OK, that's a nice improvement: no extra whitespace or commas. You can make this output even prettier by adding an `else` to the string directive, which uses the following syntax:

```
%{ if <CONDITION> }<TRUEVAL>%{ else }<FALSEVAL>%{ endif }
```

where `FALSEVAL` is the expression to render if `CONDITION` evaluates to false. Here's an example of how to use the `else` clause to add a period at the end:

```
output "for_directive_index_if_else_strip" {
  value = <<EOF
%{~ for i, name in var.names ~}
${name}%{ if i < length(var.names) - 1 }, %{ else }.%{ endif }
%{~ endfor ~}
EOF
}
```

When you run `terraform apply`, you get the following output:

```
$ terraform apply

(...)

Outputs:

for_directive_index_if_else_strip = "neo, trinity, morpheus."
```

Zero-Downtime Deployment

Now that your module has a clean and simple API for deploying a web server cluster, an important question to ask is, how do you update that cluster? That is, when you make changes to your code, how do you deploy a new Amazon Machine Image (AMI) across the cluster? And how do you do it without causing downtime for your users?

The first step is to expose the AMI as an input variable in *modules/services/webserver-cluster/variables.tf*. In real-world examples, this is all you would need because the actual web server code would be defined in the AMI. However, in the simplified examples in this book, all of the web server code is actually in the User Data script, and the AMI is just a vanilla Ubuntu image. Switching to a different version of Ubuntu won't make for much of a demonstration, so in addition to the new AMI input variable, you can also add an input variable to control the text the User Data script returns from its one-liner HTTP server:

```
variable "ami" {
  description = "The AMI to run in the cluster"
  type        = string
  default     = "ami-0fb653ca2d3203ac1"
}

variable "server_text" {
  description = "The text the web server should return"
  type        = string
  default     = "Hello, World"
}
```

Now you need to update the *modules/services/webserver-cluster/user-data.sh* Bash script to use this `server_text` variable in the `<h1>` tag it returns:

```
#!/bin/bash

cat > index.xhtml <<EOF
<h1>${server_text}</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF

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

Finally, find the launch configuration in *modules/services/webserver-cluster/main.tf*, update the `image_id` parameter to use `var.ami`, and update the `templatefile` call in the `user_data` parameter to pass in `var.server_text`:

```
resource "aws_launch_configuration" "example" {
  image_id        = var.ami
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]

  user_data       = templatefile("${path.module}/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
    server_text = var.server_text
  })

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

Now, in the staging environment, in *live/stage/services/webserver-cluster/main.tf*, you can set the new `ami` and `server_text` parameters:

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-cluster"

  ami         = "ami-0fb653ca2d3203ac1"
  server_text = "New server text"

  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-
stores/mysql/terraform.tfstate"

  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
}
```

This code uses the same Ubuntu AMI, but changes the `server_text` to a new value. If you run the `plan` command, you should see something like the following:

```
Terraform will perform the following actions:

  # module.webserver_cluster.aws_autoscaling_group.ex will be
updated in-place
  ~ resource "aws_autoscaling_group" "example" {
        id                         = "webservers-stage-terraform-
20190516"
      ~ launch_configuration       = "terraform-20190516" ->
(known after apply)
        (...)
    }

  # module.webserver_cluster.aws_launch_configuration.ex must be
replaced
+/- resource "aws_launch_configuration" "example" {
      ~ id                          = "terraform-20190516" ->
(known after apply)
        image_id                    = "ami-0fb653ca2d3203ac1"
        instance_type               = "t2.micro"
      ~ name                        = "terraform-20190516" ->
(known after apply)
      ~ user_data                   = "bd7c0a6" -> "4919a13" #
forces replacement
        (...)
    }

