You can also use the `terraform output` command to list all outputs without applying any changes:

```
$ terraform output
public_ip = "54.174.13.5"
```

And you can run `terraform output <OUTPUT_NAME>` to see the value of a specific output called `<OUTPUT_NAME>`:

```
$ terraform output public_ip
"54.174.13.5"
```

This is particularly handy for scripting. For example, you could create a deployment script that runs `terraform apply` to deploy the web server, uses `terraform output public_ip` to grab its public IP, and runs `curl` on the IP as a quick smoke test to validate that the deployment worked.

Input and output variables are also essential ingredients in creating configurable and reusable infrastructure code, a topic you'll see more of in Chapter 4.

## Deploying a Cluster of Web Servers

Running a single server is a good start, but in the real world, a single server is a single point of failure. If that server crashes, or if it becomes overloaded from too much traffic, users will be unable to access your site. The solution is to run a cluster of servers, routing around servers that go down and adjusting the size of the cluster up or down based on traffic.[14]

Managing such a cluster manually is a lot of work. Fortunately, you can let AWS take care of it for you by using an Auto Scaling Group (ASG), as shown in Figure 2-9. An ASG takes care of a lot of tasks for you completely automatically, including launching a cluster of EC2 Instances, monitoring the health of each Instance, replacing failed Instances, and adjusting the size of the cluster in response to load.

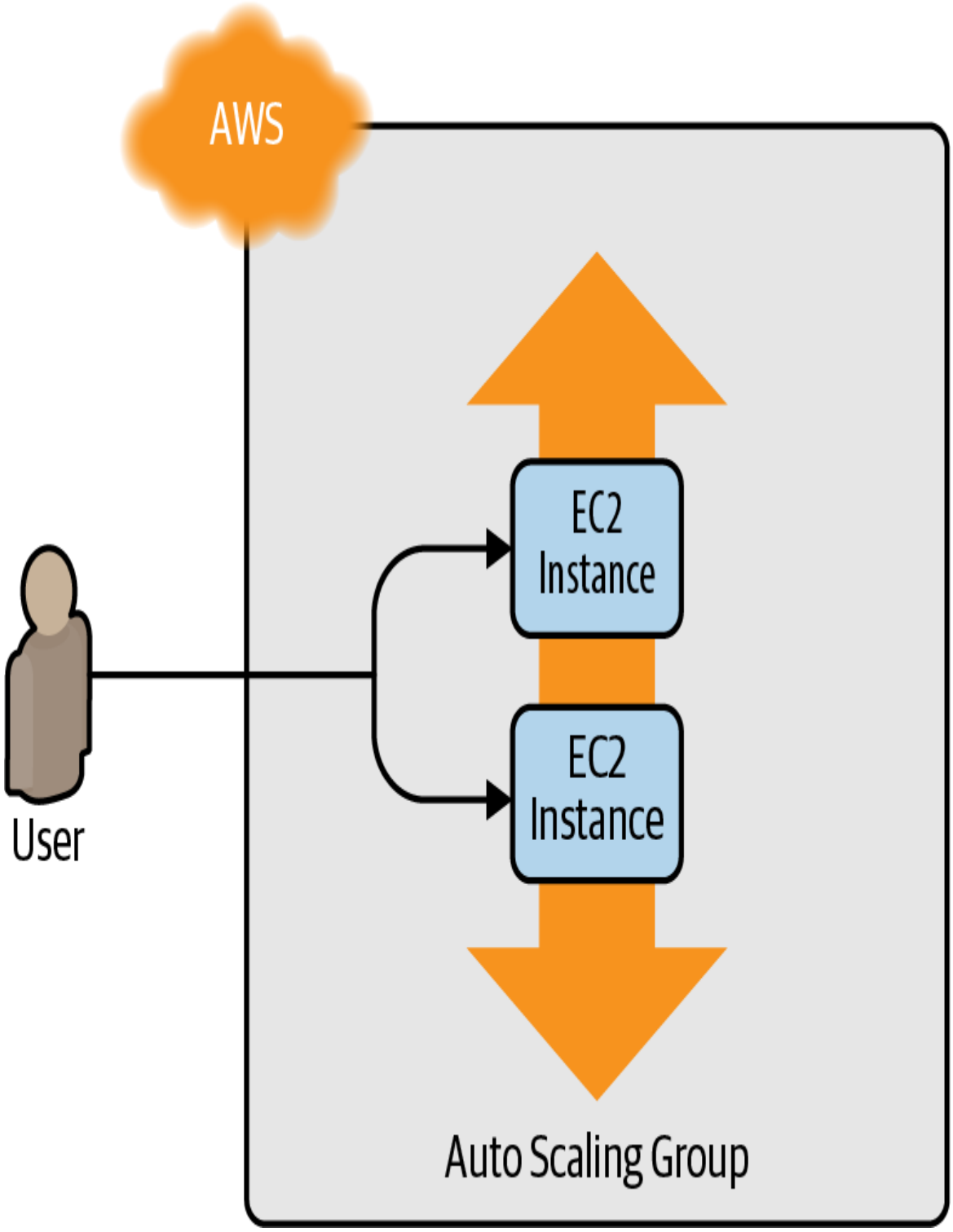


*Figure 2-9. Instead of a single web server, run a cluster of web servers using an Auto Scaling Group.*

The first step in creating an ASG is to create a *launch configuration*, which specifies how to configure each EC2 Instance in the ASG.[15] The

`aws_launch_configuration` resource uses almost the same parameters as the `aws_instance resource`, although it doesn't support tags (you'll handle these in the `aws_autoscaling_group` resource later) or the `user_data_replace_on_change` parameter (ASGs launch new instances by default, so you don't need this parameter), and two of the parameters have different names (`ami` is now `image_id`, and `vpc_security_group_ids` is now `security_groups`), so replace `aws_instance` with `aws_launch_configuration` as follows:

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

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

Now you can create the ASG itself using the `aws_autoscaling_group` resource:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name

  min_size = 2
  max_size = 10

  tag {
    key                 = "Name"
    value               = "terraform-asg-example"
    propagate_at_launch = true
  }
}
```

This ASG will run between 2 and 10 EC2 Instances (defaulting to 2 for the initial launch), each tagged with the name `terraform-asg-example`.

Note that the ASG uses a reference to fill in the launch configuration name. This leads to a problem: launch configurations are immutable, so if you change any parameter of your launch configuration, Terraform will try to replace it. Normally, when replacing a resource, Terraform would delete the old resource first and then creates its replacement, but because your ASG now has a reference to the old resource, Terraform won't be able to delete it.

To solve this problem, you can use a *lifecycle* setting. Every Terraform resource supports several lifecycle settings that configure how that resource is created, updated, and/or deleted. A particularly useful lifecycle setting is `create_before_destroy`. If you set `create_before_destroy` to `true`, Terraform will invert the order in which it replaces resources, creating the replacement resource first (including updating any references that were pointing at the old resource to point to the replacement) and then deleting the old resource. Add the `lifecycle` block to your `aws_launch_configuration` as follows:

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

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

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

There's also one other parameter that you need to add to your ASG to make it work: `subnet_ids`. This parameter specifies to the ASG into which VPC subnets the EC2 Instances should be deployed (see "Network

Security" for background info on subnets). Each subnet lives in an isolated AWS AZ (that is, isolated datacenter), so by deploying your Instances across multiple subnets, you ensure that your service can keep running even if some of the datacenters have an outage. You could hardcode the list of subnets, but that won't be maintainable or portable, so a better option is to use *data sources* to get the list of subnets in your AWS account.

A data source represents a piece of read-only information that is fetched from the provider (in this case, AWS) every time you run Terraform. Adding a data source to your Terraform configurations does not create anything new; it's just a way to query the provider's APIs for data and to make that data available to the rest of your Terraform code. Each Terraform provider exposes a variety of data sources. For example, the AWS Provider includes data sources to look up VPC data, subnet data, AMI IDs, IP address ranges, the current user's identity, and much more.

The syntax for using a data source is very similar to the syntax of a resource:

```
data "<PROVIDER>_<TYPE>" "<NAME>" {
  [CONFIG ...]
}
```

Here, `PROVIDER` is the name of a provider (e.g., `aws`), `TYPE` is the type of data source you want to use (e.g., `vpc`), `NAME` is an identifier you can use throughout the Terraform code to refer to this data source, and `CONFIG` consists of one or more arguments that are specific to that data source. For example, here is how you can use the `aws_vpc` data source to look up the data for your Default VPC (see "A Note on Default Virtual Private Clouds" for background information):

```
data "aws_vpc" "default" {
  default = true
}
```

Note that with data sources, the arguments you pass in are typically search filters that indicate to the data source what information you're looking for.

With the `aws_vpc` data source, the only filter you need is `default = true`, which directs Terraform to look up the Default VPC in your AWS account.

To get the data out of a data source, you use the following attribute reference syntax:

```
data.<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>
```

For example, to get the ID of the VPC from the `aws_vpc` data source, you would use the following:

```
data.aws_vpc.default.id
```

You can combine this with another data source, `aws_subnets`, to look up the subnets

within that VPC:

```
data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}
```

Finally, you can pull the subnet IDs out of the `aws_subnets` data source and tell your ASG to use those subnets via the (somewhat oddly named) `vpc_zone_identifier` argument:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnets.default.ids

  min_size = 2
  max_size = 10

  tag {
    key                 = "Name"
    value               = "terraform-asg-example"
```