

# How to create EKS Cluster using Terraform MODULES?

- You can find the source code for this video in my [GitHub Repo](#).

## Intro

In this video, we're going to create an [EKS](#) cluster using open-source terraform modules. First, we will create VPC from scratch and then provision Kubernetes.

- I'll show you how to add additional users to EKS by modifying the `aws-auth` configmap. We will create an IAM role with full access to Kubernetes API and let users to assume that role if they need access to EKS.
- To automatically scale the EKS cluster, we will deploy `cluster-autoscaler` using plain YAML, and `kubectl` terraform provider.
- Finally, we will deploy [AWS Load Balancer Controller](#) using the `helm` provider and create a test ingress resource.

I also have another [tutorial](#) where I use terraform resources instead of modules to create an EKS cluster.

## Create AWS VPC using Terraform

First of all, we need to define `aws` terraform provider. You have multiple ways to authenticate with AWS. It will depend on how and where you run terraform. For example, if you use your laptop to create an EKS cluster, you can simply create a local AWS profile with the `aws configure` command. If you run terraform from an EC2 instance, you should create an instance profile with the required IAM policies.

It's a best practice to define version constraints for each provider, but since in this video we will be using terraform `aws modules`, they already come with version constraints. We only need to require terraform version itself along with `kubectl` and `helm` providers. We will discuss later why I chose to use `kubectl` instead of kubernetes provider to deploy cluster autoscaler.

### terraform/0-aws-provider.tf

```
1 provider "aws" {
2     region = "us-east-1"
```

```

3 }
4
5 terraform {
6   required_providers {
7     kubectl = {
8       source  = "gavinbunney/kubectl"
9       version = ">= 1.14.0"
10    }
11    helm = {
12      source  = "hashicorp/helm"
13      version = ">= 2.6.0"
14    }
15  }
16
17  required_version = "~> 1.0"
18 }

```

To create AWS VPC, we use `terraform-aws-module` and the latest version at this moment. Let's call it `main` and provide a CIDR range. For EKS, you need at least two availability zones. Let's use `us-east-1a` and `1b`. Almost in all cases, you want to deploy your Kubernetes workers in the private subnets with a default route to NAT Gateway. However, if you're going to expose your application to the internet, you would need public subnets with a default route to the Internet Gateway. We would need to update subnet tags later in the tutorial for the AWS Load balancer controller to discover them.

Now you have multiple options for how you want to deploy the NAT gateway. You can deploy one single NAT Gateway in one availability zone or choose to create a highly available setup and deploy one NAT Gateway per zone. It depends on your budget and requirements. I always prefer to create a single NAT gateway and allocate multiple Elastic IP addresses.



## Download PDF (Free)

File size: 487KB. Full Version

PDF Central

Dc

Next is DNS support. It's common for many AWS services to require DNS, for example, if you want to use the `EFS file system` in your EKS cluster. It's handy in some cases because it allows `ReadWriteMany` mode and mount a single volume to multiple Kubernetes pods.

```

terraform/1-vpc.tf

1 module "vpc" {
2   source  = "terraform-aws-modules/vpc/aws"
3   version = "3.14.3"
4
5   name = "main"
6   cidr = "10.0.0.0/16"
7
8   azs          = ["us-east-1a", "us-east-1b"]
9   private_subnets = ["10.0.0.0/19", "10.0.32.0/19"]
10  public_subnets  = ["10.0.64.0/19", "10.0.96.0/19"]
11
12  enable_nat_gateway  = true
13  single_nat_gateway  = true
14  one_nat_gateway_per_az = false
15
16  enable_dns_hostnames = true
17  enable_dns_support   = true
18
19  tags = {
20    Environment = "staging"
21  }
22 }

```

## Create EKS using Terraform

Now we have all the components that we need to create an EKS cluster. Let's call it `my-eks` and specify the latest supported version by AWS. Right now, it's [1.23](#). If you have a bastion host or a VPN, you can enable a `private endpoint` and use it to access your cluster. Since we just created VPC, I don't have either one. I would need to enable a `public endpoint` as well to access it from my laptop.

Next is a `VPC ID` that you can dynamically pull from the VPC module. You must also provide `subnets` to your cluster where EKS will deploy workers. Let's use only private subnets. To grant access to your applications running in the EKS cluster, you can either attach the IAM role with required IAM policies to the nodes or use a more secure option which is to enable [IAM Roles for Service Accounts](#). In that way, you can limit the IAM role to a single pod. Then the node's configuration. For example, you can specify the disk size for each worker.