Plan: 1 to add, 1 to change, 1 to destroy.
```

As you can see, Terraform wants to make two changes: first, replace the old launch configuration with a new one that has the updated `user_data`; and second, modify the Auto Scaling Group in place to reference the new launch configuration. There is a problem here: merely referencing the new launch configuration will have no effect until the ASG launches new EC2 Instances. So how do you instruct the ASG to deploy new Instances?

One option is to destroy the ASG (e.g., by running `terraform destroy`) and then re-create it (e.g., by running `terraform apply`). The problem is that after you delete the old ASG, your users will experience downtime until the new ASG comes up. What you want to do instead is a *zero-downtime deployment*. The way to accomplish that is to create the replacement ASG first and then destroy the original one. As it turns out, the `create_before_destroy` lifecycle setting you first saw in Chapter 2 does exactly this.

Here's how you can take advantage of this lifecycle setting to get a zero-downtime deployment:[1]

1. Configure the `name` parameter of the ASG to depend directly on the name of the launch configuration. Each time the launch configuration changes (which it will when you update the AMI or User Data), its name changes, and therefore the ASG's name will change, which forces Terraform to replace the ASG.
2. Set the `create_before_destroy` parameter of the ASG to `true` so that each time Terraform tries to replace it, it will create the replacement ASG before destroying the original.
3. Set the `min_elb_capacity` parameter of the ASG to the `min_size` of the cluster so that Terraform will wait for at least that many servers from the new ASG to pass health checks in the ALB before it will begin destroying the original ASG.

Here is what the updated `aws_autoscaling_group` resource should look like in *modules/services/webserver-cluster/main.tf*:

```
resource "aws_autoscaling_group" "example" {
  # Explicitly depend on the launch configuration's name so each
time it's
  # replaced, this ASG is also replaced
  name =
"${var.cluster_name}-${aws_launch_configuration.example.name}"

  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnets.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"

  min_size = var.min_size
  max_size = var.max_size

  # Wait for at least this many instances to pass health checks
before
```

```
  # considering the ASG deployment complete
  min_elb_capacity = var.min_size

  # When replacing this ASG, create the replacement first, and
only delete the
  # original after
  lifecycle {
    create_before_destroy = true
  }

  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }

  dynamic "tag" {
    for_each = {
      for key, value in var.custom_tags:
      key => upper(value)
      if key != "Name"
    }

    content {
      key                 = tag.key
      value               = tag.value
      propagate_at_launch = true
    }
  }
}
```

If you rerun the `plan` command, you'll now see something that looks like the following:

```
Terraform will perform the following actions:

  # module.webserver_cluster.aws_autoscaling_group.example must
be replaced
+/- resource "aws_autoscaling_group" "example" {
      ~ id     = "example-2019" -> (known after apply)
      ~ name   = "example-2019" -> (known after apply) # forces
replacement
        (...)
    }

  # module.webserver_cluster.aws_launch_configuration.example
```

```
must be replaced
+/- resource "aws_launch_configuration" "example" {
      ~ id               = "terraform-2019" -> (known after apply)
        image_id         = "ami-0fb653ca2d3203ac1"
        instance_type    = "t2.micro"
      ~ name             = "terraform-2019" -> (known after apply)
      ~ user_data        = "bd7c0a" -> "4919a" # forces
replacement
        (...)
    }

    (...)

