After a few seconds, try the endpoint again:

```
$ curl http://localhost
Hello Terraform!
```

And there you go, the Deployment has rolled out your change automatically: under the hood, Deployments do a rolling deployment by default, similar to what you saw with Auto Scaling Groups (note that you can change deployment settings by adding a `strategy` block to the `kubernetes_deployment` resource).

Deploying Docker Containers in AWS Using Elastic Kubernetes Service

Kubernetes has one more trick up its sleeve: it's fairly portable. That is, you can reuse both the Docker images and the Kubernetes configurations in a totally different cluster and get similar results. To see this in action, let's now deploy a Kubernetes cluster in AWS.

Setting up and managing a secure, highly available, scalable Kubernetes cluster in the cloud from scratch is complicated. Fortunately, most cloud providers offer managed Kubernetes services, where they run the control plane and worker nodes for you: e.g., Elastic Kubernetes Service (EKS) in AWS, Azure Kubernetes Service (AKS) in Azure, and Google Kubernetes Engine (GKE) in Google Cloud. I'm going to show you how to deploy a very basic EKS cluster in AWS.

Create a new module in *modules/services/eks-cluster*, and define the API for the module in a *variables.tf* file with the following input variables:

```
variable "name" {
  description = "The name to use for the EKS cluster"
  type        = string
}

variable "min_size" {
  description = "Minimum number of nodes to have in the EKS
cluster"
  type        = number
}

variable "max_size" {
  description = "Maximum number of nodes to have in the EKS
cluster"
  type        = number
}

variable "desired_size" {
  description = "Desired number of nodes to have in the EKS
cluster"
  type        = number
}

variable "instance_types" {
  description = "The types of EC2 instances to run in the node
group"
  type        = list(string)
}
```

This code exposes input variables to set the EKS cluster's name, size, and the types of instances to use for the worker nodes. Next, in *main.tf*, create an IAM role for the control plane:

```
# Create an IAM role for the control plane
resource "aws_iam_role" "cluster" {
  name               = "${var.name}-cluster-role"
  assume_role_policy =
data.aws_iam_policy_document.cluster_assume_role.json
}

# Allow EKS to assume the IAM role
data "aws_iam_policy_document" "cluster_assume_role" {
  statement {
    effect  = "Allow"
    actions = ["sts:AssumeRole"]
    principals {
      type        = "Service"
      identifiers = ["eks.amazonaws.com"]
    }
  }
}

# Attach the permissions the IAM role needs
resource "aws_iam_role_policy_attachment"
"AmazonEKSClusterPolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"
  role       = aws_iam_role.cluster.name
}
```

This IAM role can be assumed by the EKS service, and it has a Managed IAM Policy attached that gives the control plane the permissions it needs. Now, add the `aws_vpc` and `aws_subnets` data sources to fetch information about the Default VPC and its subnets:

```
  # Since this code is only for learning, use the Default VPC and
  subnets.
  # For real-world use cases, you should use a custom VPC and
  private subnets.

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

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

Now you can create the control plane for the EKS cluster by using the `aws_eks_cluster` resource:

```
  resource "aws_eks_cluster" "cluster" {
    name     = var.name
    role_arn = aws_iam_role.cluster.arn
    version  = "1.21"

    vpc_config {
      subnet_ids = data.aws_subnets.default.ids
    }

    # Ensure that IAM Role permissions are created before and
  deleted after
    # the EKS Cluster. Otherwise, EKS will not be able to properly
  delete
    # EKS managed EC2 infrastructure such as Security Groups.
    depends_on = [
      aws_iam_role_policy_attachment.AmazonEKSClusterPolicy
    ]
  }
```

The preceding code configures the control plane to use the IAM role you just created, and to deploy into the Default VPC and subnets.

Next up are the worker nodes. EKS supports several different types of worker nodes: self-managed EC2 Instances (e.g., in an ASG that you create), AWS-managed EC2 Instances (known as a *managed node group*), and Fargate (serverless).[3] The simplest option to use for the examples in this chapter will be the managed node groups.

To deploy a managed node group, you first need to create another IAM role:

```
# Create an IAM role for the node group
resource "aws_iam_role" "node_group" {
  name               = "${var.name}-node-group"
  assume_role_policy =
data.aws_iam_policy_document.node_assume_role.json
}

# Allow EC2 instances to assume the IAM role
data "aws_iam_policy_document" "node_assume_role" {
  statement {
    effect  = "Allow"
    actions = ["sts:AssumeRole"]
    principals {
      type        = "Service"
      identifiers = ["ec2.amazonaws.com"]
    }
  }
}