A screenshot of a LinkedIn job listing. The title is "Solution Architect, Healthcare IT" located in Competitive, Oracle Corporation (UK) Ltd. Below the title, there is a button labeled "Apply now". In the bottom right corner of the listing, it says "Search 200,000+ jobs in the UK" and "Total jobs".

To run the workload on your Kubernetes cluster, you need to provision instance groups. You have three options.

- You can use [EKS-managed nodes](#); that is recommended approach. In that way, EKS can perform rolling upgrades for you almost without downtime if you properly define [PodDisruptionBudget](#) policies.
- Then you can use [self-managed groups](#). Basically, terraform will create a launch template with an auto-scaling group as your node pool and join the cluster. Using this approach, you would need to maintain your nodes yourself.
- Finally, you can use the [Fargate profile](#). This option allows you to only work on your workload, and EKS will manage nodes for you. It will create a dedicated node for each of your pods. It can potentially save you money if Kubernetes is badly mismanaged.

Let's create `managed node groups` for this example. First is a standard node group. You can assign custom labels such as `role` equal to `general`. It's helpful to use custom labels in Kubernetes deployment specifications in case you need to create a new node group and migrate your workload there. If you use built-in labels, they are tight to your node group. The next group is similar, but we use `spot` nodes. Those nodes are cheaper, but AWS can take them at any time. Also, you can set taints on your node group.

#### terraform/2-eks.tf

```

1 module "eks" {
2   source  = "terraform-aws-modules/eks/aws"
3   version = "18.29.0"
4
5   cluster_name    = "my-eks"
6   cluster_version = "1.23"
7
8   cluster_endpoint_private_access = true
9   cluster_endpoint_public_access  = true
10
11  vpc_id       = module.vpc.vpc_id
12  subnet_ids  = module.vpc.private_subnets
13
14  enable_irsa = true
15
16  eks_managed_node_group_defaults = {
17    disk_size = 50
18  }
19
20  eks_managed_node_groups = {
21    general = {
22      desired_size = 1
23      min_size     = 1
24      max_size     = 10
25
26      labels = {
27        role = "general"
28      }
29

```

```

30     instance_types = ["t3.small"]
31     capacity_type = "ON_DEMAND"
32   }
33
34   spot = {
35     desired_size = 1
36     min_size     = 1
37     max_size     = 10
38
39   labels = {
40     role = "spot"
41   }
42
43   taints = [ {
44     key   = "market"
45     value = "spot"
46     effect = "NO_SCHEDULE"
47   }]
48
49   instance_types = ["t3.micro"]
50   capacity_type = "SPOT"
51 }
52 }
53
54 tags = {
55   Environment = "staging"
56 }
57 }
```

That's all for now; let's go to the terminal and run terraform. Initialize first and then apply. Usually, it takes up to 10 minutes to create an EKS cluster.

```
terraform init
terraform apply
```

Before you can connect to the cluster, you need to update the Kubernetes context with the following command:

```
aws eks update-kubeconfig --name my-eks --region us-east-1
```

Then a quick check to verify that we can access EKS.

```
kubectl get nodes
```

## Add IAM User & Role to EKS

Next, I want to show you how to grant access to Kubernetes workloads to other IAM users and IAM roles. Access to the EKS is managed by using the aws-auth config map in the kube-system namespace. Initially, only the user that created a cluster can access Kubernetes and modify that configmap. Unless you provisioned EKS for your personal project, you most likely need to grant access to Kubernetes to your team members.

Terraform module that we used to create an EKS can manage permissions on your behalf. You have two options.

- You can add IAM users directly to the eks configmap. In that case, whenever you need to add someone to the cluster, you need to update the aws-auth configmap, which is not very convenient.
- The second, much better approach is to grant access to the IAM role just once using the aws-auth configmap, and then you can simply allow users outside of EKS to assume that role. Since IAM groups are not supported in EKS, this is the preferred option.

In this example, we create an IAM role with the necessary permissions and allow the IAM user to assume that role.

First, let's create an allow-eks-access IAM policy with eks:DescribeCluster action. This action is needed to initially update the Kubernetes context and get access to the cluster.

```
terraform/3-iam.tf

1 module "allow_eks_access_iam_policy" {
2   source  = "terraform-aws-modules/iam/aws//modules/iam-policy"
3   version = "5.3.1"
4
5   name      = "allow-eks-access"
6   create_policy = true
7 }
```

```

8  policy = jsonencode({
9    Version = "2012-10-17"
10   Statement = [
11     {
12       Action = [
13         "eks:DescribeCluster",
14       ]
15       Effect  = "Allow"
16       Resource = "*"
17     },
18   ],
19 }
20 }
```

Next is the IAM role that we will use to access the cluster. Let's call it eks-admin since we're going to bind it with the Kubernetes system:masters RBAC group with full access to the Kubernetes API. Optionally this module allows you to enable two-factor authentication, but it's out of the scope of this tutorial.

