Demo Project: Create AWS EKS cluster with a Node Group

This guide walks through setting up an **AWS EKS cluster** manually using the **AWS UI**, including **IAM role setup, VPC creation, EKS cluster configuration, worker node integration, and autoscaling setup**.

Step 1: Create IAM Role for EKS cluster

The **EKS cluster** requires an **IAM Role** that allows AWS to create and manage components on our behalf.

1. Navigate to **AWS Console → IAM → Roles**
2. Click **Create Role**
3. Select **AWS Service** → **EKS** → **EKS - Cluster**
4. Keep the default policy and click **Next**

5. **Role Name**: `eks-cluster-role`
6. Review and click **Create Role**

Step 2: Create VPC with Cloudformation Template for Worker Nodes

EKS requires **specific networking configurations**. Use **AWS CloudFormation** to create a VPC:

1. Navigate to **AWS Console → CloudFormation → Create Stack**
2. Paste the **S3 URL** in the **Specify Template** section:
   - AWS EKS VPC Template (https://docs.aws.amazon.com/eks/latest/userguide/creating-a-vpc.html)
3. **Stack Name**: `eks-worker-node-vpc-stack`
4. Click **Next**, review the summary, and click **Submit**
5. Once complete, navigate to **Outputs** to retrieve VPC details for EKS cluster setup.

Step 3: Create EKS Cluster

1. Navigate to **AWS Console → Elastic Kubernetes Service**
2. Click **Create Cluster**
3. Configure cluster details:
   - **Name**: `eks-cluster-test`
   - **Kubernetes Version**: Latest available
   - **Cluster Service Role**: Select `eks-cluster-role`
   - **VPC**: Choose the VPC from **Step 2**
   - **Security Group**: Choose the SG created with the VPC
   - **Cluster Endpoint Access**: Public and private (default)

- **Add-ons**:
  - Enable `CoreDNS`, `kube-proxy`, `Amazon VPC CNI`
    - `CoreDNS` - Enable service discovery within your cluster. (This will be used instead of IPs)
    - `kube-proxy` - Enable service networking within your cluster.
    - `Amazon VPC CNI` - Enable pod networking within your cluster.

4. Click **Create** (takes ~15 minutes)

Step 4: Connect to EKS cluster locally with kubectl

1. **Update the `kubeconfig` file:**

   `aws eks update-kubeconfig --name eks-cluster-test`

2. **Verify:**
   - `cat .kube/config` you will see:

     `current-context: arn:aws:eks:us-east-1:038462748802:cluster/eks-cluster-test`

   - `kubectl cluster-info` you will see for example: `Kubernetes control plane is running at https://991EF8C8A4E9F995369C2FEB4F162455.gr7.us-east-1.eks.amazonaws.com` (which is the same **API server endpoint shown in the AWS console**)

Step 5: Create EC2 IAM Role for the Node Group

Here we create this role to give EC2 service certain permissions via policies allowing EC2 service and processes running on EC2 service to communicate with other AWS services and to perform certain actions on our behalf.

1. Navigate to **AWS Console → IAM → Roles → Create Role**
2. **Trusted Entity**: AWS Service → **EC2**
3. **Attach Policies:**

- `AmazonEKSWorkerNodePolicy` - (Used for access to EC2 and EKS)
- `AmazonEC2ContainerRegistryReadOnly` - (Used for so container registery can be pulled from there)
- `AmazonEKS_CNI_Policy` - (used for internal communication in Kubernetes so pods on different servers inside the cluster can communicate with each other.)

4. **Role Name**: `eks-node-group-role`
5. Review and click **Create Role**

Step 6: Create Node Group and attach to EKS cluster

Note: Container runtime, Kubelet, and kube-proxy are all installed on these EC2 servers when we create them through the node group.

1. Navigate to **AWS Console → EKS → eks-cluster-test → Compute → Add Node Group**
2. **Name**: `eks-node-group`
3. **Node IAM Role**: Select `eks-node-group-role`
4. Configure EC2 instances:
   - **AMI Type**: Amazon Linux (`AL2_x86_64`)
   - **Instance Type**: `t3.small`
   - **Capacity Type**: On-Demand
5. **Enable Remote Access**
6. Choose an **EC2 Key Pair** (e.g., `docker-server`)
7. Click **Create** (takes ~10 minutes)
8. Verify nodes: `kubectl get nodes`

Step 7: Configure auto-scaling

Need to configure the K8s Autoscaler componet that is running inside the Kubernetes cluster which would work together with the AWS auto scaling group.

Enable Auto Scaling Group

1. Navigate to **AWS Console → EC2 → Auto Scaling Groups**
2. Locate the auto-scaling group created for **eks-node-group**

Create custom policy and attach to Node Group IAM Role

1. Navigate to **AWS Console → IAM → Policies → Create Policy**
2. Use the following **JSON Policy**:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "VisualEditor0",
         "Effect": "Allow",
         "Action": [
            "autoscaling:DescribeAutoScalingInstances",
            "autoscaling:SetDesiredCapacity",
            "autoscaling:DescribeAutoScalingGroups",
            "autoscaling:DescribeTags",
            "autoscaling:DescribeLaunchConfigurations",
            "autoscaling:TerminateInstanceInAutoScalingGroup",
            "ec2:DescribeLaunchTemplateVersions"
         ],
         "Resource": "*"
      }
```

```
    ]
  }
```

3. **Policy Name:** `node-group-autoscale-policy`
4. **Attach this policy to** **`eks-node-group-role:`**
    - Go to AWS IAM Roles → choose `eks-node-group-role`
    - Click on “Add permissions” → “Attach policies”
    - Look for “node-group-autoscale-policy” then click on “Add permissions”

Deploy K8s Autoscaler

1. **Verify Auto Scaling Group Tags**
    - Navigate to **AWS Console → EC2 → Auto Scaling Groups**
    - Select the Auto Scaling Group associated with your Node Group.
    - Scroll down to the **Tags** section and confirm that the following tags exist:
        - `k8s.io/cluster-autoscaler/eks-cluster-test`
        - `k8s.io/cluster-autoscaler/enabled`
    - These tags are **required** for the Kubernetes Cluster Autoscaler to automatically discover and manage the scaling of worker nodes.
2. **Deploy the Cluster Autoscaler**

    Apply the official Cluster Autoscaler manifest:

    `kubectl apply -f https://raw.githubusercontent.com/kubernetes/autoscaler/master/cluster-autoscaler/cloudprovider/aws/examples/cluster-autoscaler-autodiscover.yaml`
3. **Verify deployment:**
    Check if the
    `cluster-autoscaler` deployment is running:

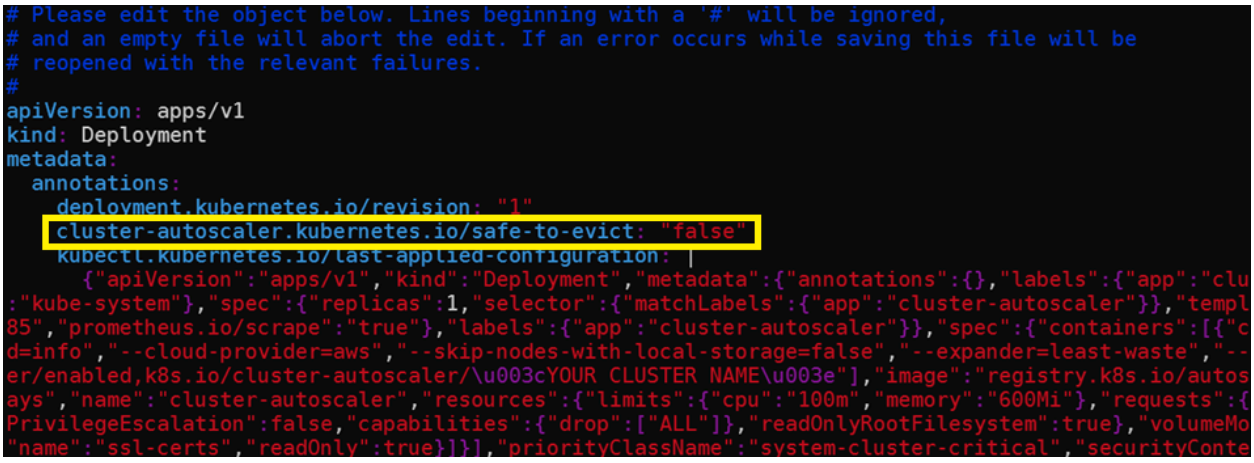
    `kubectl get deployment -n kube-system cluster-autoscaler`
4. **Edit Cluster Autoscaler Deployment**

    Modify the deployment to prevent eviction and optimize scaling behavior:

`kubectl edit deployment -n kube-system cluster-autoscaler`

- **Under** `spec.template.metadata.annotations`, add:

  `cluster-autoscaler.kubernetes.io/safe-to-evict: "false"`

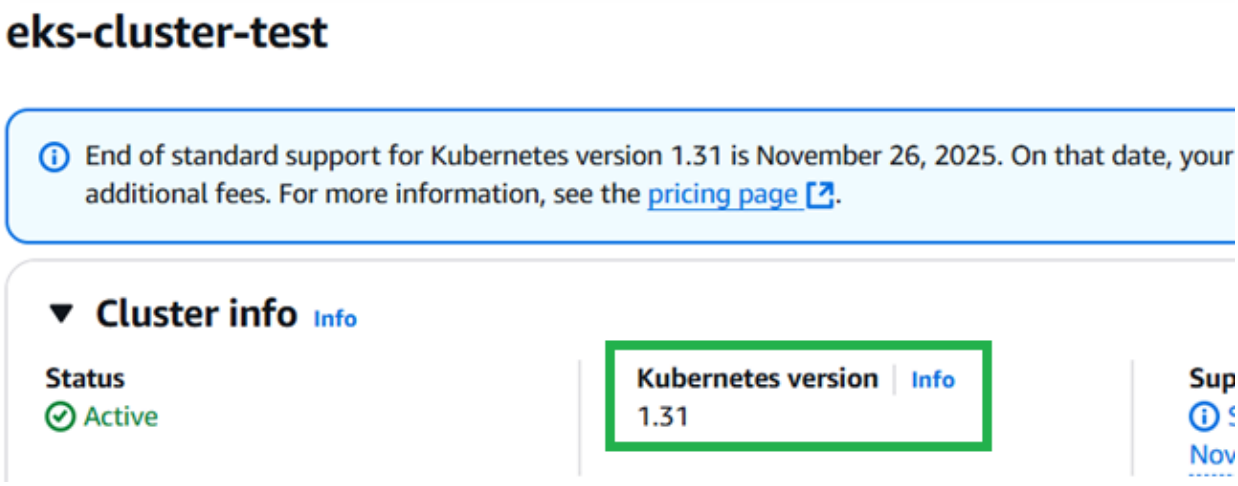


- **Under** `spec.template.spec.containers.args`, add the following:

```
- --balance-similar-node-groups
- --skip-nodes-with-system-pods=false
```

These flags improve node balancing and ensure that system pods are considered during scaling.

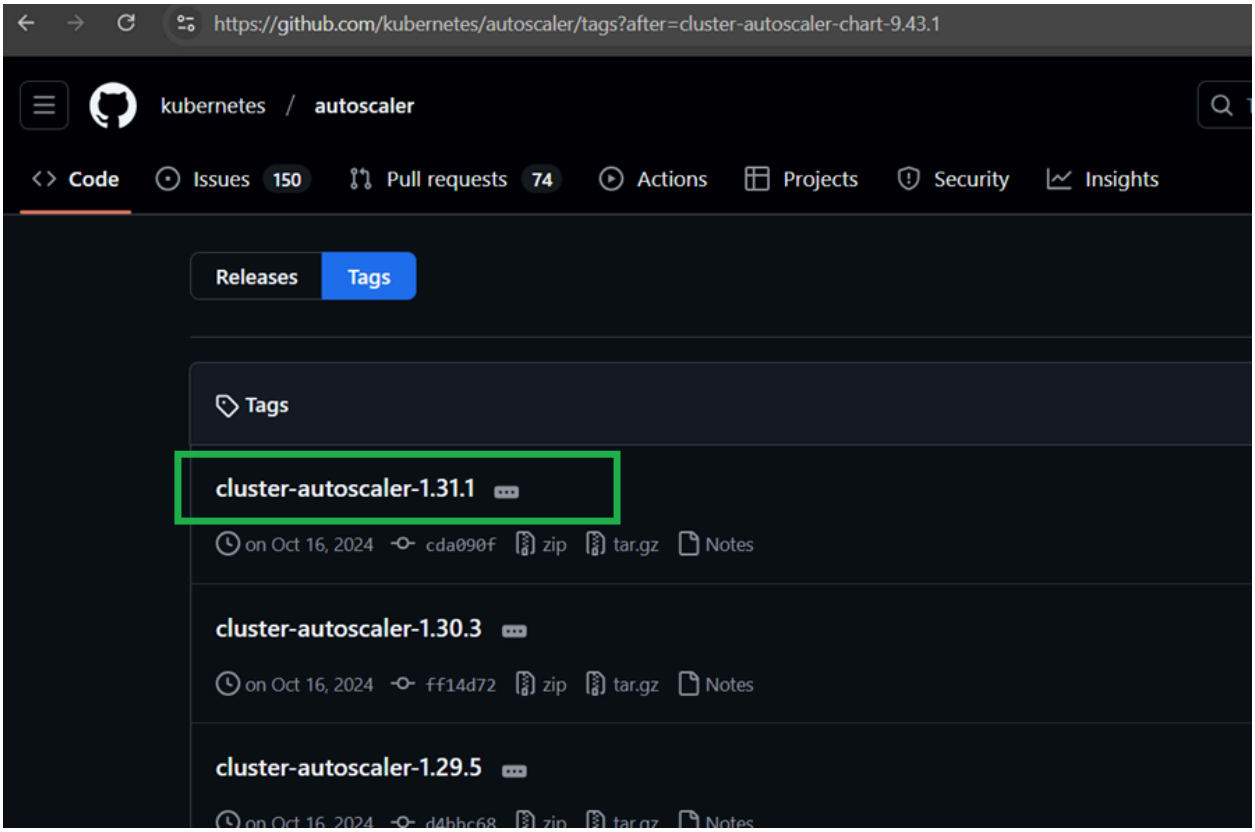
5. **Update Cluster Autoscaler Version**
   - Go to **AWS Console → EKS → Clusters → eks-cluster-test**
   - Check the **Kubernetes Version** of your cluster



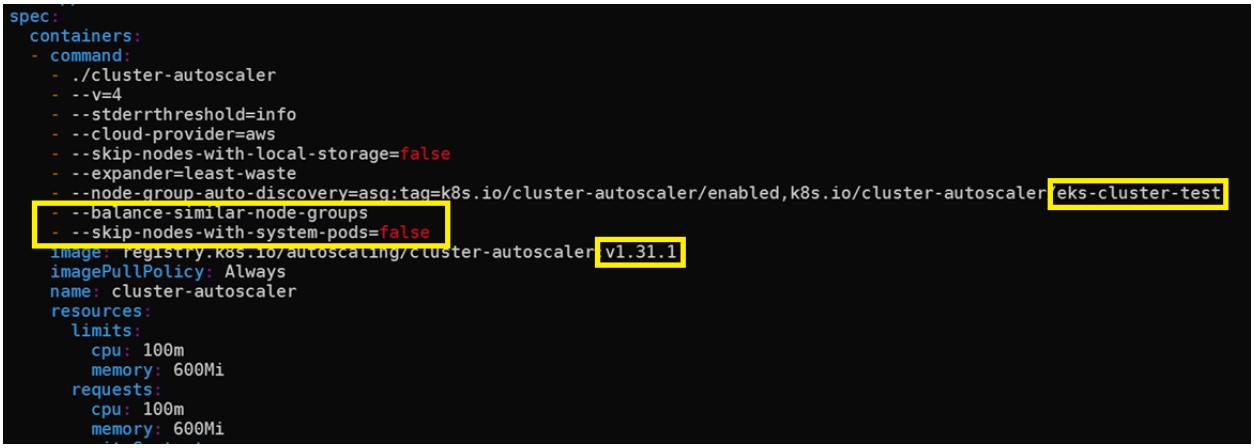
- Find the corresponding **Cluster Autoscaler version** on GitHub: Kubernetes Autoscaler Releases (

  https://github.com/kubernetes/autoscaler/tags)

  Example version mapping:



- Edit the `cluster-autoscaler` deployment and update the container image to match your Kubernetes version.



6. **Verify Cluster Autoscaler** is Running: `kubectl get pod -n kube-system`

Alternative Approach

Instead of modifying the deployment manually in step 4, you can:

- **Download the YAML file locally**
- **Make the necessary changes**
- **Apply the updated YAML file** using `kubectl apply -f <file.yaml>` This ensures that you retain a record of the changes for future reference.

Step 8: Deploy our application to our EKS cluster

- **Create** `nginx-config.yaml` **:**

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

2. **Deploy:** `kubectl apply -f nginx-config.yaml`
3. **Check deployment:** `kubectl get pod`
4. Get **LoadBalancer URL:** `kubectl get svc`

Troubleshoot:

Issue: LoadBalancer Stuck in "Pending" State

When checking if the LoadBalancer was created using `kubectl get svc`, the **EXTERNAL-IP** remains `<pending>` for an extended period:

Example output:

```
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP      10.100.0.1     <none>        443/TCP        45h
nginx        LoadBalancer   10.100.47.9    <pending>     80:30310/TCP   5m20s
```

Step 1: Describe the Service

`kubectl describe svc nginx`

Step 2: Check Logs for the Cluster Autoscaler

If the LoadBalancer is failing due to an autoscaling issue, check the logs:

`kubectl logs cluster-autoscaler-7695c746f8-67hcz -n kube-system`

Observed Error:

If you see an **AccessDenied** error similar to this:

```
  Warning  FailedBuildModel  23s (x5 over 63s)  service  (combined from similar events): Failed build
model due to operation error EC2: CreateSecurityGroup, get identity: get credentials: failed to refresh
cached credentials, operation error STS: AssumeRole, https response error StatusCode: 403, RequestID:
4bbf4359-f902-405d-9b5e-76976a47097c, api error AccessDenied: User: arn:aws:sts::061039771089:assumed-
role/EKSNetworkingChainRole/aws-go-sdk-1738053080102244122 is not authorized to perform: sts:TagSession
on resource: arn:aws:iam::038462748802:role/eks-cluster-role
```

Step 3: Solution

The error indicates that the IAM role for the EKS cluster (`eks-cluster-role`) is missing the required `sts:TagSession` permission.

1. **Go to AWS IAM → Roles → eks-cluster-role**
2. **Edit Trust Relationship**
   - Add the following permission under **"Action"**: `sts:TagSession`
3. **Save the changes** and retry.

Outcome: LoadBalancer Successfully Created:

After fixing the IAM permissions, re-run: `kubectl get svc`

Expected output:

```
NAME         TYPE           CLUSTER-IP    EXTERNAL-IP
             PORT(S)        AGE
kubernetes   ClusterIP      10.100.0.1    <none>
             443/TCP        45h
nginx        LoadBalancer   10.100.47.9   k8s-default-nginx-5bad3e5c8f-eefa1ab3c2bb48e3.elb.us-east-
1.amazonaws.com   80:30310/TCP   8m29s
```

The LoadBalancer now has an external IP and is accessible!