Plan: 2 to add, 2 to change, 2 to destroy.
```

The key thing to notice is that the `aws_autoscaling_group` resource now says `forces replacement` next to its name parameter, which means that Terraform will replace it with a new ASG running your new AMI or User Data. Run the `apply` command to kick off the deployment, and while it runs, consider how the process works.

You start with your original ASG running, say, v1 of your code (Figure 5-1).

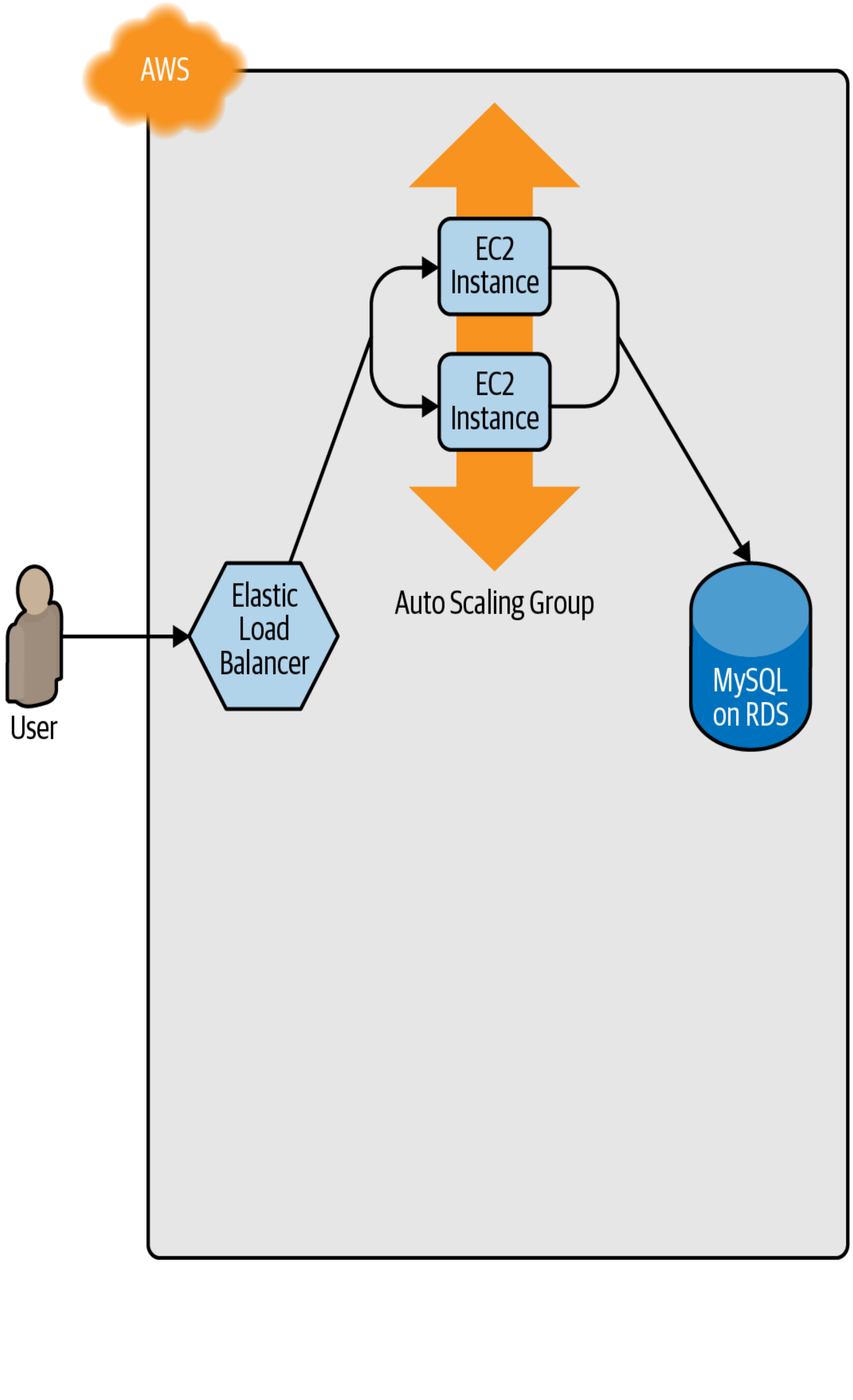

AWS
EC2 Instance
EC2 Instance
Auto Scaling Group
User
Elastic Load Balancer
MySQL on RDS

Figure 5-1. Initially, you have the original ASG running v1 of your code.