Then attach the IAM policy that we just created and, most importantly, define trusted role arns. By specifying the root potentially, every IAM user in your account could use this role. To allow the user to assume this role, we still need to attach an additional policy to the user.

#### terraform/3-iam.tf

```

22 module "eks_admins_iam_role" {
23   source  = "terraform-aws-modules/iam/aws//modules/iam-assumable-role"
24   version = "5.3.1"
25
26   role_name      = "eks-admin"
27   create_role    = true
28   role_requires_mfa = false
29
30   custom_role_policy_arns = [module.allow_eks_access_iam_policy.arn]
31
32   trusted_role_arns = [
33     "arn:aws:iam::${module.vpc.vpc_owner_id}:root"
34   ]
35 }
```

The IAM role is ready, now let's create a test IAM user that gets access to that role. Let's call it user1 and disable creating access keys and login profiles. We will generate those from the UI.

#### terraform/3-iam.tf

```

37 module "user1_iam_user" {
38   source  = "terraform-aws-modules/iam/aws//modules/iam-user"
39   version = "5.3.1"
40
41   name          = "user1"
42   create_iam_access_key = false
43   create_iam_user_login_profile = false
44
45   force_destroy = true
46 }
```

Then IAM policy to allow assume eks-admin IAM role.

#### terraform/3-iam.tf

```

48 module "allow_assume_eks_admins_iam_policy" {
49   source  = "terraform-aws-modules/iam/aws//modules/iam-policy"
50   version = "5.3.1"
51
52   name          = "allow-assume-eks-admin-iam-role"
53   create_policy = true
54
55   policy = jsonencode({
56     Version = "2012-10-17"
57     Statement = [
58       {
59         Action = [
60           "sts:AssumeRole",
61         ]
62     }
63   })
64 }
```

```

62     Effect  = "Allow"
63     Resource = module.eks_admins_iam_role.iam_role_arn
64   },
65 }
66 }
67 }

```

Finally, we need to create an IAM group with the previous policy and put our user1 in this group.

```

terraform/3-iam.tf

69 module "eks_admins_iam_group" {
70   source  = "terraform-aws-modules/iam/aws//modules/iam-group-with-policies"
71   version = "5.3.1"
72
73   name          = "eks-admin"
74   attach_iam_self_management_policy = false
75   create_group  = true
76   group_users   = [module.user1_iam_user.iam_user_name]
77   custom_group_policy_arns = [module.allow_assume_eks_admins_iam_policy.arn]
78 }

```

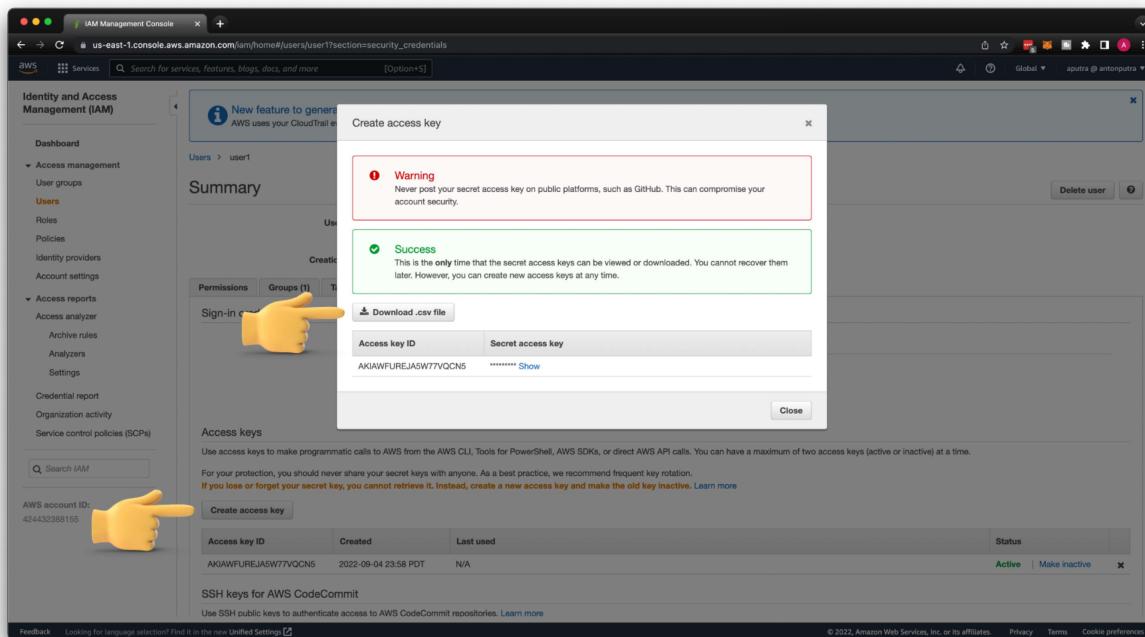
Let's go ahead and apply terraform to create all those IAM entities.

```

terraform init
terraform apply

```

Now let's generate new credentials for user1 and create a local AWS profile.



To create an AWS profile, you need to run `aws configure` and provide the profile name, in our case, `user1`.

```

aws configure --profile user1

```

Then verify that you can access AWS services with that profile.

```

aws sts get-caller-identity --profile user1

```

To let user1 to assume the eks-admin IAM role, we need to create another AWS profile with the role name. You need to replace `role_arn` with yours.

```
vim ~/.aws/config
```

#### ~/.aws/config

```
1 [profile eks-admin]
2 role_arn = arn:aws:iam::424432388155:role/eks-admin
3 source_profile = user1
```

Let's test if we can assume the `eks-admin` IAM role.

```
aws sts get-caller-identity --profile eks-admin
```

Now we can update Kubernetes config to use the `eks-admin` IAM role.

```
aws eks update-kubeconfig \
--name my-eks \
--region us-east-1 \
--profile eks-admin
```

If you try to access EKS right now, you'll get an error saying `You must be logged in to the server (Unauthorized).`

```
kubectl auth can-i "*" "*"
```

To add the `eks-admin` IAM role to the EKS cluster, we need to update the `aws-auth` configmap.

#### terraform/2-eks.tf

```
54 manage_aws_auth_configmap = true
55 aws_auth_roles = [
56   {
57     rolearn = module.eks_admins_iam_role.iam_role.arn
58     username = module.eks_admins_iam_role.iam_role.name
59     groups   = ["system:masters"]
60   },
61 ]
```

Also, you need to authorize terraform to access Kubernetes API and modify `aws-auth` configmap. To do that, you need to define `terraform kubernetes` provider. To authenticate with the cluster, you can use either use `token` which has an expiration time or an `exec` block to retrieve this token on each terraform run.

#### terraform/2-eks.tf

```
68 # https://github.com/terraform-aws-modules/terraform-aws-eks/issues/2009
69 data "aws_eks_cluster" "default" {
70   name = module.eks.cluster_id
71 }
72
73 data "aws_eks_cluster_auth" "default" {
74   name = module.eks.cluster_id
75 }
76
77 provider "kubernetes" {
78   host           = data.aws_eks_cluster.default.endpoint
79   cluster_ca_certificate = base64decode(data.aws_eks_cluster.default.certificate_authority[0].data)
80   # token ..... = data.aws_eks_cluster_auth.default.token
81
82   exec {
83     api_version = "client.authentication.k8s.io/v1beta1"
84     args        = ["eks", "get-token", "--cluster-name", data.aws_eks_cluster.default.id]
85     command     = "aws"
86   }
87 }
```

Now you can run terraform.

```
terraform apply
```

Let's check if we can access the cluster using the `eks-admin` role.

```
kubectl auth can-i "*" "*"
```

Since we mapped the `eks-admin` role with the Kubernetes `system:masters` RBAC group, we have full access to the Kubernetes API.

Suppose you want to grant `read-only` access to the cluster, for example, for your developers. You can create a custom Kubernetes RBAC group and map it to the `IAM` role.

