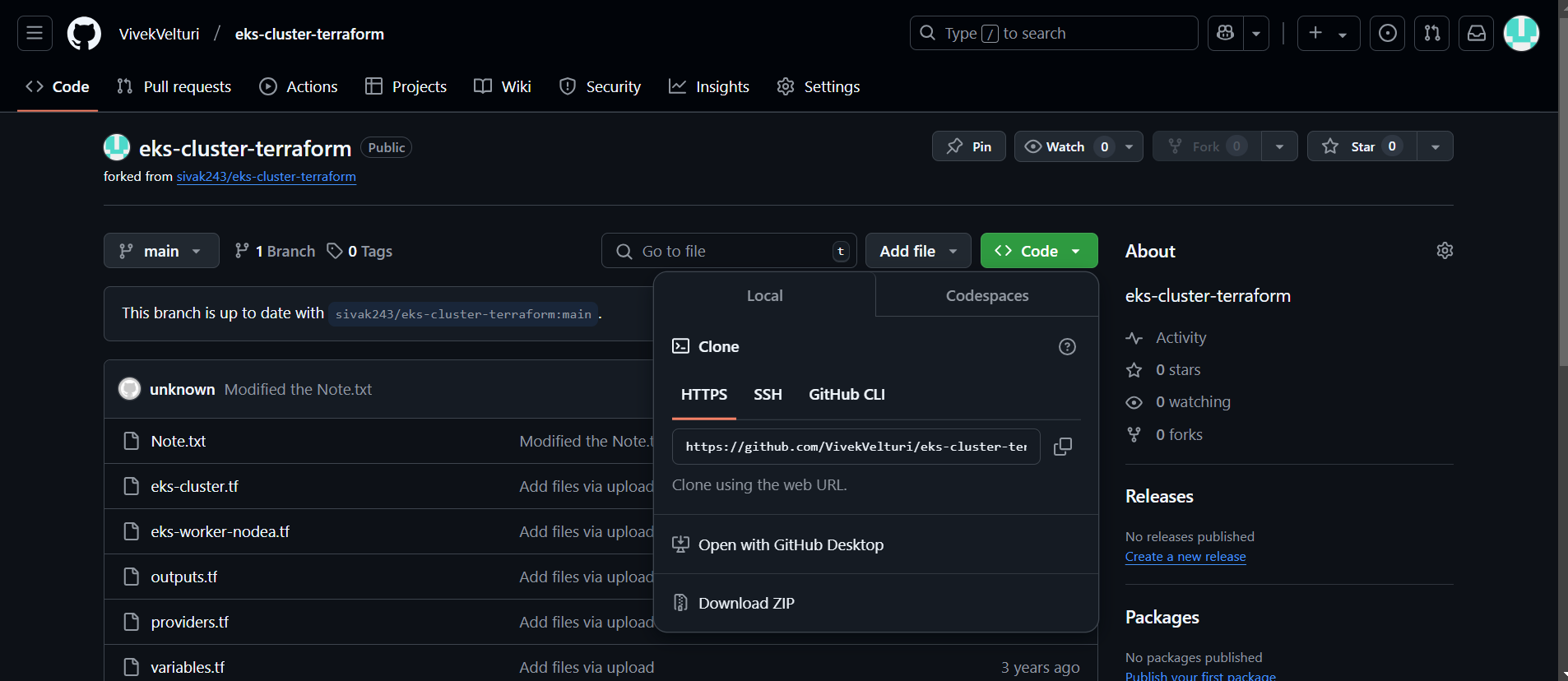
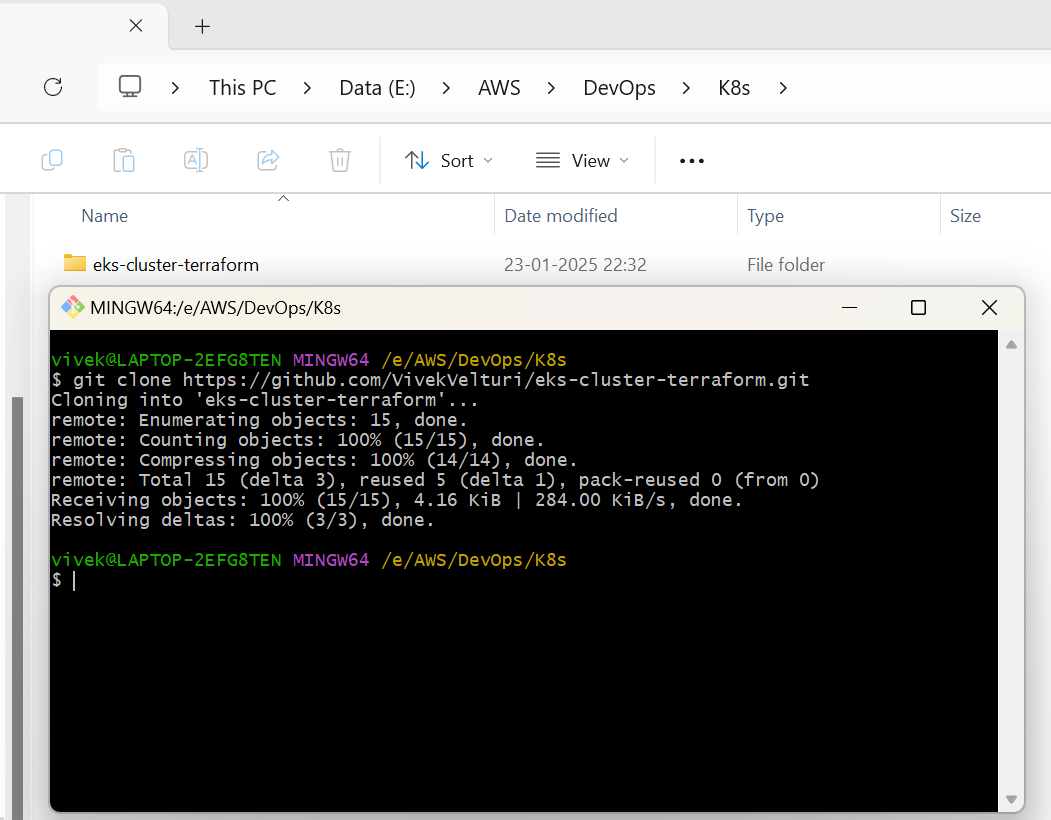
**EKS Cluster Setup and Application Deployment with NodePort Access Using Terraform**

