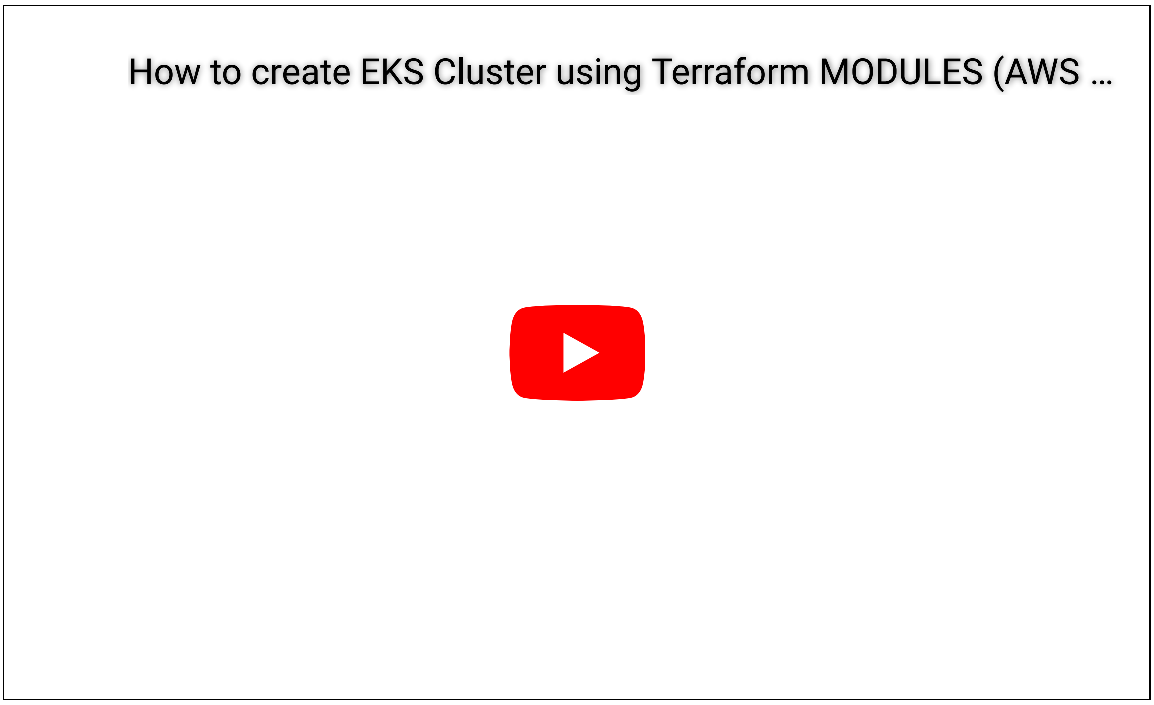
How to create EKS Cluster using Terraform MODULES?



How to create EKS Cluster using Terraform MODULES (AWS ...

- 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 vesion 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

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

terraform {
  required_providers {
    kubectl = {
      source  = "gavinbunney/kubectl"
      version = ">= 1.14.0"
    }
    helm = {
      source  = "hashicorp/helm"
      version = ">= 2.6.0"
    }
  }

  required_version = "~> 1.0"
}
```

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.

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

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.14.3"

  name = "main"
  cidr = "10.0.0.0/16"

  azs             = ["us-east-1a", "us-east-1b"]
  private_subnets = ["10.0.0.0/19", "10.0.32.0/19"]
  public_subnets  = ["10.0.64.0/19", "10.0.96.0/19"]

  enable_nat_gateway     = true
  single_nat_gateway     = true
  one_nat_gateway_per_az = false

  enable_dns_hostnames = true
  enable_dns_support   = true

  tags = {
    Environment = "staging"
  }
}
```

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.

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

```
module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "18.29.0"

  cluster_name    = "my-eks"
  cluster_version = "1.23"

  cluster_endpoint_private_access = true
  cluster_endpoint_public_access  = true

  vpc_id     = module.vpc.vpc_id
  subnet_ids = module.vpc.private_subnets

  enable_irsa = true

  eks_managed_node_group_defaults = {
    disk_size = 50
  }

  eks_managed_node_groups = {
    general = {
      desired_size = 1
      min_size     = 1
      max_size     = 10

      labels = {
        role = "general"
      }

      instance_types = ["t3.small"]
      capacity_type  = "ON_DEMAND"
    }

    spot = {
      desired_size = 1
      min_size     = 1
      max_size     = 10

      labels = {
        role = "spot"
      }

      taints = [{
        key    = "market"
        value  = "spot"
        effect = "NO_SCHEDULE"
      }]

      instance_types = ["t3.micro"]
      capacity_type  = "SPOT"
    }
  }

  tags = {
    Environment = "staging"
  }
}
```

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

```
module "allow_eks_access_iam_policy" {
  source  = "terraform-aws-modules/iam/aws//modules/iam-policy"
  version = "5.3.1"

  name          = "allow-eks-access"
  create_policy = true

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = [
          "eks:DescribeCluster",
        ]
        Effect   = "Allow"
        Resource = "*"
      },
    ]
  })
}
```

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

```
module "eks_admins_iam_role" {
  source  = "terraform-aws-modules/iam/aws//modules/iam-assumable-role"
  version = "5.3.1"

  role_name         = "eks-admin"
  create_role       = true
  role_requires_mfa = false

  custom_role_policy_arns = [module.allow_eks_access_iam_policy.arn]

  trusted_role_arns = [
    "arn:aws:iam::${module.vpc.vpc_owner_id}:root"
  ]
}
```

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

```
module "user1_iam_user" {
  source  = "terraform-aws-modules/iam/aws//modules/iam-user"
  version = "5.3.1"

  name                          = "user1"
  create_iam_access_key         = false
  create_iam_user_login_profile = false

  force_destroy = true
}
```

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

terraform/3-iam.tf

```
module "allow_assume_eks_admins_iam_policy" {
  source  = "terraform-aws-modules/iam/aws//modules/iam-policy"
  version = "5.3.1"

  name          = "allow-assume-eks-admin-iam-role"
  create_policy = true

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = [
          "sts:AssumeRole",
        ]
        Effect   = "Allow"
        Resource = module.eks_admins_iam_role.iam_role_arn
      },
    ]
  })
}
```

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

terraform/3-iam.tf

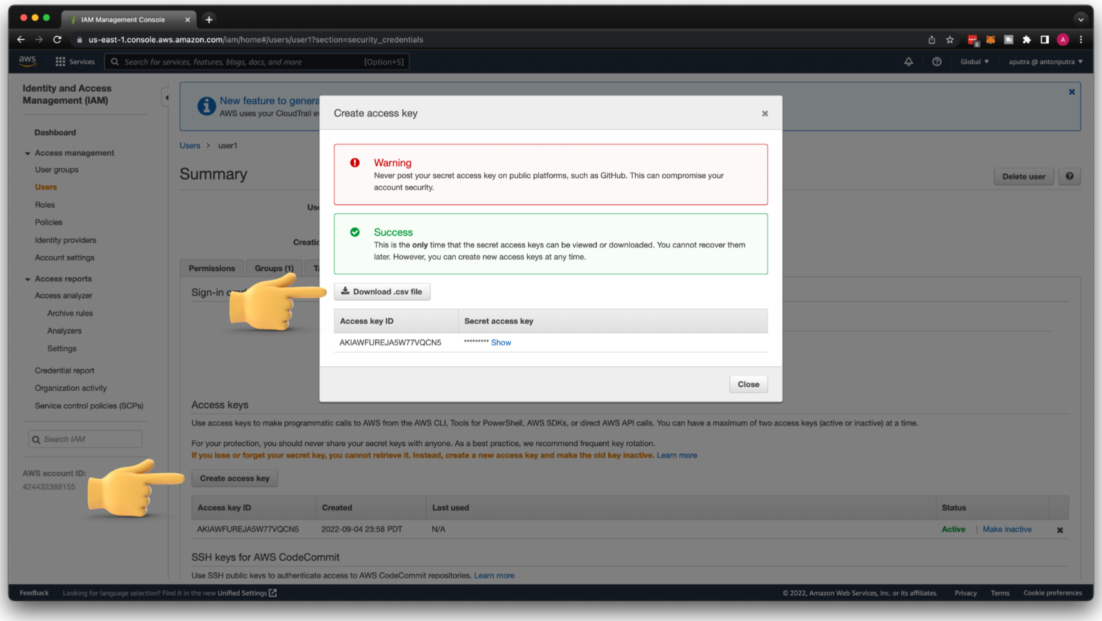
```
module "eks_admins_iam_group" {
  source  = "terraform-aws-modules/iam/aws//modules/iam-group-with-policies"
  version = "5.3.1"

  name                              = "eks-admin"
  attach_iam_self_management_policy = false
  create_group                      = true
  group_users                       = [module.user1_iam_user.iam_user_name]
  custom_group_policy_arns          = [module.allow_assume_eks_admins_iam_policy.arn]
}
```

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.



AWS uses your CloudTrail e

Create access key

Warning
Never post your secret access key on public platforms, such as GitHub. This can compromise your account security.

Success
This is the **only** time that the secret access keys can be viewed or downloaded. You cannot recover them later. However, you can create new access keys at any time.

Access key ID	Secret access key
AKIAWFUREJA5W77VQCN5	********* Show

AWS account ID: 424432388155

Access keys

Use access keys to make programmatic calls to AWS from the AWS CLI, Tools for PowerShell, AWS SDKs, or direct AWS API calls. You can have a maximum of two access keys (active or inactive) at a time.

For your protection, you should never share your secret keys with anyone. As a best practice, we recommend frequent key rotation.
If you lose or forget your secret key, you cannot retrieve it. Instead, create a new access key and make the old key inactive. Learn more

Access key ID	Created	Last used	Status
AKIAWFUREJA5W77VQCN5	2022-09-04 23:58 PDT	N/A	Active | Make inactive

SSH keys for AWS CodeCommit

Use SSH public keys to authenticate access to AWS CodeCommit repositories. Learn more

Looking for language selection? Find it in the new Unified Settings

© 2022, Amazon Web Services, Inc. or its affiliates.

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

```
[profile eks-admin]
role_arn = arn:aws:iam::424432388155:role/eks-admin
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

```
  manage_aws_auth_configmap = true
  aws_auth_roles = [
    {
      rolearn  = module.eks_admins_iam_role.iam_role_arn
      username = module.eks_admins_iam_role.iam_role_name
      groups   = ["system:masters"]
    },
  ]
```

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

```
# https://github.com/terraform-aws-modules/terraform-aws-eks/issues/2009
data "aws_eks_cluster" "default" {
  name = module.eks.cluster_id
}

data "aws_eks_cluster_auth" "default" {
  name = module.eks.cluster_id
}

provider "kubernetes" {
  host                   = data.aws_eks_cluster.default.endpoint
  cluster_ca_certificate =
base64decode(data.aws_eks_cluster.default.certificate_authority[0].data)
  # token                  = data.aws_eks_cluster_auth.default.token

  exec {
    api_version = "client.authentication.k8s.io/v1beta1"
    args        = ["eks", "get-token", "--cluster-name",
data.aws_eks_cluster.default.id]
    command     = "aws"
  }
}
```

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

```
module "cluster_autoscaler_irsa_role" {
  source  = "terraform-aws-modules/iam/aws//modules/iam-role-for-service-accounts-eks"
  version = "5.3.1"

  role_name                        = "cluster-autoscaler"
  attach_cluster_autoscaler_policy = true
  cluster_autoscaler_cluster_ids   = [module.eks.cluster_id]

  oidc_providers = {
    ex = {
      provider_arn               = module.eks.oidc_provider_arn
      namespace_service_accounts = ["kube-system:cluster-autoscaler"]
    }
  }
}
```

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

terraform/5-autoscaler-manifest.tf

```
provider "kubectl" {
  host                   = data.aws_eks_cluster.default.endpoint
  cluster_ca_certificate =
base64decode(data.aws_eks_cluster.default.certificate_authority[0].data)
  load_config_file       = false

  exec {
    api_version = "client.authentication.k8s.io/v1beta1"
    args        = ["eks", "get-token", "--cluster-name",
data.aws_eks_cluster.default.id]
    command     = "aws"
  }
}

resource "kubectl_manifest" "service_account" {
  yaml_body = <<-EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
  name: cluster-autoscaler
  namespace: kube-system
  annotations:
    eks.amazonaws.com/role-arn:
${module.cluster_autoscaler_irsa_role.iam_role_arn}
EOF
}

resource "kubectl_manifest" "role" {
  yaml_body = <<-EOF
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: cluster-autoscaler
  namespace: kube-system
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
rules:
  - apiGroups: [""]
    resources: ["configmaps"]
    verbs: ["create","list","watch"]
  - apiGroups: [""]
    resources: ["configmaps"]
    resourceNames: ["cluster-autoscaler-status", "cluster-autoscaler-priority-expander"]
    verbs: ["delete", "get", "update", "watch"]
EOF
}

resource "kubectl_manifest" "role_binding" {
  yaml_body = <<-EOF
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: cluster-autoscaler
  namespace: kube-system
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: cluster-autoscaler
subjects:
  - kind: ServiceAccount
    name: cluster-autoscaler
    namespace: kube-system
EOF
}

resource "kubectl_manifest" "cluster_role" {
  yaml_body = <<-EOF
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-autoscaler
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
rules:
  - apiGroups: [""]
    resources: ["events", "endpoints"]
    verbs: ["create", "patch"]
  - apiGroups: [""]
    resources: ["pods/eviction"]
    verbs: ["create"]
  - apiGroups: [""]
    resources: ["pods/status"]
    verbs: ["update"]
  - apiGroups: [""]
    resources: ["endpoints"]
    resourceNames: ["cluster-autoscaler"]
    verbs: ["get", "update"]
  - apiGroups: [""]
    resources: ["nodes"]
    verbs: ["watch", "list", "get", "update"]
  - apiGroups: [""]
    resources:
      - "namespaces"
      - "pods"
      - "services"
      - "replicationcontrollers"
      - "persistentvolumeclaims"
      - "persistentvolumes"
    verbs: ["watch", "list", "get"]
  - apiGroups: ["extensions"]
    resources: ["replicasets", "daemonsets"]
    verbs: ["watch", "list", "get"]
  - apiGroups: ["policy"]
    resources: ["poddisruptionbudgets"]
    verbs: ["watch", "list"]
  - apiGroups: ["apps"]
    resources: ["statefulsets", "replicasets", "daemonsets"]
    verbs: ["watch", "list", "get"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["storageclasses", "csinodes", "csidrivers",
"csistoragecapacities"]
    verbs: ["watch", "list", "get"]
  - apiGroups: ["batch", "extensions"]
    resources: ["jobs"]
    verbs: ["get", "list", "watch", "patch"]
  - apiGroups: ["coordination.k8s.io"]
    resources: ["leases"]
    verbs: ["create"]
  - apiGroups: ["coordination.k8s.io"]
    resourceNames: ["cluster-autoscaler"]
    resources: ["leases"]
    verbs: ["get", "update"]
EOF
}

resource "kubectl_manifest" "cluster_role_binding" {
  yaml_body = <<-EOF
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-autoscaler
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-autoscaler
subjects:
  - kind: ServiceAccount
    name: cluster-autoscaler
    namespace: kube-system
EOF
}

resource "kubectl_manifest" "deployment" {
  yaml_body = <<-EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cluster-autoscaler
  namespace: kube-system
  labels:
    app: cluster-autoscaler
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cluster-autoscaler
  template:
    metadata:
      labels:
        app: cluster-autoscaler
    spec:
      priorityClassName: system-cluster-critical
      securityContext:
        runAsNonRoot: true
        runAsUser: 65534
        fsGroup: 65534
      serviceAccountName: cluster-autoscaler
      containers:
        - image: k8s.gcr.io/autoscaling/cluster-autoscaler:v1.23.1
          name: cluster-autoscaler
          resources:
            limits:
              cpu: 100m
              memory: 600Mi
            requests:
              cpu: 100m
              memory: 600Mi
          command:
            - ./cluster-autoscaler
            - --v=4
            - --stderrthreshold=info
            - --cloud-provider=aws
            - --skip-nodes-with-local-storage=false
            - --expander=least-waste
            - --node-group-auto-discovery=asg:tag=k8s.io/cluster-autoscaler/enabled,k8s.io/cluster-autoscaler/${module.eks.cluster_id}
          volumeMounts:
            - name: ssl-certs
              mountPath: /etc/ssl/certs/ca-certificates.crt
              readOnly: true
      volumes:
        - name: ssl-certs
          hostPath:
            path: "/etc/ssl/certs/ca-bundle.crt"
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

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 4
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        resources:
          requests:
            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.

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

```
provider "helm" {
  kubernetes {
    host                   = data.aws_eks_cluster.default.endpoint
    cluster_ca_certificate = base64decode(data.aws_eks_cluster.default.certificate_authority[0].data)
    exec {
      api_version = "client.authentication.k8s.io/v1beta1"
      args        = ["eks", "get-token", "--cluster-name", data.aws_eks_cluster.default.id]
      command     = "aws"
    }
  }
}
```

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

```
module "aws_load_balancer_controller_irsa_role" {
  source  = "terraform-aws-modules/iam/aws//modules/iam-role-for-service-accounts-eks"
  version = "5.3.1"

  role_name = "aws-load-balancer-controller"

  attach_load_balancer_controller_policy = true

  oidc_providers = {
    ex = {
      provider_arn               = module.eks.oidc_provider_arn
      namespace_service_accounts = ["kube-system:aws-load-balancer-controller"]
    }
  }
}
```

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

```
resource "helm_release" "aws_load_balancer_controller" {
  name = "aws-load-balancer-controller"

  repository = "https://aws.github.io/eks-charts"
  chart      = "aws-load-balancer-controller"
  namespace  = "kube-system"
  version    = "1.4.4"

  set {
    name  = "replicaCount"
    value = 1
  }

  set {
    name  = "clusterName"
    value = module.eks.cluster_id
  }

  set {
    name  = "serviceAccount.name"
    value = "aws-load-balancer-controller"
  }

  set {
    name  = "serviceAccount.annotations.eks\\.amazonaws\\.com/role-arn"
    value = module.aws_load_balancer_controller_irsa_role.iam_role_arn
  }
}
```

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

```
  public_subnet_tags = {
    "kubernetes.io/role/elb" = "1"
  }
  private_subnet_tags = {
    "kubernetes.io/role/internal-elb" = "1"
  }
```

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

```
  node_security_group_additional_rules = {
    ingress_allow_access_from_control_plane = {
      type                          = "ingress"
      protocol                      = "tcp"
      from_port                     = 9443
      to_port                       = 9443
      source_cluster_security_group = true
      description                   = "Allow access from control plane to
webhook port of AWS load balancer controller"
    }
  }
```

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

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: echoserver
  namespace: default
spec:
  selector:
    matchLabels:
      app: echoserver
  replicas: 1
  template:
    metadata:
      labels:
        app: echoserver
    spec:
      containers:
      - image: k8s.gcr.io/e2e-test-images/echoserver:2.5
```

```
        name: echoserver
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: echoserver
  namespace: default
spec:
  ports:
  - port: 8080
    protocol: TCP
  type: ClusterIP
  selector:
    app: echoserver
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: echoserver
  namespace: default
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
spec:
  ingressClassName: alb
  rules:
    - host: echo.devopsbyexample.io
      http:
        paths:
          - path: /
            pathType: Exact
            backend:
              service:
                name: echoserver
                port:
                  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
```