# Attach the permissions the node group needs
resource "aws_iam_role_policy_attachment"
"AmazonEKSWorkerNodePolicy" {
  policy_arn =
"arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy"
  role       = aws_iam_role.node_group.name
}

resource "aws_iam_role_policy_attachment"
"AmazonEC2ContainerRegistryReadOnly" {
  policy_arn =
"arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly"
```

```
  role       = aws_iam_role.node_group.name
}

resource "aws_iam_role_policy_attachment" "AmazonEKS_CNI_Policy"
{
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy"
  role       = aws_iam_role.node_group.name
}
```

This IAM role can be assumed by the EC2 service (which makes sense, as managed node groups use EC2 Instances under the hood), and it has several Managed IAM Policies attached that give the managed node group the permissions it needs. Now you can use the `aws_eks_node_group` resource to create the managed node group itself:

```
resource "aws_eks_node_group" "nodes" {
  cluster_name    = aws_eks_cluster.cluster.name
  node_group_name = var.name
  node_role_arn   = aws_iam_role.node_group.arn
  subnet_ids      = data.aws_subnets.default.ids
  instance_types  = var.instance_types

  scaling_config {
    min_size     = var.min_size
    max_size     = var.max_size
    desired_size = var.desired_size
  }

  # Ensure that IAM Role permissions are created before and
deleted after
  # the EKS Node Group. Otherwise, EKS will not be able to
properly
  # delete EC2 Instances and Elastic Network Interfaces.
  depends_on = [
    aws_iam_role_policy_attachment.AmazonEKSWorkerNodePolicy,

aws_iam_role_policy_attachment.AmazonEC2ContainerRegistryReadOnly
,
    aws_iam_role_policy_attachment.AmazonEKS_CNI_Policy,
  ]
}
```

This code configures the managed node group to use the control plane and IAM role you just created, to deploy into the Default VPC, and to use the name, size, and instance type parameters passed in as input variables.

In *outputs.tf*, add the following output variables:

```
output "cluster_name" {
  value       = aws_eks_cluster.cluster.name
  description = "Name of the EKS cluster"
}

output "cluster_arn" {
  value       = aws_eks_cluster.cluster.arn
  description = "ARN of the EKS cluster"
}

output "cluster_endpoint" {
  value       = aws_eks_cluster.cluster.endpoint
  description = "Endpoint of the EKS cluster"
}

output "cluster_certificate_authority" {
  value       = aws_eks_cluster.cluster.certificate_authority
  description = "Certificate authority of the EKS cluster"
}
```

OK, the `eks-cluster` module is now ready to roll. Let's use it and the `k8s-app` module from earlier to deploy an EKS cluster and to deploy the `training/webapp` Docker image into that cluster. Create *examples/kubernetes-eks/main.tf*, and configure the `eks-cluster` module as follows:

```
provider "aws" {
  region = "us-east-2"
}

module "eks_cluster" {
  source = "../../modules/services/eks-cluster"

  name         = "example-eks-cluster"
  min_size     = 1
  max_size     = 2
  desired_size = 1

  # Due to the way EKS works with ENIs, t3.small is the smallest
  # instance type that can be used for worker nodes. If you try
```

```
    # something smaller like t2.micro, which only has 4 ENIs,
    # they'll all be used up by system services (e.g., kube-proxy)
    # and you won't be able to deploy your own Pods.
    instance_types = ["t3.small"]
  }
```

Next, configure the `k8s-app` module as follows:

```
provider "kubernetes" {
  host = module.eks_cluster.cluster_endpoint
  cluster_ca_certificate = base64decode(
    module.eks_cluster.cluster_certificate_authority[0].data
  )
  token = data.aws_eks_cluster_auth.cluster.token
}

data "aws_eks_cluster_auth" "cluster" {
  name = module.eks_cluster.cluster_name
}

module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Terraform"
  }

  # Only deploy the app after the cluster has been deployed
  depends_on = [module.eks_cluster]
}
```

The preceding code configures the Kubernetes provider to authenticate to the EKS cluster, rather than your local Kubernetes cluster (from Docker Desktop). It then uses the `k8s-app` module to deploy the `training/webapp` Docker image exactly the same way as you did when deploying it to Docker Desktop; the only difference is the addition of the `depends_on` parameter to ensure that Terraform only tries to deploy the Docker image after the EKS cluster has been deployed.

Next, pass through the service endpoint as an output variable:

```
output "service_endpoint" {
  value       = module.simple_webapp.service_endpoint
  description = "The K8S Service endpoint"
}
```

OK, now you're ready to deploy! Run `terraform apply` as usual (note that EKS clusters can take 10–20 minutes to deploy, so be patient):

```
$ terraform apply

(...)