In this Project we will be setting up EKS Cluster using terraform and Deploy Pods with Nodeport Access

* Clone the Terraform scripts from the GitHub.
  + Copy the URL for the Terraform script and clone them into folder in your local computer using the command git clone <URL of the scripts repository> in my case: git clone <https://github.com/VivekVelturi/eks-cluster-terraform.git>





The copied terraform script is as follows:



**eks-cluster.tf**

#

# EKS Cluster Resources

#  \* IAM Role to allow EKS service to manage other AWS services

#  \* EC2 Security Group to allow networking traffic with EKS cluster

#  \* EKS Cluster

#

resource "aws\_iam\_role" "demo-cluster" { # Creates an IAM role for the EKS cluster

  name = "terraform-eks-demo-cluster" # Name of the IAM role

  assume\_role\_policy = <<POLICY

{ # Policy allowing EKS service to assume this IAM role

  "Version": "2012-10-17", # Defines the policy version

  "Statement": [ # List of statements granting permissions

    {

      "Effect": "Allow", # Allows the action

      "Principal": { # The entity that can assume the role

        "Service": "eks.amazonaws.com" # Specifies the EKS service

      },

      "Action": "sts:AssumeRole" # Grants permission to assume this role

    }

  ]

}

POLICY

}

resource "aws\_iam\_role\_policy\_attachment" "demo-cluster-AmazonEKSClusterPolicy" {

  policy\_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy" # Attaches the EKS Cluster Policy

  role       = aws\_iam\_role.demo-cluster.name # Associates the policy with the demo-cluster IAM role

}

resource "aws\_iam\_role\_policy\_attachment" "demo-cluster-AmazonEKSVPCResourceController" {

  policy\_arn = "arn:aws:iam::aws:policy/AmazonEKSVPCResourceController" # Attaches the VPC Resource Controller policy

  role       = aws\_iam\_role.demo-cluster.name # Associates the policy with the IAM role