You make an update to some aspect of the launch configuration, such as switching to an AMI that contains v2 of your code, and run the `apply` command. This forces Terraform to begin deploying a new ASG with v2 of your code (Figure 5-2).

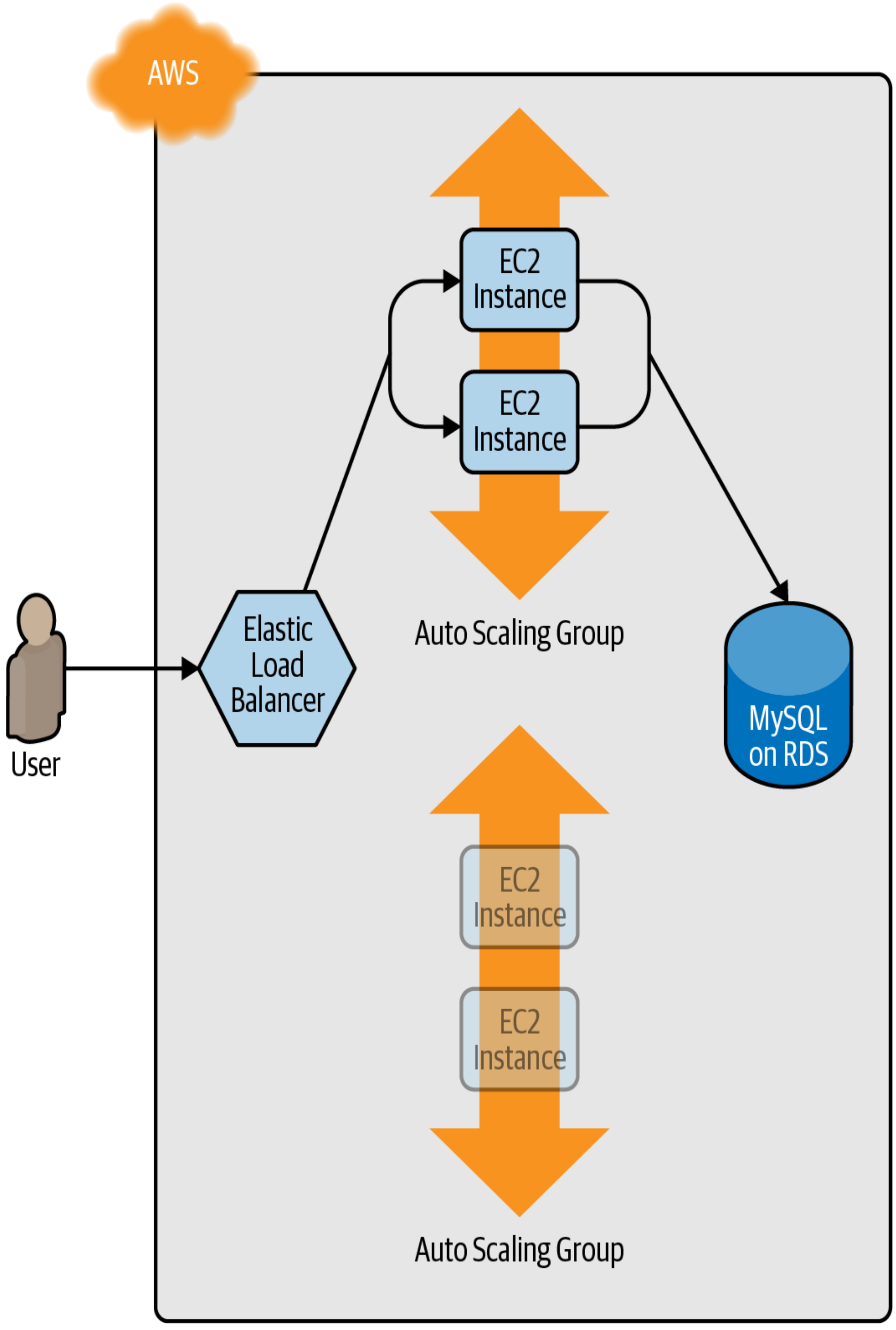

AWS
EC2 Instance
EC2 Instance
User
Elastic Load Balancer
Auto Scaling Group
MySQL on RDS
EC2 Instance
EC2 Instance
Auto Scaling Group

Figure 5-2. Terraform begins deploying the new ASG with v2 of your code.

After a minute or two, the servers in the new ASG have booted, connected to the database, registered in the ALB, and started to pass health checks. At this point, both the v1 and v2 versions of your app will be running simultaneously; and which one users see depends on where the ALB happens to route them (Figure 5-3).

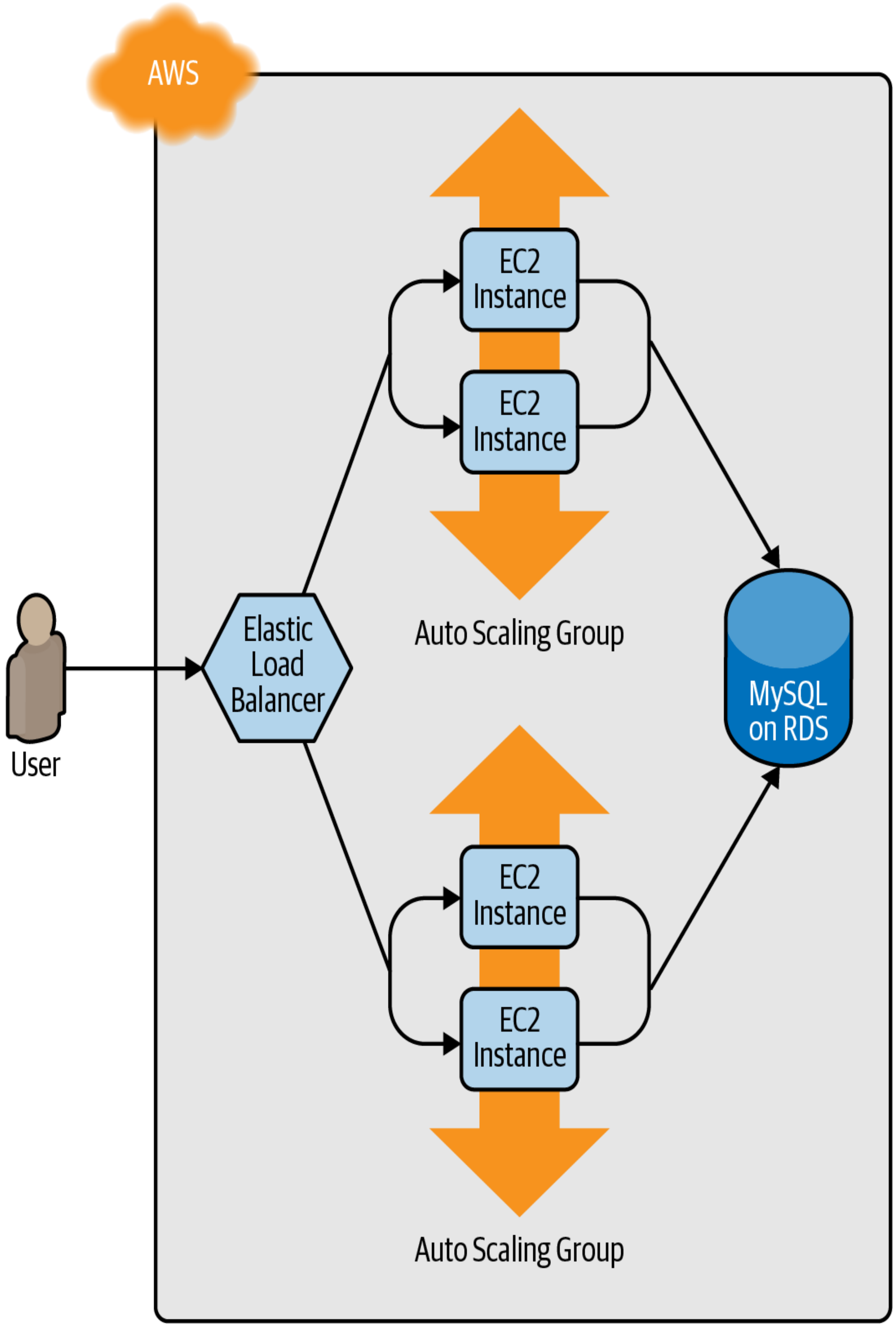

AWS
User
Elastic Load Balancer
EC2 Instance
EC2 Instance
Auto Scaling Group
EC2 Instance
EC2 Instance
Auto Scaling Group
MySQL on RDS

Figure 5-3. The servers in the new ASG boot up, connect to the DB, register in the ALB, and begin serving traffic.

After `min_elb_capacity` servers from the v2 ASG cluster have registered in the ALB, Terraform will begin to undeploy the old ASG, first by deregistering the servers in that ASG from the ALB, and then by shutting them down (Figure 5-4).

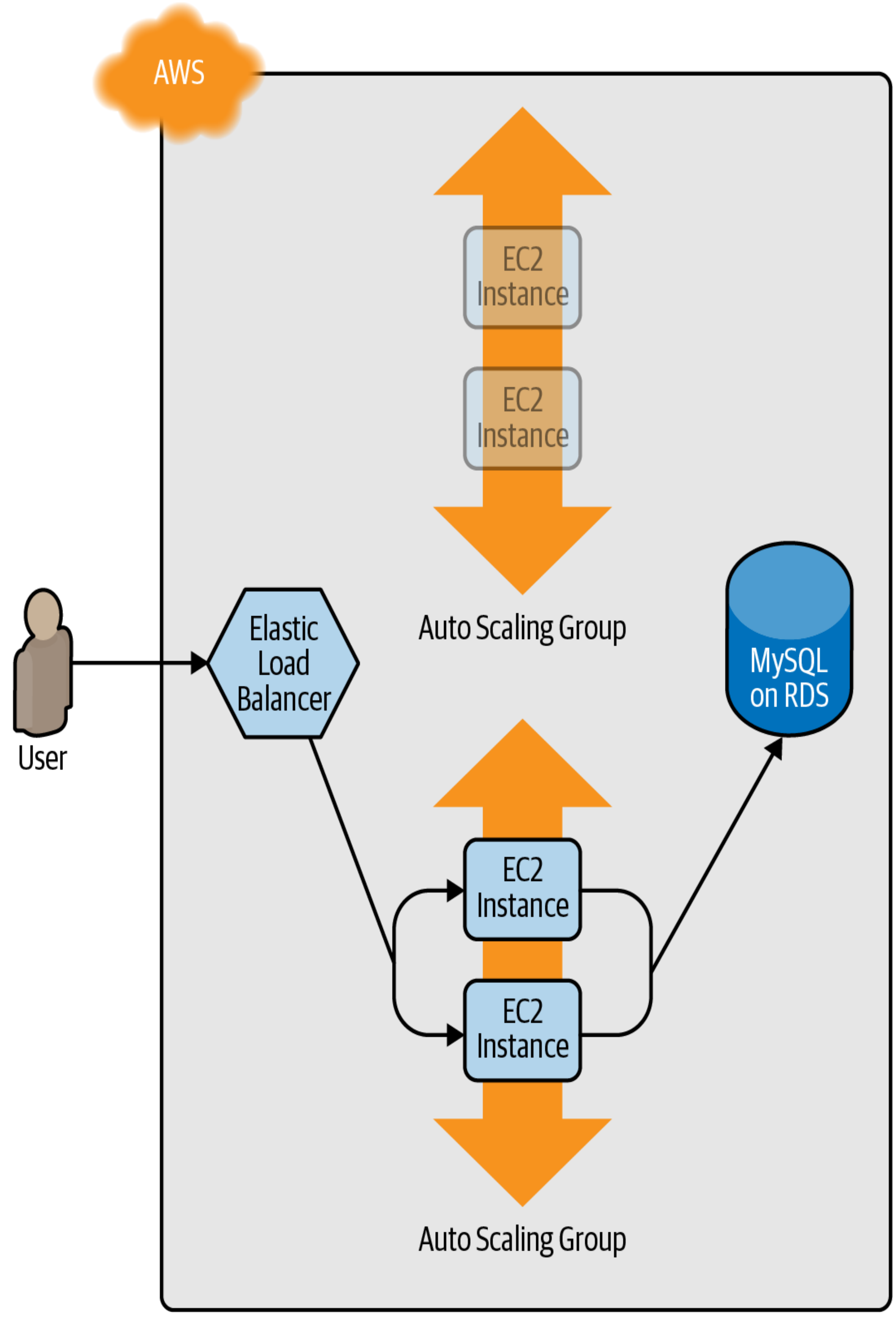

AWS
EC2 Instance
EC2 Instance
Auto Scaling Group
User
Elastic Load Balancer
MySQL on RDS
EC2 Instance
EC2 Instance
Auto Scaling Group

Figure 5-4. The servers in the old ASG begin to shut down.

After a minute or two, the old ASG will be gone, and you will be left with just v2 of your app running in the new ASG (Figure 5-5).

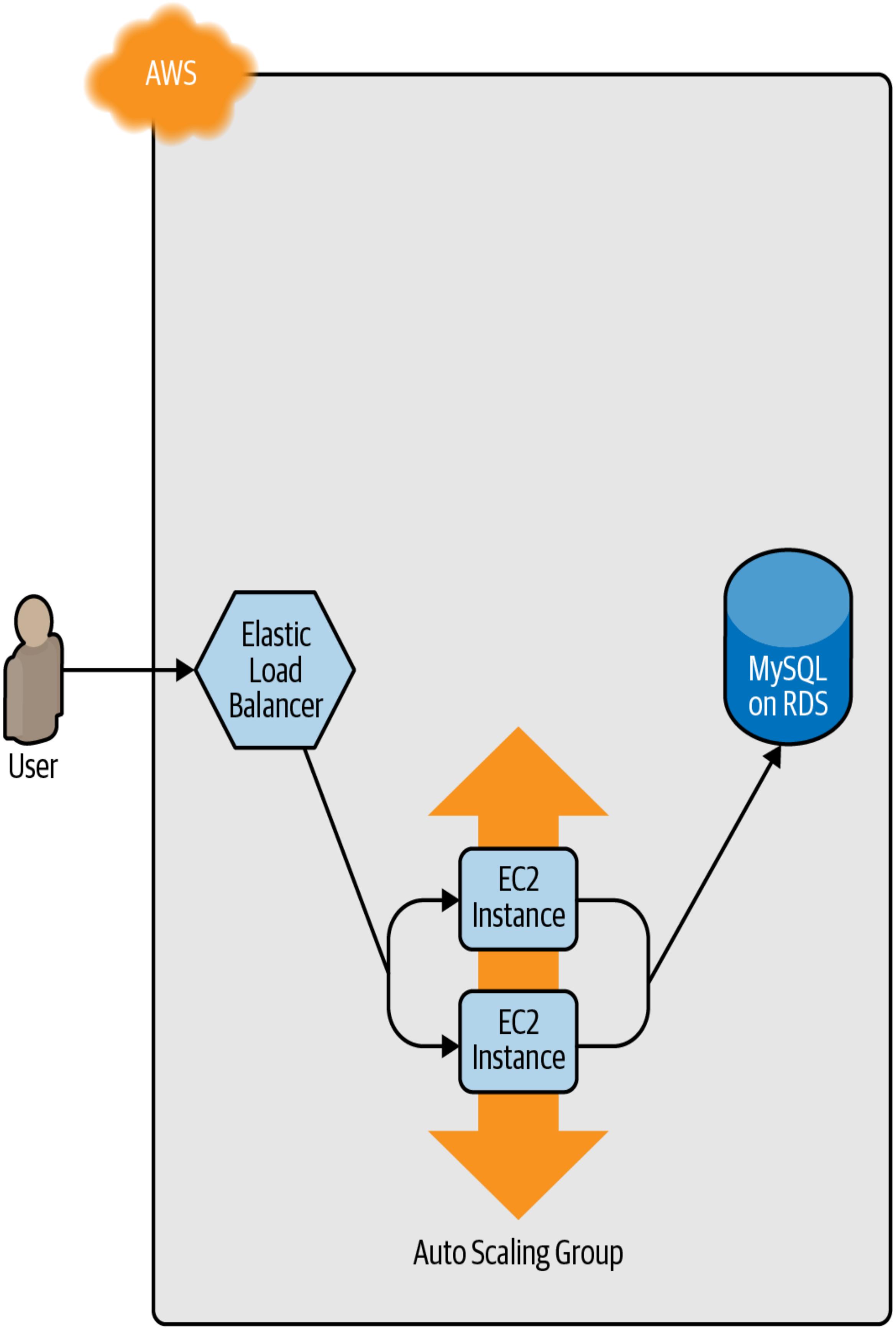
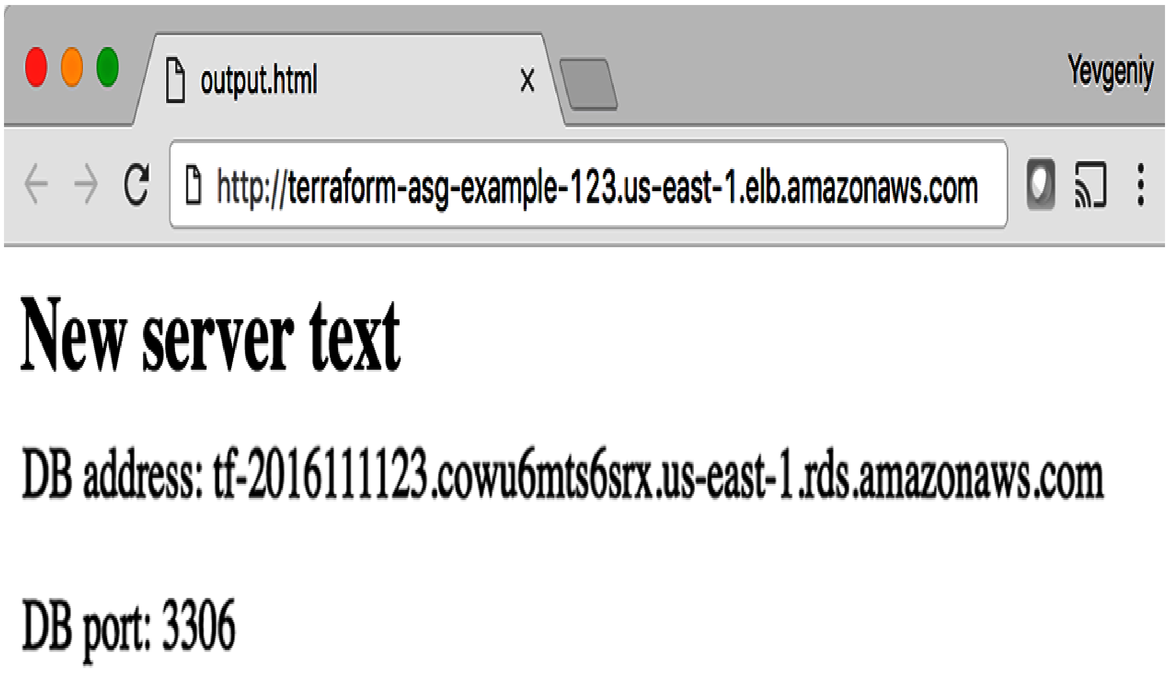

AWS
User
Elastic Load Balancer
MySQL on RDS
EC2 Instance
EC2 Instance
Auto Scaling Group

Figure 5-5. Now, only the new ASG remains, which is running v2 of your code.

During this entire process, there are always servers running and handling requests from the ALB, so there is no downtime. Open the ALB URL in your browser, and you should see something like Figure 5-6.


output.html
Yevgeniy
http://terraform-asg-example-123.us-east-1.elb.amazonaws.com

New server text

DB address: tf-2016111123.cowu6mts6srx.us-east-1.rds.amazonaws.com

DB port: 3306

Figure 5-6. The new code is now deployed.

Success! The new server text has deployed. As a fun experiment, make another change to the `server_text` parameter—for example, update it to say "foo bar"—and run the `apply` command. In a separate terminal tab, if you're on Linux/Unix/macOS, you can use a Bash one-liner to run `curl` in a loop, hitting your ALB once per second and allowing you to see the zero-downtime deployment in action:

```
$ while true; do curl http://<load_balancer_url>; sleep 1; done
```

For the first minute or so, you should see the same response: `New server text`. Then, you'll begin seeing it alternate between `New server text` and `foo bar`. This means the new Instances have registered in the ALB and passed health checks. After another minute, the `New server text` message will disappear, and you'll see only `foo bar`, which means the