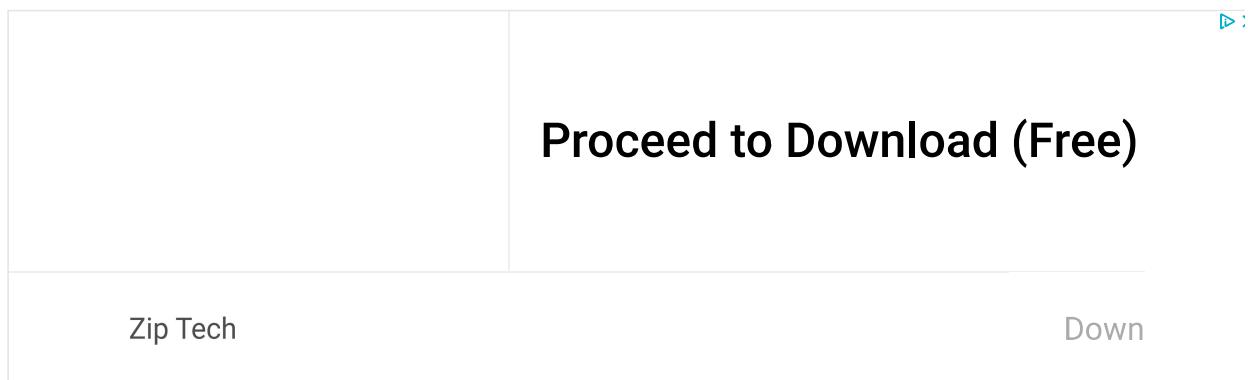
## Deploy Cluster Autoscaler

One of the many reasons we chose to use Kubernetes is that it can automatically scale based on the load. To autoscale the Kubernetes cluster, you need to deploy an additional component. You also have at least two options.

- You can deploy [Karpenter](#), which creates Kubernetes nodes using EC2 instances. Based on your workload, it can select the appropriate EC2 instance type. I have a [video](#) dedicated to Karpenter if you want to learn more.
- The second option is to use [cluster-autoscaler](#). It uses auto-scaling groups to adjust the `desired size` based on your load.

In my opinion, Karpenter would be a more efficient way to scale Kubernetes because it's not tight to the auto-scaling group. It is something between `cluster-autoscaler` and [Fargate profile](#).

Since I already have a dedicated tutorial for Karpenter, let's deploy cluster-autoscaler in this video. We have already created OpenID connect provider to enable [IAM roles for service accounts](#). Now we can use another terraform module, `iam-role-for-service-accounts-eks`, to create an `IAM` role for the cluster-autoscaler.



It needs AWS permissions to access and modify AWS auto-scaling groups. Let's call this role `cluster-autoscaler`. Then we need to specify the Kubernetes namespace and a service account name where we're going to deploy cluster-autoscaler.

```
terraform/4-autoscaler-iam.tf
1 module "cluster_autoscaler_irsa_role" {
2   source  = "terraform-aws-modules/iam/aws//modules/iam-role-for-service-accounts-eks"
3   version = "5.3.1"
4
5   role_name          = "cluster-autoscaler"
6   attach_cluster_autoscaler_policy = true
7   cluster_autoscaler_cluster_ids  = [module.eks.cluster_id]
8
9   oidc_providers = {
10     ex = {
11       provider_arn      = module.eks.oidc_provider_arn
12       namespace_service_accounts = ["kube-system:cluster-autoscaler"]
13     }
14   }
15 }
```

Now let's deploy autoscaler to Kubernetes. We're going to use [Helm](#) next to deploy the [AWS load balancer controller](#). To give you other options, I'll use plain [YAML](#) to deploy cluster-autoscaler. For YAML, you can use the `kubernetes` provider that we have already defined, or you can use the `kubectl` provider.

- With the `kubernetes` provider, there is no option, for now, to wait till EKS is provisioned before applying YAML. In that case, you would need to split your workflow of creating EKS into two parts. First, create a cluster, then apply terraform again and deploy autoscaler.

- On the other hand, the `kubectl` provider can wait till EKS is ready and then apply YAML in a single workflow.

When deploying autoscaler, preferably, you need to match the EKS version with the autoscaler version.