}

resource "aws\_security\_group" "demo-cluster" {

  name        = "terraform-eks-demo-cluster" # Security group name

  description = "Cluster communication with worker nodes" # Description of the security group

  vpc\_id      = aws\_vpc.demo.id # Associates the security group with a specific VPC

  egress { # Egress rule to allow all outbound traffic

    from\_port   = 0 # Start of port range (0 means all)

    to\_port     = 0 # Start of port range (0 means all)

    protocol    = "-1" # Protocol "-1" allows all protocols

    cidr\_blocks = ["0.0.0.0/0"] # CIDR block for all IPv4 addresses

  }

  tags = { # Tags for identifying the resource

    Name = "terraform-eks-demo" # Name tag for the security group

  }

}

resource "aws\_security\_group\_rule" "demo-cluster-ingress-workstation-https" {

  cidr\_blocks       = [local.workstation-external-cidr] # Allows access from the workstation's external IP range

  description       = "Allow workstation to communicate with the cluster API Server" # Description of the rule

  from\_port         = 443 # Start of port range (443 is used for HTTPS)

  protocol          = "tcp" # Restricts to TCP protocol

  security\_group\_id = aws\_security\_group.demo-cluster.id # Associates the rule with the demo-cluster security group

  to\_port           = 443 # End of port range (443 is used for HTTPS).

  type              = "ingress" # Ingress type means traffic coming into the cluster

}

resource "aws\_eks\_cluster" "demo" {

  name     = var.cluster-name # Name of the cluster, passed as a variable

  role\_arn = aws\_iam\_role.demo-cluster.arn # IAM role to be used by the EKS cluster

  vpc\_config { # VPC networking configuration for the cluster

    security\_group\_ids = [aws\_security\_group.demo-cluster.id] # VPC networking configuration for the cluster

    subnet\_ids         = aws\_subnet.demo[\*].id # Subnets for the cluster (referenced from another resource)

  }

  depends\_on = [ # Ensures these policies are attached before creating the cluster

    aws\_iam\_role\_policy\_attachment.demo-cluster-AmazonEKSClusterPolicy,

    aws\_iam\_role\_policy\_attachment.demo-cluster-AmazonEKSVPCResourceController,

  ]

}

This Terraform configuration:

1. Creates an IAM role with necessary permissions for EKS.
2. Sets up a security group for secure communication.
3. Deploys an EKS cluster in a VPC with the above configurations.

**eks-worker-nodea.tf**

#

# EKS Worker Nodes Resources

#  \* IAM role allowing Kubernetes actions to access other AWS services

#  \* EKS Node Group to launch worker nodes

#

resource "aws\_iam\_role" "demo-node" { # Creates an IAM role for EKS worker nodes

  name = "terraform-eks-demo-node" # Assigns a name to the IAM role

  assume\_role\_policy = <<POLICY # Policy that allows the EC2 service to assume this IAM role

{

  "Version": "2012-10-17", # Policy version

  "Statement": [ # List of policy statements

    {

      "Effect": "Allow", # Allows the specified action

      "Principal": { # The entity that can assume this role

        "Service": "ec2.amazonaws.com" # Specifies EC2 instances

      },

      "Action": "sts:AssumeRole" # Grants permission to assume this role

    }

  ]

}

POLICY

}

resource "aws\_iam\_role\_policy\_attachment" "demo-node-AmazonEKSWorkerNodePolicy" {

  policy\_arn = "arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy" # Attaches the Worker Node Policy to the IAM role

  role       = aws\_iam\_role.demo-node.name # Associates the policy with the demo-node IAM role

}

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

  policy\_arn = "arn:aws:iam::aws:policy/AmazonEKS\_CNI\_Policy" # Attaches the Amazon EKS CNI Policy

  role       = aws\_iam\_role.demo-node.name # Associates the policy with the demo-node IAM role

}

resource "aws\_iam\_role\_policy\_attachment" "demo-node-AmazonEC2ContainerRegistryReadOnly" {

  policy\_arn = "arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly" # Attaches a policy to allow read-only access to ECR

  role       = aws\_iam\_role.demo-node.name # Associates the policy with the demo-node IAM role

}

resource "aws\_eks\_node\_group" "demo" { # Creates an EKS Node Group for worker nodes

  cluster\_name    = aws\_eks\_cluster.demo.name # Specifies the name of the EKS cluster to associate with the node group

  node\_group\_name = "demo" # Assigns a name to the node group

  node\_role\_arn   = aws\_iam\_role.demo-node.arn # Specifies the IAM role to be assumed by the worker nodes

  subnet\_ids      = aws\_subnet.demo[\*].id # Associates the node group with the specified subnets

  scaling\_config { # Configures scaling for the node group

    desired\_size = 1 # Sets the desired number of nodes in the group

    max\_size     = 1 # Sets the maximum number of nodes

    min\_size     = 1 # Sets the minimum number of nodes

  }

  depends\_on = [ # Ensures the IAM policies are attached before creating the node group

    aws\_iam\_role\_policy\_attachment.demo-node-AmazonEKSWorkerNodePolicy,

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

    aws\_iam\_role\_policy\_attachment.demo-node-AmazonEC2ContainerRegistryReadOnly,

  ]

}

This configuration sets up the resources needed for EKS worker nodes:

1. An IAM role (aws\_iam\_role.demo-node) is created, allowing EC2 instances (worker nodes) to assume the role and interact with AWS services.
2. Three managed policies are attached to the role:
   * AmazonEKSWorkerNodePolicy: Grants worker nodes the necessary permissions to interact with the EKS cluster.
   * AmazonEKS\_CNI\_Policy: Enables the nodes to manage network interfaces required for Kubernetes pods.
   * AmazonEC2ContainerRegistryReadOnly: Provides read-only access to the Amazon Elastic Container Registry (ECR) for pulling container images.
3. An EKS Node Group (aws\_eks\_node\_group.demo) is created to launch and manage worker nodes. It uses the IAM role and is associated with specific subnets. The scaling configuration is set to maintain exactly one worker node in the group. The depends\_on block ensures that the IAM role and its policies are attached before creating the node group, avoiding dependency issues.

**outputs.tf**

#

# Outputs

#

locals { # Defines reusable local variables for the Terraform configuration

  config\_map\_aws\_auth = <<CONFIGMAPAWSAUTH # Creates a multi-line string for the "aws-auth" ConfigMap in Kubernetes

apiVersion: v1 # Specifies the Kubernetes API version used for the ConfigMap

kind: ConfigMap # Declares the resource type as ConfigMap

metadata: # Metadata section for identifying the resource

  name: aws-auth # Assigns the name "aws-auth" to the ConfigMap

  namespace: kube-system # Places the ConfigMap in the "kube-system" namespace

data: # Contains the actual data for the ConfigMap

  mapRoles: | # Maps IAM roles to Kubernetes users and groups

    - rolearn: ${aws\_iam\_role.demo-node.arn} # References the ARN of the IAM role for worker nodes

      username: system:node:{{EC2PrivateDNSName}} # Sets the username format for the worker nodes

      groups: # Assigns worker nodes to Kubernetes groups

        - system:bootstrappers # Group responsible for bootstrapping nodes

        - system:nodes # Group for nodes in the Kubernetes cluster

CONFIGMAPAWSAUTH # End of the multi-line string for the ConfigMap

  kubeconfig = <<KUBECONFIG # Creates a multi-line string for the Kubernetes kubeconfig file

apiVersion: v1 # Specifies the Kubernetes API version used for the kubeconfig

clusters: # List of Kubernetes clusters the kubeconfig connects to

- cluster: # Cluster definition block

    server: ${aws\_eks\_cluster.demo.endpoint} # References the API server endpoint for the EKS cluster

    certificate-authority-data: ${aws\_eks\_cluster.demo.certificate\_authority[0].data} # Specifies the certificate for secure communication.

  name: Kubernetes # Assigns a name to the cluster

contexts: # List of contexts defining how to connect to the cluster

- context: # Context definition block

    cluster: Kubernetes # Specifies the cluster name for the context

    user: aws # Specifies the user for authentication

  name: aws # Assigns a name to the context

current-context: aws # Sets the current context to "aws"

kind: Config # Declares the resource type as Config

preferences: {} # Placeholder for user preferences (empty in this case)

users: # List of users for the kubeconfig

- name: aws # Assigns the name "aws" to the user

  user: # User authentication method

    exec: # Executes an external command for authentication

      apiVersion: client.authentication.k8s.io/v1beta1 # API version for the authentication command

      command: aws-iam-authenticator # Specifies the authentication tool

      args: # List of arguments for the authenticator command

        - "token" # Generates a token for authentication

        - "-i" # Specifies the cluster name as an input parameter

        - "${var.cluster-name}" # References the cluster name variable

KUBECONFIG # End of the multi-line string for the kubeconfig

}

output "config\_map\_aws\_auth" { # Declares an output for the "aws-auth" ConfigMap

  value = local.config\_map\_aws\_auth # Outputs the content of the aws-auth ConfigMap template

