primer on how Terraform works, and then I'll compare it to the other popular IaC options out there, such as Chef, Puppet, and Ansible.

How Does Terraform Work?

Here is a high-level and somewhat simplified view of how Terraform works. Terraform is an open source tool created by HashiCorp and written in the Go programming language. The Go code compiles down into a single binary (or rather, one binary for each of the supported operating systems) called, not surprisingly, `terraform`.

You can use this binary to deploy infrastructure from your laptop or a build server or just about any other computer, and you don't need to run any extra infrastructure to make that happen. That's because under the hood, the `terraform` binary makes API calls on your behalf to one or more *providers*, such as AWS, Azure, Google Cloud, DigitalOcean, OpenStack, and more. This means that Terraform gets to leverage the infrastructure those providers are already running for their API servers, as well as the authentication mechanisms you're already using with those providers (e.g., the API keys you already have for AWS).

How does Terraform know what API calls to make? The answer is that you create *Terraform configurations*, which are text files that specify what infrastructure you want to create. These configurations are the "code" in "infrastructure as code." Here's an example Terraform configuration:

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}

resource "google_dns_record_set" "a" {
  name         = "demo.google-example.com"
  managed_zone = "example-zone"
  type         = "A"
  ttl          = 300
  rrdatas      = [aws_instance.example.public_ip]
}
```

Even if you've never seen Terraform code before, you shouldn't have too much trouble reading it. This snippet instructs Terraform to make API calls to AWS to deploy a server, and then make API calls to Google Cloud to create a Domain Name System (DNS) entry pointing to the AWS server's IP address. In just a single, simple syntax (which you'll learn in Chapter 2), Terraform allows you to deploy interconnected resources across multiple cloud providers.

You can define your entire infrastructure—servers, databases, load balancers, network topology, and so on—in Terraform configuration files and commit those files to version control. You then run certain Terraform commands, such as `terraform apply`, to deploy that infrastructure. The `terraform` binary parses your code, translates it into a series of API calls to the cloud providers specified in the code, and makes those API calls as efficiently as possible on your behalf, as shown in Figure 1-6.

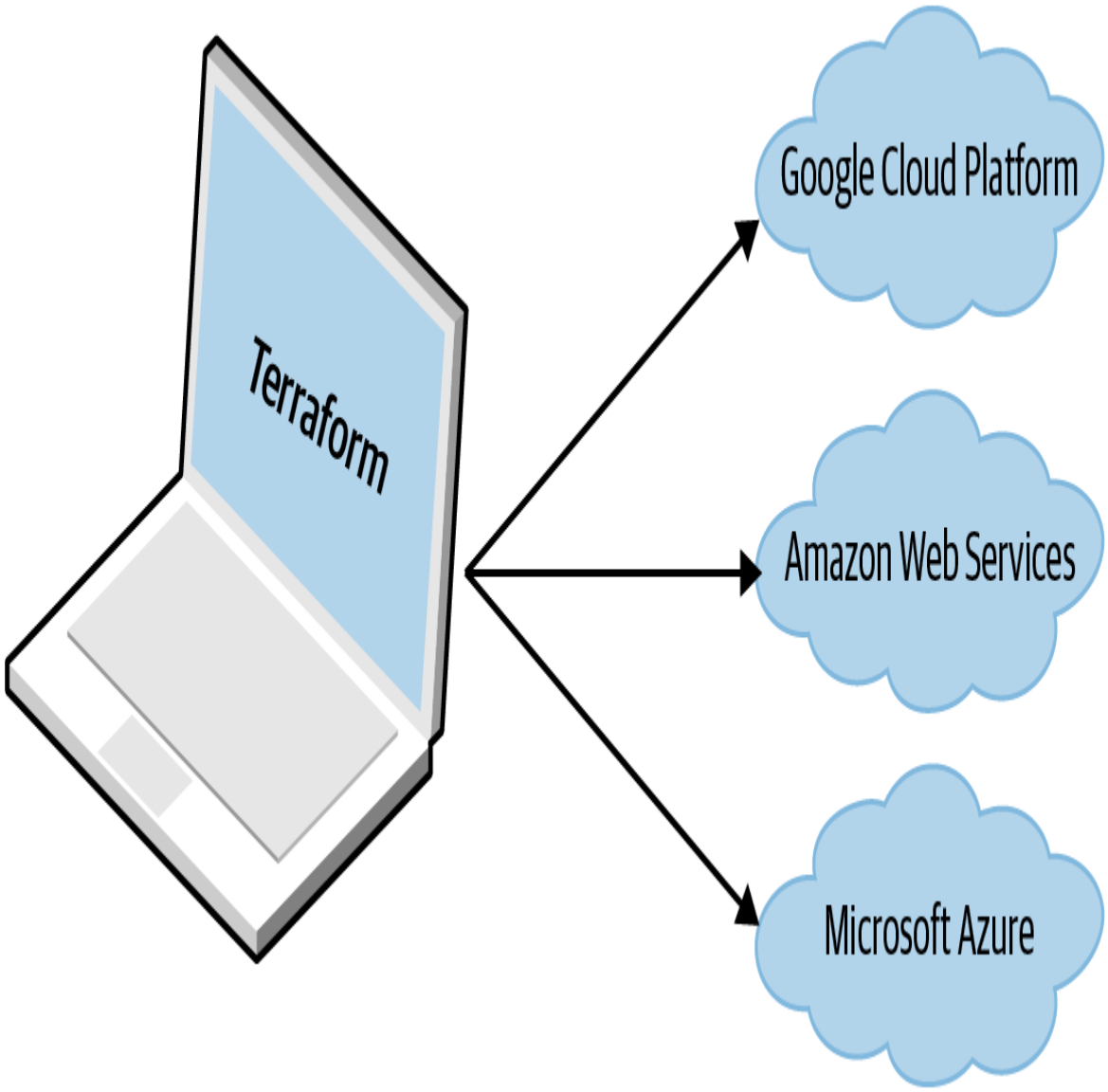

Figure 1-6. Terraform is a binary that translates the contents of your configurations into API calls to cloud providers.

When someone on your team needs to make changes to the infrastructure, instead of updating the infrastructure manually and directly on the servers, they make their changes in the Terraform configuration files, validate those changes through automated tests and code reviews, commit the updated code to version control, and then run the `terraform apply` command to have Terraform make the necessary API calls to deploy the changes.