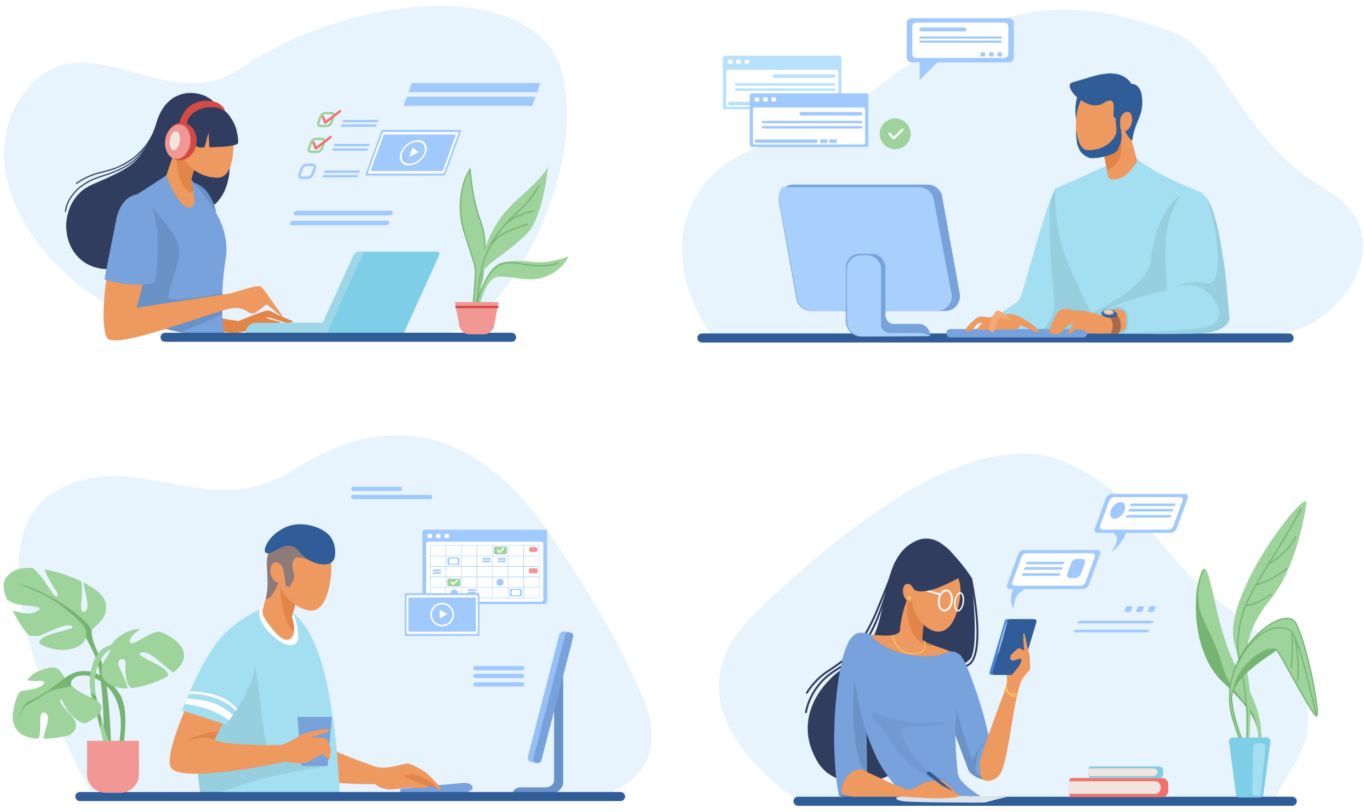
Creating a low cost recoverable EC2 instance utilising terraform