Apply complete! Resources: 10 added, 0 changed, 0 destroyed.

Outputs:

service_endpoint = "http://774696355.us-east-2.elb.amazonaws.com"
```

Wait a little while for the web app to spin up and pass health checks, and then test out the `service_endpoint`:

```
$ curl http://774696355.us-east-2.elb.amazonaws.com
Hello Terraform!
```

And there you have it! The same Docker image and Kubernetes code is now running in an EKS cluster in AWS, just the way it ran on your local computer. All the same features work here too. For example, try updating `environment_variables` to a different `PROVIDER` value, such as "Readers":

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Readers"
  }

  # Only deploy the app after the cluster has been deployed
  depends_on = [module.eks_cluster]
}
```

Rerun `apply`, and just a few seconds later, the Kubernetes Deployment will have deployed the changes:

```
$ curl http://774696355.us-east-2.elb.amazonaws.com
Hello Readers!
```

This is one of the advantages of using Docker: changes can be deployed very quickly.

You can use `kubectl` again to see what's happening in your cluster. To authenticate `kubectl` to the EKS cluster, you can use the `aws eks update-kubeconfig` command to automatically update your *$HOME/.kube/config* file:

```
$ aws eks update-kubeconfig --region <REGION> --name
<EKS_CLUSTER_NAME>
```

where `REGION` is the AWS region and `EKS_CLUSTER_NAME` is the name of your EKS cluster. In the Terraform module, you deployed to the `us-east-2` region and named the cluster `kubernetes-example`, so the command will look like this:

```
$ aws eks update-kubeconfig --region us-east-2 --name kubernetes-
example
```

Now, just as before, you can use the `get nodes` command to inspect the worker nodes in your cluster, but this time, add the `-o wide` flag to get a bit more info:

```
$ kubectl get nodes
NAME                                STATUS   AGE   EXTERNAL-IP
OS-IMAGE
```

```
xxx.us-east-2.compute.internal   Ready    22m   3.134.78.187
Amazon Linux 2
```

The preceding snippet is highly truncated to fit into the book, but in the real output, you should be able to see the one worker node, its internal and external IP, version information, OS information, and much more.

You can use the `get deployments` command to inspect your Deployments:

```
$ kubectl get deployments
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
simple-webapp   2/2     2            2           19m
```

Next, run `get pods` to see the Pods:

```
$ kubectl get pods
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
simple-webapp   2/2     2            2           19m
```

And finally, run `get services` to see the Services:

```
$ kubectl get services
NAME            TYPE           EXTERNAL-IP
PORT(S)
kubernetes      ClusterIP      <none>
443/TCP
simple-webapp   LoadBalancer   774696355.us-east-
2.elb.amazonaws.com    80/TCP
```

You should be able to see your load balancer and the URL you used to test it.

So there you have it: two different providers, both working in the same cloud, helping you to deploy containerized workloads.

That said, just as in previous sections, I want to leave you with a few warnings:

Warning 1: These Kubernetes examples are very simplified!

Kubernetes is complicated, and it's rapidly evolving and changing; trying to explain all the details can easily fill a book all by itself. Since this is a book about Terraform, and not Kubernetes, my goal with the Kubernetes examples in this chapter was to keep them as simple and minimal as possible. Therefore, while I hope the code examples you've seen have been useful from a learning and experimentation perspective, if you are going to use Kubernetes for real-world, production use cases, you'll need to change many aspects of this code, such as configuring a number of additional services and settings in the `eks-cluster` module (e.g., ingress controllers, secret envelope encryption, security groups, OIDC authentication, Role-Based Access Control (RBAC) mapping, VPC CNI, kube-proxy, CoreDNS), exposing many other settings in the `k8s-app` module (e.g., secrets management, volumes, liveness probes, readiness probes, labels, annotations, multiple ports, multiple containers), and using a custom VPC with private subnets for your EKS cluster instead of the Default VPC and public subnets.[4]

Warning 2: Use multiple providers sparingly

Although you certainly can use multiple providers in a single module, I don't recommend doing it too often, for similar reasons to why I don't recommend using provider aliases too often: in most cases, you want each provider to be isolated in its own module so that you can manage it separately and limit the blast radius from mistakes or attackers.

Moreover, Terraform doesn't have great support for dependency ordering between providers. For example, in the Kubernetes example, you had a single module that deployed both the EKS cluster, using the AWS Provider, and a Kubernetes app into that cluster, using the Kubernetes provider. As it turns out, the Kubernetes provider documentation explicitly recommends *against* this pattern: