**How to Create Amazon EKS Cluster Using Terraform**

### Prerequisites

Before you start creating, you’ll need the following:

* an AWS account;
* [identity and access management (IAM) credentials and programmatic access;](https://aws.amazon.com/iam/?gclid=CjwKCAiAopuvBhBCEiwAm8jaMRr63J8flYOSdCcw52cGaELcZ4P16MNpwSnoUhTMHEh20VyNhIWm4RoC8G4QAvD_BwE&trk=858d3377-dc99-4b71-b7d9-dfbd53b3fb6c&sc_channel=ps&ef_id=CjwKCAiAopuvBhBCEiwAm8jaMRr63J8flYOSdCcw52cGaELcZ4P16MNpwSnoUhTMHEh20VyNhIWm4RoC8G4QAvD_BwE:G:s&s_kwcid=AL!4422!3!651612429260!e!!g!!amazon%20iam!19836375022!146902912253)
* [AWS credentials that are set up locally with aws configure;](https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html)
* [a code or text editor, like VS Code](https://code.visualstudio.com/download);
* [install and Set Up kubectl on Windows](https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/).

Once you have finished with the prerequisites, it is time to start writing the code to create an EKS cluster.

****

we’ll explore how to leverage Terraform to automate the creation of Amazon EKS clusters | How to Create Amazon EKS Cluster Using Terraform. enabling efficient deployment and management of Kubernetes workloads on AWS infrastructure.

### **What is Terraform?**

### It is an open-source IaaC (Infrastructure as a code) software tool where you define and create resources using providers in the declarative configuration language example JSON.

With Terraform, You can package and reuse the code in the form of modules.

It supports a number of cloud infrastructure providers such as AWS, Azure, GCP, IBM Cloud, OCI, etc.

### **What is Amazon EKS?**

### Amazon EKS (Elastic Kubernetes Service) is a managed Kubernetes service provided by Amazon Web Services (AWS). It simplifies the process of deploying, managing, and scaling containerized applications using Kubernetes on AWS infrastructure.

With Amazon EKS, users can leverage the power of Kubernetes without the complexity of managing the underlying infrastructure. EKS handles tasks such as cluster provisioning, upgrades, and scaling, allowing developers to focus on building and deploying applications.

Example: Imagine a software development team wants to deploy a microservices-based application using containers. Instead of managing the Kubernetes cluster themselves, they can use Amazon EKS. They define their application’s architecture, specify resource requirements, and deploy it to the EKS cluster. Amazon EKS takes care of provisioning and managing the Kubernetes control plane, worker nodes, networking, and other infrastructure components. This allows the team to focus on developing their application while benefiting from the scalability, reliability, and flexibility of Kubernetes on AWS.

### Steps to set up Amazon EKS Cluster using Terraform

### **Step#1: Create AWS provider.tf file**

Theprovider.tf file in Terraform is a configuration file that specifies the cloud provider and its corresponding plugin that Terraform will use to manage resources in that provider.

*provider.tf*

*provider "aws" {*

*region = "ap-south-1"*

*profile= "default"*

*}*

*terraform {*

*required\_providers {*

*aws = {*

*source = "hashicorp/aws"*

*version = "~> 3.0"*

*}*

*}*

*}*

### **Step#2:Create a virtual private cloud in AWS using Terraform.**

The next step is to create a virtual private cloud in AWS using the aws\_vpc resource. There is one required field that you need to provide, which is the size of your network. 10.0.0.0/16 will give you approximately 65 thousand IP addresses.

*vpc.tf*

*resource "aws\_vpc" "aws-eks-vpc" {*

*cidr\_block = "10.0.0.0/16"*

*tags = {*

*Name = "aws-eks-vpc"*

*}*

*}*

**Step#3 :Create Internet Gateway in AWS using Terraform**

In order to enable internet connectivity for your services, it’s essential to incorporate an internet gateway within your VPC infrastructure. This internet gateway must be attached to the VPC we’ve recently established, serving as the default route for public subnets.

Save this configuration file as **internetgatwy.tf**

internetgatwy.tf

resource "aws\_internet\_gateway" "igw" {

vpc\_id = aws\_vpc.aws-eks-vpc.id

tags = {

Name = "igw"

}

}

**Step#4:Create private and public subnets in AWS using Terraform**

Next, we proceed to create four subnets, adhering to EKS guidelines. We are required to establish two public and two private subnets across distinct availability zones.

Save this subnet configuration file as **subnets.tf**

**subnets.tf**

resource "aws\_subnet" "private-ap-south-1a" {

vpc\_id = aws\_vpc.aws-eks-vpc.id

cidr\_block = "10.0.0.0/19"

availability\_zone = "ap-south-1a"

tags = {

"Name" = "private-ap-south-1a"

"kubernetes.io/role/internal-elb" = "1"

"kubernetes.io/cluster/demo" = "owned"

}

}

resource "aws\_subnet" "private-ap-south-1b" {

vpc\_id = aws\_vpc.aws-eks-vpc.id

cidr\_block = "10.0.32.0/19"

availability\_zone = "ap-south-1b"

tags = {

"Name" = "private-ap-south-1b"

"kubernetes.io/role/internal-elb" = "1"

"kubernetes.io/cluster/demo" = "owned"

}

}

resource "aws\_subnet" "public-ap-south-1a" {

vpc\_id = aws\_vpc.aws-eks-vpc.id

cidr\_block = "10.0.64.0/19"

availability\_zone = "ap-south-1a"

map\_public\_ip\_on\_launch = true

tags = {

"Name" = "public-ap-south-1a"

"kubernetes.io/role/elb" = "1"

"kubernetes.io/cluster/demo" = "owned"

}

}

resource "aws\_subnet" "public-ap-south-1b" {

vpc\_id = aws\_vpc.aws-eks-vpc.id

cidr\_block = "10.0.96.0/19"

availability\_zone = "ap-south-1b"

map\_public\_ip\_on\_launch = true

tags = {

"Name" = "public-ap-south-1b"

"kubernetes.io/role/elb" = "1"

"kubernetes.io/cluster/demo" = "owned"

}

}

**Step#5:Create NAT Gateway in AWS using Terraform**

Let’s now set up a NAT gateway essential for private subnet connectivity to the internet. To begin, we allocate a public IP address. Subsequently, we utilize it within the aws\_nat\_gateway resource, placing it within the designated public subnet, which must have an internet gateway as its default route.

Save this configuration file as natgateway.tf

natgateway.tf

resource "aws\_eip" "nat" {

vpc = true

tags = {

Name = "nat"

}

}

resource "aws\_nat\_gateway" "nat" {

allocation\_id = aws\_eip.nat.id

subnet\_id = aws\_subnet.public-ap-south-1a.id

tags = {

Name = "nat"

}

depends\_on = [aws\_internet\_gateway.igw]

}

**Step#6:Create routing tables and associate subnets in Aws using Terraform**

By now, we have created subnets, internet gateway, and nat gateway. It’s time to create routing tables and associate subnets with them.

Save this configuration file as routes.tf

routes.tf

# 1 : Create a VPC

resource "aws\_vpc" "eksvpc"{

cidr\_block = "10.0.0.0/16"

tags = {

Name = "eksVPC"

}

}

#2: Create a public subnet

resource "aws\_subnet" "PublicSubnet"{

vpc\_id = aws\_vpc.eksvpc.id

availability\_zone = "us-east-1a"

cidr\_block = "10.0.1.0/24"

}

#3 : create a private subnet

resource "aws\_subnet" "PrivSubnet"{

vpc\_id = aws\_vpc.eksvpc.id

cidr\_block = "10.0.2.0/24"

map\_public\_ip\_on\_launch = true

}

#4 : create IGW

resource "aws\_internet\_gateway" "myIgw"{

vpc\_id = aws\_vpc.eksvpc.id

}

#5 : route Tables for public subnet

resource "aws\_route\_table" "PublicRT"{

vpc\_id = aws\_vpc.eksvpc.id

route {

cidr\_block = "0.0.0.0/0"

gateway\_id = aws\_internet\_gateway.myIgw.id

}

}

#7 : route table association public subnet

resource "aws\_route\_table\_association" "PublicRTAssociation"{

subnet\_id = aws\_subnet.PublicSubnet.id

route\_table\_id = aws\_route\_table.PublicRT.id

}

**Step#7:Create Amazon EKS cluster using Terraform.**

Before setting up Amazon EKS clusters, establish an IAM role named “eks.tf” with the AmazonEKSClusterPolicy to enable Kubernetes

clusters to manage AWS resources autonomously.

eks.tf

resource "aws\_iam\_role" "eks" {

name = "eks-cluster-eks"

assume\_role\_policy = <<POLICY

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Principal": {

"Service": "eks.amazonaws.com"

},

"Action": "sts:AssumeRole"

}

]

}

POLICY

}

resource "aws\_iam\_role\_policy\_attachment" "eks-AmazonEKSClusterPolicy" {

policy\_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"

role = aws\_iam\_role.eks.name

}

resource "aws\_eks\_cluster" "eks\_cluster" {

name = "eks\_cluster"

role\_arn = aws\_iam\_role.eks.arn

vpc\_config {

subnet\_ids = [

aws\_subnet.private-ap-south-1a.id,

aws\_subnet.private-ap-south-1b.id,

aws\_subnet.public-ap-south-1a.id,

aws\_subnet.public-ap-south-1b.id

]

}

depends\_on = [aws\_iam\_role\_policy\_attachment.eks-AmazonEKSClusterPolicy]

}

**Step#8:Create a single instance group for Kubernetes**

Next, we’ll establish a single instance group for Kubernetes, which, like the EKS cluster, necessitates an IAM role.

Save this configuration file as nodes.tf

nodes.tf

resource "aws\_iam\_role" "nodes" {

name = "eks-node-group-nodes"

assume\_role\_policy = jsonencode({

Statement = [{

Action = "sts:AssumeRole"

Effect = "Allow"

Principal = {

Service = "ec2.amazonaws.com"

}

}]

Version = "2012-10-17"

})

}

resource "aws\_iam\_role\_policy\_attachment" "nodes-AmazonEKSWorkerNodePolicy" {

policy\_arn = "arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy"

role = aws\_iam\_role.nodes.name

}

resource "aws\_iam\_role\_policy\_attachment" "nodes-AmazonEKS\_CNI\_Policy" {

policy\_arn = "arn:aws:iam::aws:policy/AmazonEKS\_CNI\_Policy"

role = aws\_iam\_role.nodes.name

}

resource "aws\_iam\_role\_policy\_attachment" "nodes-AmazonEC2ContainerRegistryReadOnly" {

policy\_arn = "arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly"

role = aws\_iam\_role.nodes.name

}

resource "aws\_eks\_node\_group" "private-nodes" {

cluster\_name = aws\_eks\_cluster.eks\_cluster.name

node\_group\_name = "private-nodes"

node\_role\_arn = aws\_iam\_role.nodes.arn

subnet\_ids = [

aws\_subnet.private-ap-south-1a.id,

aws\_subnet.private-ap-south-1b.id

]

capacity\_type = "ON\_DEMAND"

instance\_types = ["t2.micro"]

scaling\_config {

desired\_size = 1

max\_size = 5

min\_size = 0

}

update\_config {

max\_unavailable = 1

}

labels = {

role = "general"

}

depends\_on = [

aws\_iam\_role\_policy\_attachment.nodes-AmazonEKSWorkerNodePolicy,

aws\_iam\_role\_policy\_attachment.nodes-AmazonEKS\_CNI\_Policy,

aws\_iam\_role\_policy\_attachment.nodes-AmazonEC2ContainerRegistryReadOnly,

]

}

**Step#9:Create IAM OIDC provider EKS using Terraform.**

To manage permissions for your applications that you deploy in Kubernetes. You can either attach policies to Kubernetes nodes directly. In that case, every pod will get the same access to AWS resources. Or you can create OpenID connect provider, which will allow granting IAM permissions based on the service account used by the pod.

File name is iam-oidc.tf

iam-oidc.tf

data "tls\_certificate" "eks" {

url = aws\_eks\_cluster.eks\_cluster.identity[0].oidc[0].issuer

}

resource "aws\_iam\_openid\_connect\_provider" "eks" {

client\_id\_list = ["sts.amazonaws.com"]

thumbprint\_list = [data.tls\_certificate.eks.certificates[0].sha1\_fingerprint]

url = aws\_eks\_cluster.eks\_cluster.identity[0].oidc[0].issuer

}

**Step#10:Testing the provider first before deploying**

I highly recommend testing the provider first before deploying the autoscaller. It can save you a lot of time.

File name is iam-test.tf

**iam-test.tf**

data "aws\_iam\_policy\_document" "test\_oidc\_assume\_role\_policy" {

statement {

actions = ["sts:AssumeRoleWithWebIdentity"]

effect = "Allow"

condition {

test = "StringEquals"

variable = "${replace(aws\_iam\_openid\_connect\_provider.eks.url, "https://", "")}:sub"

values = ["system:serviceaccount:default:aws-test"]

}

principals {

identifiers = [aws\_iam\_openid\_connect\_provider.eks.arn]

type = "Federated"

}

}

}

resource "aws\_iam\_role" "test\_oidc" {

assume\_role\_policy = data.aws\_iam\_policy\_document.test\_oidc\_assume\_role\_policy.json

name = "test-oidc"

}

resource "aws\_iam\_policy" "test-policy" {

name = "test-policy"

policy = jsonencode({

Statement = [{

Action = [

"s3:ListAllMyBuckets",

"s3:GetBucketLocation"

]

Effect = "Allow"

Resource = "arn:aws:s3:::\*"

}]

Version = "2012-10-17"

})

}

resource "aws\_iam\_role\_policy\_attachment" "test\_attach" {

role = aws\_iam\_role.test\_oidc.name

policy\_arn = aws\_iam\_policy.test-policy.arn

}

output "test\_policy\_arn" {

value = aws\_iam\_role.test\_oidc.arn

}

Step#11:Enhance Your Kubernetes Cluster with Add-ons

Customize and optimize your Kubernetes cluster by incorporating essential add-ons through Terraform.

File name is addons.tf

addons.tf

# Define CoreDNS addon

resource "aws\_eks\_addon" "coredns" {

cluster\_name = aws\_eks\_cluster.eks\_cluster.name

addon\_name = "coredns"

}

# Define kube-proxy addon

resource "aws\_eks\_addon" "kube\_proxy" {

cluster\_name = aws\_eks\_cluster.eks\_cluster.name

addon\_name = "kube-proxy"

}

# Define Amazon VPC CNI addon

resource "aws\_eks\_addon" "vpc\_cni" {

cluster\_name = aws\_eks\_cluster.eks\_cluster.name

addon\_name = "vpc-cni"

}

Step#12:Create AWS EKS Cluster using Terraform

**terraform init**

The terraform init the command is used to initialize a new or existing Terraform configuration. This command downloads the required provider plugins and sets up the backend for storing state.

**terraform plan**

The terraform plan the command is used to create an execution plan for the Terraform configuration. This command shows what resources Terraform will create, modify, or delete when applied.

**terraform apply**

The terraform apply the command is used to apply the Terraform configuration and create or modify resources in the target environment.

To export Kubernetes context you can use aws eks … command; just replace region and name of the cluster:

**aws eks --region ap-south-1 update-kubeconfig --name eks\_cluster**

To check connection to EKS cluster run the following command:

**kubectl get svc**

**Additional to next demo**

### **EKS cluster auto scaling demo**

Next is to create a pod to test IAM roles for service accounts. First, we are going to omit annotations to bind the service account with the role. The way it works, you create a service account and use it in your pod spec. It can be anything, deployment, statefulset, or some jobs. Give it a name k8s/aws-test.yaml.

aws-test.yaml

---

apiVersion: v1

kind: ServiceAccount

metadata:

name: aws-test

namespace: default

---

apiVersion: v1

kind: Pod

metadata:

name: aws-cli

namespace: default

spec:

serviceAccountName: aws-test

containers:

- name: aws-cli

image: amazon/aws-cli

command: [ "/bin/bash", "-c", "--" ]

args: [ "while true; do sleep 30; done;" ]

tolerations:

- operator: Exists

effect: NoSchedule

Then you need to apply it using kubectl apply -f <folder/file> command.

kubectl apply -f k8s/aws-test.yaml

Now, let's check if can list S3 buckets in our account.

kubectl exec aws-cli -- aws s3api list-buckets

Let's add missing annotation to the service account and redeploy the pod. Don't forget to replace 424432388155 with your AWS account number.

aws-test.yaml

---

...

annotations:

eks.amazonaws.com/role-arn: arn:aws:iam::424432388155:role/test-oidc

...

kubectl delete -f k8s/aws-test.yaml

kubectl apply -f k8s/aws-test.yaml

Try to list buckets again.

kubectl exec aws-cli -- aws s3api list-buckets

Create public load balancer on EKS¶

Next, let's deploy the sample application and expose it using public and private load balancers. The first is a deployment object with a base nginx image. File name is k8s/deployment.yaml.

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx

spec:

replicas: 1

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- name: web

containerPort: 80

resources:

requests:

memory: 256Mi

cpu: 250m

limits:

memory: 256Mi

cpu: 250m

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: role

operator: In

values:

- general

# tolerations:

# - key: team

# operator: Equal

# value: devops

# effect: NoSchedule

To expose the application to the internet, you can create a Kubernetes service of a type load balancer and use annotations to configure load balancer properties. By default, Kubernetes will create a load balancer in public subnets, so you don't need to provide any additional configurations. Also, if you want a new network load balancer instead of the old classic load balancer, you can add aws-load-balancer-type equal to nlb. Call it k8s/public-lb.yaml.

public-lb.yaml

---

apiVersion: v1

kind: Service

metadata:

name: public-lb

annotations:

service.beta.kubernetes.io/aws-load-balancer-type: nlb

spec:

type: LoadBalancer

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: web

Create both deployment and the service objects.

kubectl apply -f k8s/deployment.yaml

kubectl apply -f k8s/public-lb.yaml

Find load balancer in AWS console by name. Verify that LB was created in public subnets

Create private load balancer on EKS

Sometimes if you have a large infrastructure with many different services, you have a requirement to expose the application only within your VPC. For that, you can create a private load balancer. To make it private, you need additional annotation: aws-load-balancer-internal and then provide the CIDR range. Usually, you use 0.0.0.0/0 to allow any services within your VPC to access it. Give it a name k8s/private-lb.yaml

private-lb.yaml

---

apiVersion: v1

kind: Service

metadata:

name: private-lb

annotations:

service.beta.kubernetes.io/aws-load-balancer-type: nlb

service.beta.kubernetes.io/aws-load-balancer-internal: 0.0.0.0/0

spec:

type: LoadBalancer

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: web

Let's go back to the terminal and apply it. You can grab the load balancer name and find it in the AWS console as well.

kubectl apply -f k8s/private-lb.yaml

Find load balancer in AWS console by name. Verify that LB was created in private subnets

Deploy EKS cluster autoscaler

Finally, we got to the EKS autoscaller. We will be using OpenID connect provider to create an IAM role and bind it with the autoscaller. Let's create an IAM policy and role first. It's similar to the previous one, but autoscaller will be deployed in the kube-system namespace. File name is terraform/10-iam-autoscaler.tf.

10-iam-autoscaler.tf

data "aws\_iam\_policy\_document" "eks\_cluster\_autoscaler\_assume\_role\_policy" {

statement {

actions = ["sts:AssumeRoleWithWebIdentity"]

effect = "Allow"

condition {

test = "StringEquals"

variable = "${replace(aws\_iam\_openid\_connect\_provider.eks.url, "https://", "")}:sub"

values = ["system:serviceaccount:kube-system:cluster-autoscaler"]

}

principals {

identifiers = [aws\_iam\_openid\_connect\_provider.eks.arn]

type = "Federated"

}

}

}

resource "aws\_iam\_role" "eks\_cluster\_autoscaler" {

assume\_role\_policy = data.aws\_iam\_policy\_document.eks\_cluster\_autoscaler\_assume\_role\_policy.json

name = "eks-cluster-autoscaler"

}

resource "aws\_iam\_policy" "eks\_cluster\_autoscaler" {

name = "eks-cluster-autoscaler"

policy = jsonencode({

Statement = [{

Action = [

"autoscaling:DescribeAutoScalingGroups",

"autoscaling:DescribeAutoScalingInstances",

"autoscaling:DescribeLaunchConfigurations",

"autoscaling:DescribeTags",

"autoscaling:SetDesiredCapacity",

"autoscaling:TerminateInstanceInAutoScalingGroup",

"ec2:DescribeLaunchTemplateVersions"

]

Effect = "Allow"

Resource = "\*"

}]

Version = "2012-10-17"

})

}

resource "aws\_iam\_role\_policy\_attachment" "eks\_cluster\_autoscaler\_attach" {

role = aws\_iam\_role.eks\_cluster\_autoscaler.name

policy\_arn = aws\_iam\_policy.eks\_cluster\_autoscaler.arn

}

output "eks\_cluster\_autoscaler\_arn" {

value = aws\_iam\_role.eks\_cluster\_autoscaler.arn

}

Let's apply the terraform again to create those objects.

terraform apply

Let's create autoscaller itself. You can find the source code for autoscaller here.

Go back to the terminal and apply.

kubectl apply -f k8s/cluster-autoscaler.yaml

You can verify that the autoscaler pod is up and running with the following command.

kubectl get pods -n kube-system

It's a good practice to check logs for any errors.

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

**EKS cluster auto scaling demo**

Verify that AG (aws autoscaling group) has required tags:

k8s.io/cluster-autoscaler/<cluster-name> : owned

k8s.io/cluster-autoscaler/enabled : TRUE

Split the terminal screen. In the first window run:

watch -n 1 -t kubectl get pods

In the second window run:

watch -n 1 -t kubectl get nodes

Now, to trigger autoscaling, increase replica for nginx deployment from 1 to 5.

kubectl apply -f k8s/deployment.yaml

terraform destroy