#### terraform/5-autoscaler-manifest.tf

```

1 provider "kubectl" {
2   host           = data.aws_eks_cluster.default.endpoint
3   cluster_ca_certificate = base64decode(data.aws_eks_cluster.default.certificate_authority[0].data)
4   load_config_file = false
5
6   exec {
7     api_version = "client.authentication.k8s.io/v1beta1"
8     args        = ["eks", "get-token", "--cluster-name", data.aws_eks_cluster.default.id]
9     command     = "aws"
10   }
11 }
12
13 resource "kubectl_manifest" "service_account" {
14   yaml_body = <<-EOF
15   apiVersion: v1
16   kind: ServiceAccount
17   metadata:
18     labels:
19       k8s-addon: cluster-autoscaler.addons.k8s.io
20       k8s-app: cluster-autoscaler
21     name: cluster-autoscaler
22     namespace: kube-system
23     annotations:
24       eks.amazonaws.com/role-arn: ${module.cluster_autoscaler_irsa_role.iam_role_arn}
25 EOF
26 }
27
28 resource "kubectl_manifest" "role" {
29   yaml_body = <<-EOF
30   apiVersion: rbac.authorization.k8s.io/v1
31   kind: Role
32   metadata:
33     name: cluster-autoscaler
34     namespace: kube-system
35     labels:
36       k8s-addon: cluster-autoscaler.addons.k8s.io
37       k8s-app: cluster-autoscaler
38   rules:
39     - apiGroups: []
40       resources: ["configmaps"]
41       verbs: ["create", "list", "watch"]
42     - apiGroups: []
43       resources: ["configmaps"]
44       resourceNames: ["cluster-autoscaler-status", "cluster-autoscaler-priority-expander"]
45       verbs: ["delete", "get", "update", "watch"]
46 EOF
47 }
48
49 resource "kubectl_manifest" "role_binding" {
50   yaml_body = <<-EOF
51   apiVersion: rbac.authorization.k8s.io/v1
52   kind: RoleBinding
53   metadata:
54     name: cluster-autoscaler
55     namespace: kube-system
56     labels:
57       k8s-addon: cluster-autoscaler.addons.k8s.io
58       k8s-app: cluster-autoscaler
59   roleRef:
60     apiGroup: rbac.authorization.k8s.io
61     kind: Role
62     name: cluster-autoscaler
63   subjects:
64     - kind: ServiceAccount
65       name: cluster-autoscaler
66       namespace: kube-system
67 EOF
68 }
69
70 resource "kubectl_manifest" "cluster_role" {
71   yaml_body = <<-EOF
72   apiVersion: rbac.authorization.k8s.io/v1

```

```

73 kind: ClusterRole
74 metadata:
75   name: cluster-autoscaler
76   labels:
77     k8s-addon: cluster-autoscaler.addons.k8s.io
78     k8s-app: cluster-autoscaler
79   rules:
80     - apiGroups: [""]
81       resources: ["events", "endpoints"]
82       verbs: ["create", "patch"]
83     - apiGroups: [""]
84       resources: ["pods/eviction"]
85       verbs: ["create"]
86     - apiGroups: [""]
87       resources: ["pods/status"]
88       verbs: ["update"]
89     - apiGroups: [""]
90       resources: ["endpoints"]
91       resourceName: ["cluster-autoscaler"]
92       verbs: ["get", "update"]
93     - apiGroups: [""]
94       resources: ["nodes"]
95       verbs: ["watch", "list", "get", "update"]
96     - apiGroups: [""]
97       resources:
98         - "namespaces"
99         - "pods"
100        - "services"
101        - "replicationcontrollers"
102        - "persistentvolumeclaims"
103        - "persistentvolumes"
104       verbs: ["watch", "list", "get"]
105     - apiGroups: ["extensions"]
106       resources: ["replicaset", "daemonset"]
107       verbs: ["watch", "list", "get"]
108     - apiGroups: ["policy"]
109       resources: ["poddisruptionbudgets"]
110       verbs: ["watch", "list"]
111     - apiGroups: ["apps"]
112       resources: ["statefulset", "replicaset", "daemonset"]
113       verbs: ["watch", "list", "get"]
114     - apiGroups: ["storage.k8s.io"]
115       resources: ["storageclasses", "csinodes", "csidrivers", "csistoragecapacities"]
116       verbs: ["watch", "list", "get"]
117     - apiGroups: ["batch", "extensions"]
118       resources: ["jobs"]
119       verbs: ["get", "list", "watch", "patch"]
120     - apiGroups: ["coordination.k8s.io"]
121       resources: ["leases"]
122       verbs: ["create"]
123     - apiGroups: ["coordination.k8s.io"]
124       resourceName: ["cluster-autoscaler"]
125       resources: ["leases"]
126       verbs: ["get", "update"]
127 EOF
128 }
129
130 resource "kubectl_manifest" "cluster_role_binding" {
131   yaml_body = <<-EOF
132   apiVersion: rbac.authorization.k8s.io/v1
133   kind: ClusterRoleBinding
134   metadata:
135     name: cluster-autoscaler
136     labels:
137       k8s-addon: cluster-autoscaler.addons.k8s.io
138       k8s-app: cluster-autoscaler
139   roleRef:
140     apiGroup: rbac.authorization.k8s.io
141     kind: ClusterRole
142     name: cluster-autoscaler
143   subjects:
144     - kind: ServiceAccount
145       name: cluster-autoscaler
146       namespace: kube-system
147 EOF
148 }
149
150 resource "kubectl_manifest" "deployment" {
151   yaml_body = <<-EOF
152   apiVersion: apps/v1

