```
the
resources that the count depends on.
```

Terraform requires that it can compute `count` and `for_each` during the `plan` phase, *before* any resources are created or modified. This means that `count` and `for_each` can reference hardcoded values, variables, data sources, and even lists of resources (so long as the length of the list can be determined during `plan`), but not computed resource outputs.

Zero-Downtime Deployment Has Limitations

There are a couple of gotchas with using `create_before_destroy` with an ASG to do zero-downtime deployment.

The first issue is that it doesn't work with auto scaling policies. Or, to be more accurate, it resets your ASG size back to its `min_size` after each deployment, which can be a problem if you had used auto scaling policies to increase the number of running servers. For example, the `webserver-cluster` module includes a couple of `aws_autoscaling_schedule` resources that increase the number of servers in the cluster from 2 to 10 at 9 a.m. If you ran a deployment at, say, 11 a.m., the replacement ASG would boot up with only 2 servers, rather than 10, and it would stay that way until 9 a.m. the next day. There are several possible workarounds, such as tweaking the `recurrence` parameter on the `aws_autoscaling_schedule` or setting the `desired_capacity` parameter of the ASG to get its value from a custom script that uses the AWS API to figure out how many instances were running before deployment.

However, the second, and bigger, issue is that, for important and complicated tasks like a zero-downtime deployment, you really want to use native, first-class solutions, and not workarounds that require you to haphazardly glue together `create_before_destroy`, `min_elb_capacity`, custom scripts, etc. As it turns out, for Auto Scaling Groups, AWS now offers a native solution called *instance refresh*.

Go back to your `aws_autoscaling_group` resource and undo the zero-downtime deployment changes:

- Set `name` back to `var.cluster_name`, instead of having it depend on the `aws_launch_configuration` name.
- Remove the `create_before_destroy` and `min_elb_capacity` settings.

And now, update the `aws_autoscaling_group` resource to instead use an `instance_refresh` block as follows:

```
resource "aws_autoscaling_group" "example" {
  name                 = var.cluster_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

  # Use instance refresh to roll out changes to the ASG
  instance_refresh {
    strategy = "Rolling"
    preferences {
      min_healthy_percentage = 50
    }
  }
}
```

If you deploy this ASG, and then later change some parameter (e.g., change `server_text`) and run `plan`, the diff will be back to just updating the `aws_launch_configuration`:

```
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.
```

If you run `apply`, it'll complete very quickly, and at first, nothing new will be deployed. However, in the background, because you modified the launch configuration, AWS will kick off the instance refresh process, as shown in Figure 5-7.

