If you're using a provisioning tool such as Terraform to deploy machine images created by Docker or Packer, most "changes" are actually deployments of a completely new server. For example, to deploy a new version of OpenSSL, you would use Packer to create a new image with the new version of OpenSSL, deploy that image across a set of new servers, and then terminate the old servers. Because every deployment uses immutable images on fresh servers, this approach reduces the likelihood of configuration drift bugs, makes it easier to know exactly what software is running on each server, and allows you to easily deploy any previous version of the software (any previous image) at any time. It also makes your automated testing more effective, because an immutable image that passes your tests in the test environment is likely to behave exactly the same way in the production environment.

Of course, it's possible to force configuration management tools to do immutable deployments, too, but it's not the idiomatic approach for those tools, whereas it's a natural way to use provisioning tools. It's also worth mentioning that the immutable approach has downsides of its own. For example, rebuilding an image from a server template and redeploying all your servers for a trivial change can take a long time. Moreover, immutability lasts only until you actually run the image. After a server is up and running, it will begin making changes on the hard drive and experiencing some degree of configuration drift (although this is mitigated if you deploy frequently).

Procedural Language Versus Declarative Language

Chef and Ansible encourage a *procedural* style in which you write code that specifies, step by step, how to achieve some desired end state.

Terraform, CloudFormation, Puppet, OpenStack Heat, and Pulumi all encourage a more *declarative* style in which you write code that specifies your desired end state, and the IaC tool itself is responsible for figuring out how to achieve that state.

To demonstrate the difference, let's go through an example. Imagine that you want to deploy 10 servers (*EC2 Instances* in AWS lingo) to run an AMI with ID `ami-0fb653ca2d3203ac1` (Ubuntu 20.04). Here is a simplified example of an Ansible template that does this using a procedural approach:

```
- ec2:
    count: 10
    image: ami-0fb653ca2d3203ac1
    instance_type: t2.micro
```

And here is a simplified example of a Terraform configuration that does the same thing using a declarative approach:

```
resource "aws_instance" "example" {
  count         = 10
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

On the surface, these two approaches might look similar, and when you initially execute them with Ansible or Terraform, they will produce similar results. The interesting thing is what happens when you want to make a change.

For example, imagine traffic has gone up, and you want to increase the number of servers to 15. With Ansible, the procedural code you wrote earlier is no longer useful; if you just updated the number of servers to 15 and reran that code, it would deploy 15 new servers, giving you 25 total! So instead, you need to be aware of what is already deployed and write a totally new procedural script to add the five new servers:

```
- ec2:
    count: 5
    image: ami-0fb653ca2d3203ac1
    instance_type: t2.micro
```

With declarative code, because all you do is declare the end state that you want and Terraform figures out how to get to that end state, Terraform will also be aware of any state it created in the past. Therefore, to deploy five more servers, all you need to do is go back to the same Terraform configuration and update the count from 10 to 15:

```
resource "aws_instance" "example" {
  count         = 15
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

If you applied this configuration, Terraform would realize it had already created 10 servers and therefore all it needs to do is create five new servers. In fact, before applying this configuration, you can use Terraform's `plan` command to preview what changes it would make:

```
$ terraform plan

# aws_instance.example[11] will be created
+ resource "aws_instance" "example" {
    + ami            = "ami-0fb653ca2d3203ac1"
    + instance_type  = "t2.micro"
    + (...)
  }

# aws_instance.example[12] will be created
+ resource "aws_instance" "example" {
    + ami            = "ami-0fb653ca2d3203ac1"
    + instance_type  = "t2.micro"
    + (...)
  }

# aws_instance.example[13] will be created
+ resource "aws_instance" "example" {
    + ami            = "ami-0fb653ca2d3203ac1"
    + instance_type  = "t2.micro"
    + (...)
  }

# aws_instance.example[14] will be created
+ resource "aws_instance" "example" {
    + ami            = "ami-0fb653ca2d3203ac1"
```

```
      + instance_type  = "t2.micro"
      + (...)
    }

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

Now what happens when you want to deploy a different version of the app, such as AMI ID `ami-02bcbb802e03574ba`? With the procedural approach, both of your previous Ansible templates are again not useful, so you need to write yet another template to track down the 10 servers you deployed previously (or was it 15 now?) and carefully update each one to the new version. With the declarative approach of Terraform, you go back to the exact same configuration file again and simply change the `ami` parameter to `ami-02bcbb802e03574ba`:

```
resource "aws_instance" "example" {
  count         = 15
  ami           = "ami-02bcbb802e03574ba"
  instance_type = "t2.micro"
}
```

Obviously, these examples are simplified. Ansible does allow you to use tags to search for existing EC2 Instances before deploying new ones (e.g., using the `instance_tags` and `count_tag` parameters), but having to manually figure out this sort of logic for every single resource you manage with Ansible, based on each resource's past history, can be surprisingly complicated: for example, you may have to manually configure your code to look up existing Instances not only by tag but also by image version, Availability Zone, and other parameters. This highlights two major problems with procedural IaC tools:

Procedural code does not fully capture the state of the infrastructure

> Reading through the three preceding Ansible templates is not enough to know what's deployed. You'd also need to know the *order* in which those templates were applied. Had you applied them in a different order, you might have ended up with different infrastructure, and that's not something you can see in the codebase itself. In other words, to reason