```

```

153 kind: Deployment
154   metadata:
155     name: cluster-autoscaler
156     namespace: kube-system
157     labels:
158       app: cluster-autoscaler
159   spec:
160     replicas: 1
161     selector:
162       matchLabels:
163         app: cluster-autoscaler
164     template:
165       metadata:
166         labels:
167           app: cluster-autoscaler
168       spec:
169         priorityClassName: system-cluster-critical
170         securityContext:
171           runAsNonRoot: true
172           runAsUser: 65534
173           fsGroup: 65534
174         serviceAccountName: cluster-autoscaler
175         containers:
176           - image: k8s.gcr.io/autoscaling/cluster-autoscaler:v1.23.1
177             name: cluster-autoscaler
178             resources:
179               limits:
180                 cpu: 100m
181                 memory: 600Mi
182               requests:
183                 cpu: 100m
184                 memory: 600Mi
185             command:
186               - ./cluster-autoscaler
187               - --v=4
188               - --stderrthreshold=info
189               - --cloud-provider=aws
190               - --skip-nodes-with-local-storage=false
191               - --expander=least-waste
192               - --node-group-auto-discovery=asg:tag=k8s.io/cluster-autoscaler/enabled,k8s.io/cluster-
193             autoscaler/${module.eks.cluster_id}
194             volumeMounts:
195               - name: ssl-certs
196                 mountPath: /etc/ssl/certs/ca-certificates.crt
197                 readOnly: true
198             volumes:
199               - name: ssl-certs
200                 hostPath:
201                   path: "/etc/ssl/certs/ca-bundle.crt"
202 EOF
}

```

Go back to the terminal and apply terraform.

```

terraform init
terraform apply

```

Verify that the autoscaler is running.

```

kubectl get pods -n kube-system

```

To test autoscaler, let's create nginx deployment.

#### k8s/nginx.yaml

```

1  ---
2  apiVersion: apps/v1
3  kind: Deployment
4  metadata:
5    name: nginx-deployment
6  spec:
7    replicas: 4
8    selector:
9      matchLabels:
10        app: nginx
11    template:

```

```

12   metadata:
13     labels:
14       app: nginx
15   spec:
16     containers:
17       - name: nginx
18         image: nginx:1.14.2
19       resources:
20         requests:
21           cpu: "1"

```

In a separate terminal, you can watch autoscaler logs just to make sure you don't have any errors.

```
kubectl logs -f \
-n kube-system \
-l app=cluster-autoscaler
```

Now let's apply nginx Kubernetes deployment.

```
kubectl apply -f k8s/nginx.yaml
```

In a few seconds, you should get a few more nodes.

```
watch -n 1 -t kubectl get nodes
```

## Deploy AWS Load Balancer Controller

Finally, let's deploy the [AWS Load Balancer Controller](#) to the EKS cluster. You can use it to create `ingresses` as well as `services` of type `LoadBalancer`. For the ingress load balancer controller creates an [application load balancer](#), and for the service, it creates a [network load balancer](#).

I also have a detailed tutorial and a bunch of examples of how to use this controller. In this video, we are going to deploy it with Helm and quickly verify that we can create ingress.

The screenshot shows a PDF Central download page. At the top right is a blue 'Download' button. Below it is a large red 'Download PDF (Free)' button. Underneath that, the text 'File size: 487KB, Full Version' is displayed. At the bottom left is the 'PDF Central' logo, and at the bottom right is a small 'Dc' icon.

Since we're going to deploy a load balancer controller with Helm, we need to define terraform `helm` provider first.

```
terraform/6-helm-provider.tf

1 provider "helm" {
2   kubernetes {
3     host            = data.aws_eks_cluster.default.endpoint
4     cluster_ca_certificate = base64decode(data.aws_eks_cluster.default.certificate_authority[0].data)
5     exec {
6       api_version = "client.authentication.k8s.io/v1beta1"
7       args        = ["eks", "get-token", "--cluster-name", data.aws_eks_cluster.default.id]
8       command    = "aws"
9     }
10   }
11 }
```

Similar to `cluster-autoscaler`, we need to create an IAM role for the load balancer controller with permissions to create and manage AWS load balancers. We're going to deploy it to the same `kube-system` namespace in Kubernetes.