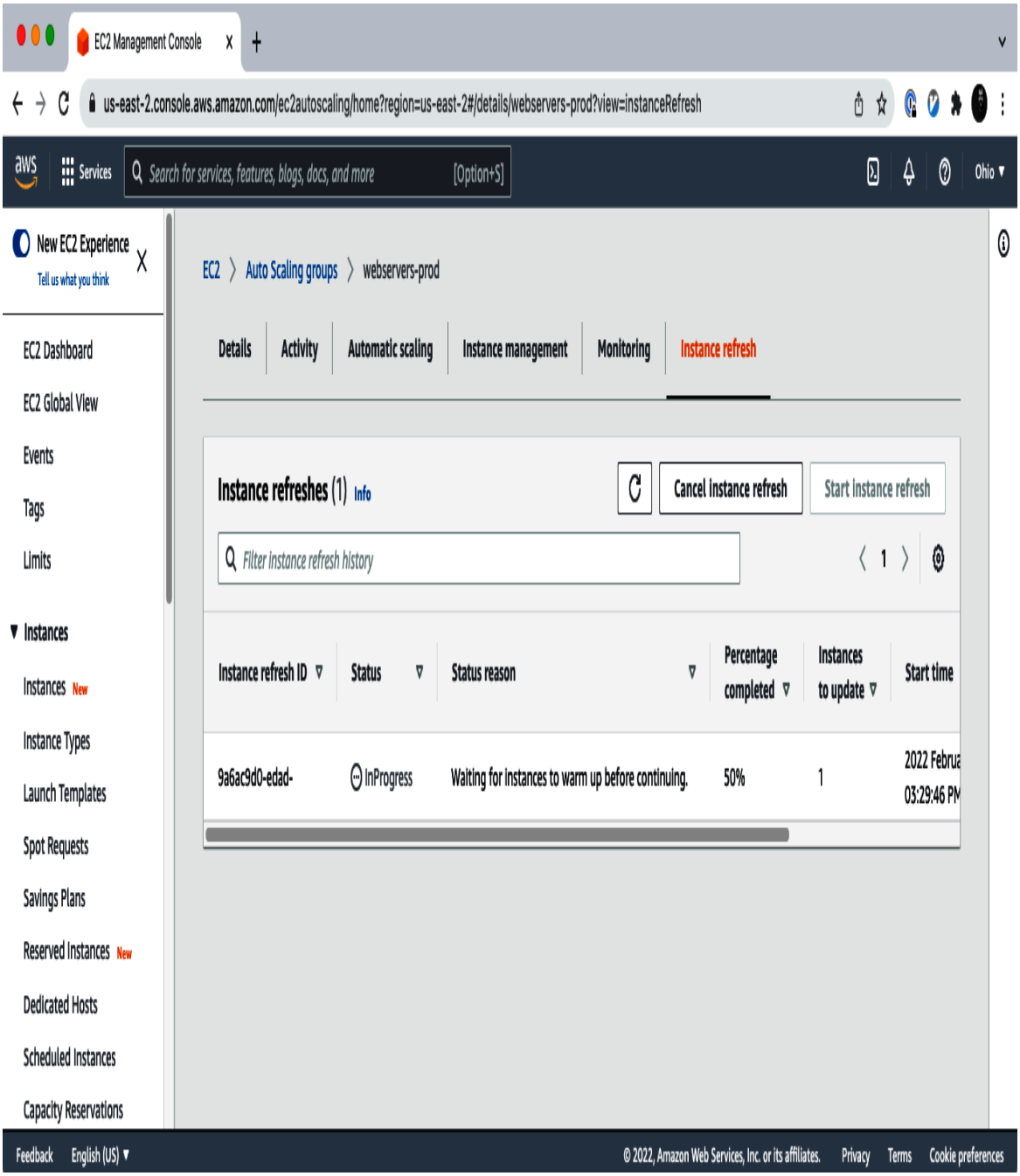

Figure 5-7. An instance refresh is in progress.

AWS will initially launch one new instance, wait for it to pass health checks, shut down one of the older instances, and then repeat the process with the second instance, until the instance refresh is completed, as shown in Figure 5-8.

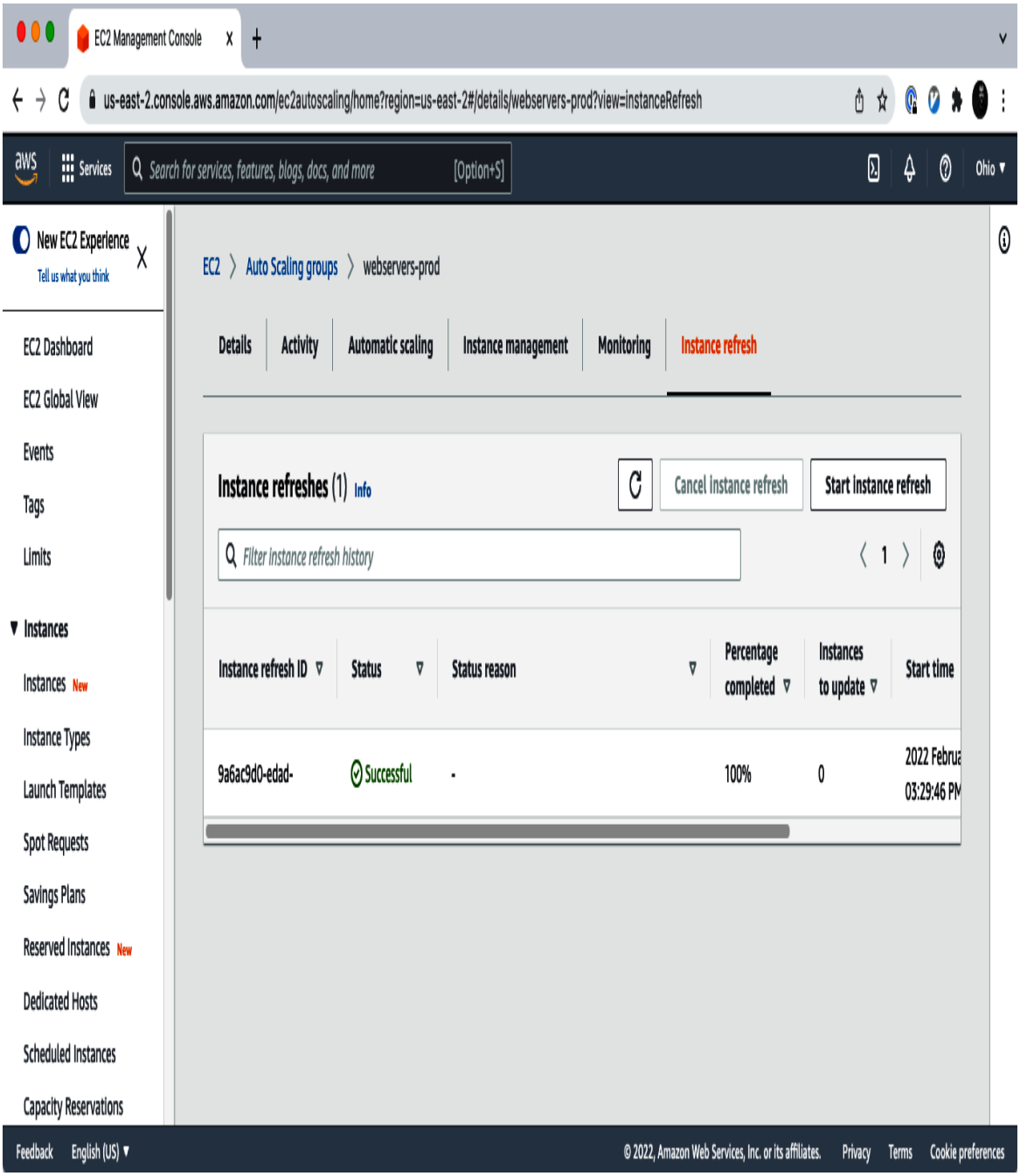

Figure 5-8. An instance refresh is completed.

This process is entirely managed by AWS, is reasonably configurable, handles errors pretty well, and requires no workarounds. The only drawback is the process can sometimes be slow (taking up to 20 minutes to replace just two servers), but other than that, it's a much more robust solution to use for most zero-downtime deployments.