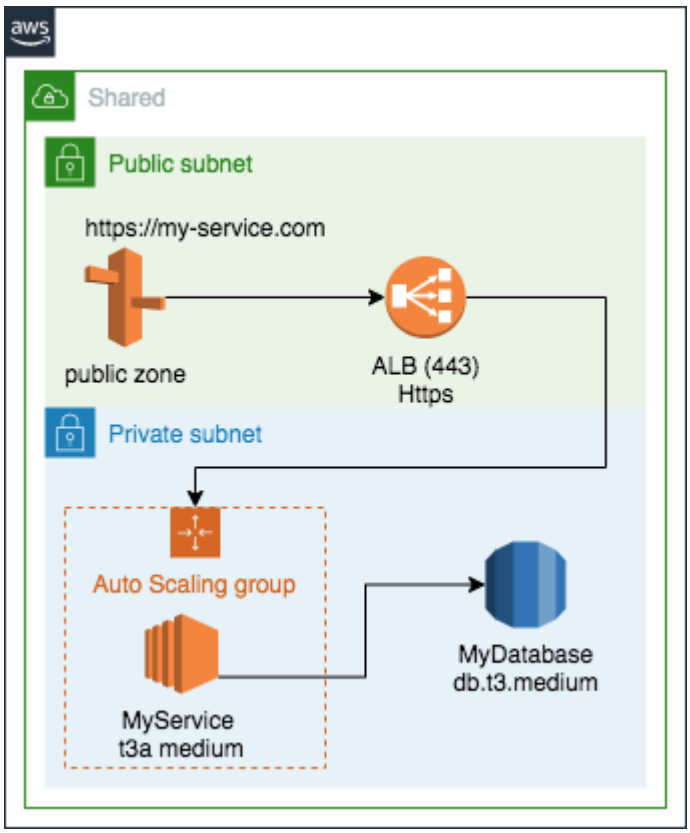
Craig Godden-Payne
Jul 7 · 5 min read ★



I've recently had to work on services which require to be installed on Windows, and because of this using a container was out of the question.

It was part of a migration which due to time constraints required lifting and shifting the machines rather than spending time automating and getting it right.

Due to cost, it was not necessary to create high availability as long as the service could recover within a reasonable time (10 minutes) without any loss of backing data.



The idea was to use EC2 in combination with an autoscaling group, to allow an almost always running service, which would automatically recover if terminated either by ourselves, or AWS.

The load balancer target group would be updated by the autoscaling group, so any downtime would be the length of the time for the ASG to make corrections.

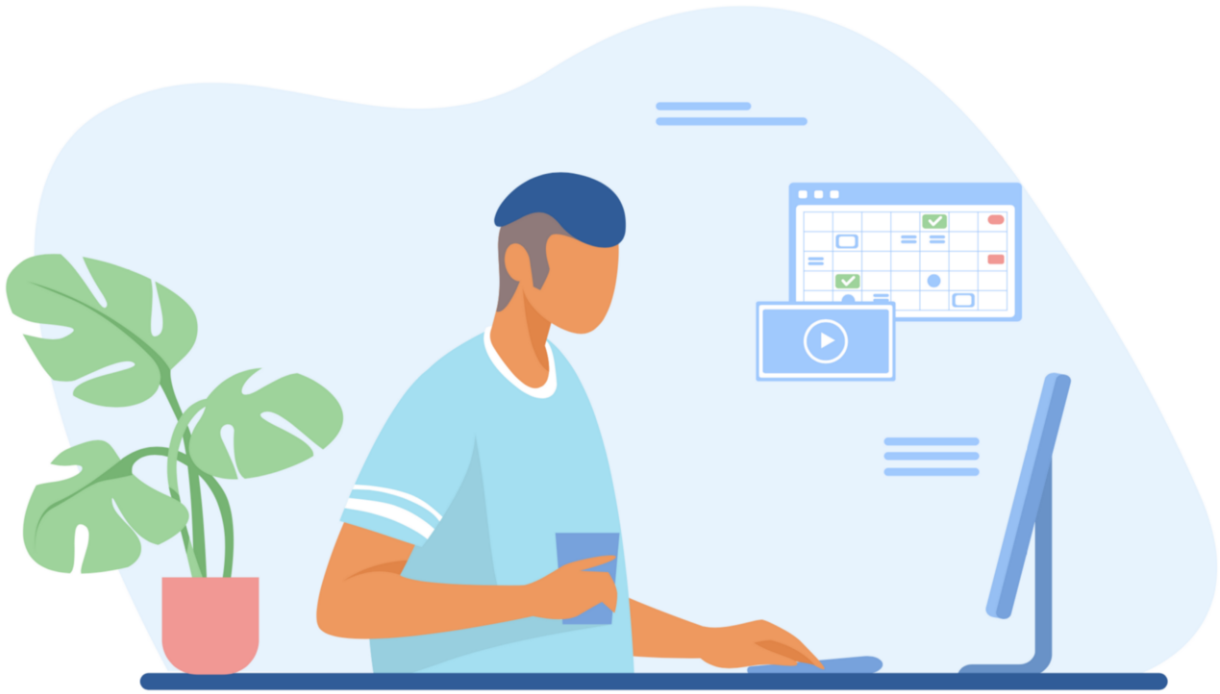
Where to start?

So the first part of this migration was to look at what was running on the existing server. It required to be installed as a windows service, and it connected to a MSSQL backend.

The first step in the mission was to figure out what size of EC2 would need to be used to be comparable to what currently existed, for me this was t3.medium.

I then was able to manually spin up my EC2 instance. This could (and probably should) be done with a tool like packer, but since time was not on my side, I had to work with the minimum to get it going.

I installed all the prerequisite software on the machine, and then was ready to make an AMI.



Warning

Make sure you use sysprep from within EC2 or you will have a nightmare trying to figure out why your AMI does not work as you expect.

> *Sysprep is a Microsoft tool that is used to capture custom Windows images. Sysprep removes unique information from the Amazon Elastic Compute Cloud (Amazon EC2) Windows instance, including the instance security identifiers (SID), computer name, and drivers. Duplicate SIDs can cause issues with Windows Server Update Services (WSUS), log-in issues, Windows volume key activation, Microsoft Office, and third-party products.*

Once I had my AMI of my image, it was time to start writing terraform to manage the infrastructure.

Using Terraform to automate

I used terraform to automate the provisioning of the infrastructure, after all this will likely be reworked many times.

I created a data lookup to find the AMI I created:

```
data aws_ami my_image {
   most_recent = true
   owners = ["self"]
   filter {
     name   = "name"
     values = ["my-service*"]
   }
 }
```

- This will lookup the latest version of the AMI based on the name wildcard. This is only applied via terraform, so just being clear, a new AMI will not automatically be used each time one is created, it will be terraform which updates the autoscaling group with the id of the latest AMI.

Then I create the load balancer that sits in front of the autoscaling instances, and security groups to enable access:

```
resource aws_alb public_my_service {
   name = "my-service-public"
   internal = false
   load_balancer_type = "application"
   security_groups = [aws_security_group.my_service_lb.id]
   subnets = ["192.168.0.0/24"]
   enable_deletion_protection = false
 }
```

```
resource aws_security_group my_service_lb {
   name = "my-service-lb"
   vpc_id = XXXXXXXXXX

   ingress {
     from_port = 80
     protocol = "TCP"
     to_port = 80
     cidr_blocks = ["x.x.x.x"]
     description = "Access to allow redirect to 443"
   }

   ingress {
     from_port = 443
     protocol = "TCP"
     to_port = 443
     cidr_blocks = ["x.x.x.x"]
     description = "Access to MyService"
   }

   egress {
     from_port = 0
     protocol = "-1"
     to_port = 0
     cidr_blocks = ["0.0.0.0/0"]
     description = "Allow all Outbound"
   }
 }
```

Once I have the security layer setup, I need to then create a security group for the instance to allow access from the load balancer

```
resource aws_security_group my_service_instance {
   name = "my-service-instance"
   vpc_id = XXXXXXXXXX

   ingress {
     from_port = 443
     protocol = "TCP"
     to_port = 443
     security_groups = [aws_security_group.my_service_lb.id]
     description = "Load Balancer Access"
   }

   ingress {
     from_port = 80
     protocol = "TCP"
     to_port = 80
     security_groups = [aws_security_group.my_service_lb.id]
     description = "Load Balancer Access"
   }

   egress {
```

```
    from_port = 0
    protocol = "-1"
    to_port = 0
    cidr_blocks = ["0.0.0.0/0"]
    description = "All Outbound"
  }
}
```

. . .

Next I need to create the target group that will eventually be updated by the autoscaling group.

```
resource aws_alb_target_group public_my_service {
  name        = "my-service"
  port        = 80
  protocol    = "HTTP"
  vpc_id      = XXXXXXXXXX
  target_type = "instance"
  health_check {
    port     = "traffic-port"
    protocol = "HTTP"
    path = "/health"
  }
}
```

I now need to create the autoscaling group to allow instantiating the instance and maintaining it at 1 running instance.

```
resource aws_autoscaling_group my_service_asg {
   name = "my-service"
   launch_template {
     id      = aws_launch_template.my_service_launch_template.id
     version = "$Latest"
   }
   max_size         = 2
   min_size         = 1
   desired_capacity = 1

   vpc_zone_identifier = ["XXXXXXXXX"]
   default_cooldown          = 180
   health_check_grace_period = 180
   health_check_type         = "EC2"
   target_group_arns = [aws_alb_target_group.public_my_service.arn]
   termination_policies = ["OldestLaunchConfiguration"]
   lifecycle { create_before_destroy = true }
 }

resource aws_launch_template my_service_launch_template {
   name = "my-service"
   image_id      = data.aws_ami.my_service_image.id
   instance_type = "t3.medium"
   monitoring { enabled = true }
   network_interfaces {
     associate_public_ip_address = false
     security_groups             =
[aws_security_group.my_service.id]
     delete_on_termination       = true
   }
   key_name = "my-service"
   lifecycle { create_before_destroy = true }

   tag_specifications {
     resource_type = "instance"
     tags          = { Name = "my-service" }
   }

   tag_specifications {
     resource_type = "volume"
     tags          = { Name = "my-service" }
   }
 }
```

And then to just make things a bit easier, a Route53 record to allow a better DNS name:

```
resource aws_route53_record my_service {
   name = "my-service"
```

```
    type = "A"
    zone_id = "XXXXXXXX"
    alias {
      evaluate_target_health = false
      name = aws_alb.public_my_service.dns_name
      zone_id = aws_alb.public_my_service.zone_id
    }
  }
```

Graphic Attributions:
https://www.freepik.com/free-photos-vectors/people
pch.vector

Ec2 AWS Automation Migration DevOps