**terraform/7-helm-load-balancer-controller.tf**

```

1 module "aws_load_balancer_controller_irsa_role" {
2   source  = "terraform-aws-modules/iam/aws//modules/iam-role-for-service-accounts-eks"
3   version = "5.3.1"
4
5   role_name = "aws-load-balancer-controller"
6
7   attach_load_balancer_controller_policy = true
8
9   oidc_providers = {
10     ex = {
11       provider_arn          = module.eks.oidc_provider_arn
12       namespace_service_accounts = ["kube-system:aws-load-balancer-controller"]
13     }
14   }
15 }
```

Then the helm release. By default, it creates two replicas, but for the demo, we can use a single one. Then you need to specify the EKS cluster name, Kubernetes service account name and provide annotation to allow this service account to assume the AWS IAM role.

**terraform/7-helm-load-balancer-controller.tf**

```

17 resource "helm_release" "aws_load_balancer_controller" {
18   name = "aws-load-balancer-controller"
19
20   repository = "https://aws.github.io/eks-charts"
21   chart      = "aws-load-balancer-controller"
22   namespace  = "kube-system"
23   version    = "1.4.4"
24
25   set {
26     name  = "replicaCount"
27     value = 1
28   }
29
30   set {
31     name  = "clusterName"
32     value = module.eks.cluster_id
33   }
34
35   set {
36     name  = "serviceAccount.name"
37     value = "aws-load-balancer-controller"
38   }
39
40   set {
41     name  = "serviceAccount.annotations.eks\\.amazonaws\\.com/role-arn"
42     value = module.aws_load_balancer_controller_irsa_role.iam_role_arn
43   }
44 }
```

The load balancer controller uses tags to discover subnets in which it can create load balancers. We also need to update terraform vpc module to include them. It uses an `elb` tag to deploy public load balancers to expose services to the internet and `internal-elb` for the private load balancers to expose services only within your VPC.

**terraform/1-vpc.tf**

```

12 public_subnet_tags = {
13   "kubernetes.io/role/elb" = "1"
14 }
15 private_subnet_tags = {
16   "kubernetes.io/role/internal-elb" = "1"
17 }
```

The last change that we need to make in our EKS cluster is to allow access from the EKS control plane to the `webhook` port of the AWS load balancer controller.

**terraform/2-eks.tf**

```

63  node_security_group_additional_rules = {
64    ingress_allow_access_from_control_plane = {
65      type          = "ingress"
66      protocol     = "tcp"
67      from_port    = 9443
68      to_port      = 9443
69      source_cluster_security_group = true
70      description   = "Allow access from control plane to webhook port of AWS load balancer controller"
71    }
72  }

```

We're done with terraform; now let's apply.

```
terraform init
terraform apply
```

Check if the controller is running.

```
kubectl get pods -n kube-system
```

You can watch logs with the following command.

```
kubectl logs -f -n kube-system \
-l app.kubernetes.io/name=aws-load-balancer-controller
```

To test, let's create an echo server deployment with ingress.

#### k8s/echoserver.yaml

```

1  ---
2  apiVersion: apps/v1
3  kind: Deployment
4  metadata:
5    name: echoserver
6    namespace: default
7  spec:
8    selector:
9      matchLabels:
10     app: echoserver
11  replicas: 1
12  template:
13    metadata:
14      labels:
15        app: echoserver
16    spec:
17      containers:
18        - image: k8s.gcr.io/e2e-test-images/echoserver:2.5
19          name: echoserver
20          ports:
21            - containerPort: 8080
22  ---
23  apiVersion: v1
24  kind: Service
25  metadata:
26    name: echoserver
27    namespace: default
28  spec:
29    ports:
30      - port: 8080
31        protocol: TCP
32        type: ClusterIP
33        selector:
34          app: echoserver
35  ---
36  apiVersion: networking.k8s.io/v1
37  kind: Ingress
38  metadata:
39    name: echoserver
40    namespace: default
41    annotations:
42      alb.ingress.kubernetes.io/scheme: internet-facing
43      alb.ingress.kubernetes.io/target-type: ip
44  spec:
45    ingressClassName: alb

```

```
46   rules:
47     - host: echo.devopsbyexample.io
48       http:
49         paths:
50           - path: /
51             pathType: Exact
52             backend:
53               service:
54                 name: echoserver
55                 port:
56                   number: 8080
```

Then apply the YAML.

```
kubectl apply -f k8s/echoserver.yaml
```

To make ingress work, we need to create a CNAME record. Get the application load balancer DNS name and create a CNAME record in your DNS hosting provider.

```
kubectl get ingress
```

In a few minutes, you can try to access your ingress.

```
curl http://echo.devopsbyexample.io
```

**Custom-built software solutions for exact business needs.**

RabIT Solutions