}

output "kubeconfig" { # Declares an output for the kubeconfig file

  value = local.kubeconfig # Outputs the content of the kubeconfig template

}

This configuration generates and outputs two essential resources for the EKS cluster:

1. **config\_map\_aws\_auth**:
   * A Kubernetes ConfigMap named aws-auth is generated. It maps the worker node IAM role (demo-node) to Kubernetes users and groups, allowing the nodes to join the cluster and perform their functions.
   * The system:bootstrappers and system:nodes groups give worker nodes the required permissions in Kubernetes.
2. **kubeconfig**:
   * A kubeconfig file is generated, which contains the necessary configuration to connect to the EKS cluster. It includes the API server endpoint, the certificate for secure communication, and the authentication method using aws-iam-authenticator.

By outputting these configurations, users can easily deploy them to their Kubernetes cluster and access the cluster securely from their local machine or other tools.

**vpc.tf**

#

# VPC Resources

#  \* VPC

#  \* Subnets

#  \* Internet Gateway

#  \* Route Table

#

resource "aws\_vpc" "demo" { # Defines an AWS Virtual Private Cloud (VPC) resource

  cidr\_block = "10.0.0.0/16" # Specifies the CIDR block for the VPC, which provides a large address space (`10.0.0.0/16`)

  tags = tomap({ # Tags for the VPC, for organizational and identification purposes

    "Name"                                      = "terraform-eks-demo-node",

    "kubernetes.io/cluster/${var.cluster-name}" = "shared",

  })

}

resource "aws\_subnet" "demo" { # Defines an AWS Subnet resource

  count = 2 # Creates two subnets (subnet1 and subnet2)

  availability\_zone       = data.aws\_availability\_zones.available.names[count.index] # Fetches the availability zone names in a loop using count.index.

  cidr\_block              = "10.0.${count.index}.0/24" # Generates CIDR blocks `10.0.0.0/24` and `10.0.1.0/24`

  map\_public\_ip\_on\_launch = true # Maps public IPs to the subnet upon launch (enables internet access)

  vpc\_id                  = aws\_vpc.demo.id # Associates the subnet with the previously created VPC (id)

  tags = tomap({ # Tags applied to each subnet for organization and identification

    "Name"                                      = "terraform-eks-demo-node",

    "kubernetes.io/cluster/${var.cluster-name}" = "shared",

  })

}

resource "aws\_internet\_gateway" "demo" { # Defines an Internet Gateway (IGW) for the VPC

  vpc\_id = aws\_vpc.demo.id # Attaches the internet gateway to the VPC (id)

  tags = { # Tags to help identify the internet gateway

    Name = "terraform-eks-demo"

  }

}

resource "aws\_route\_table" "demo" { # Defines an AWS Route Table

  vpc\_id = aws\_vpc.demo.id # The route table is associated with the VPC created earlier

  route { # Creates a route within the route table

    cidr\_block = "0.0.0.0/0" # Default route for all traffic

    gateway\_id = aws\_internet\_gateway.demo.id # Route traffic to the internet via the internet gateway

  }

}

resource "aws\_route\_table\_association" "demo" { # Associates the created route table with subnets

  count = 2 # Associates the route table with two subnets (subnet1 and subnet2)

  subnet\_id      = aws\_subnet.demo.\*.id[count.index] # Iterates over both subnets

  route\_table\_id = aws\_route\_table.demo.id # Associates the subnets with the route table created earlier

}

This set of Terraform resources builds the foundational networking setup for an AWS cluster:

1. **aws\_vpc.demo**: Creates a VPC with a large CIDR block (10.0.0.0/16) and applies organizational tags.
2. **aws\_subnet.demo**: Creates two subnets (10.0.0.0/24 and 10.0.1.0/24), each in separate availability zones, with public IPs mapped at launch.
3. **aws\_internet\_gateway.demo**: Attaches an internet gateway to the VPC to allow internet access.
4. **aws\_route\_table.demo**: Creates a route table for the VPC, routing all traffic to the internet via the internet gateway.
5. **aws\_route\_table\_association.demo**: Associates the created route table with both subnets to ensure proper internet connectivity.

This configuration ensures that instances deployed within the VPC have reliable connectivity to the internet, enabling communication with other AWS services or external systems.

**providers.tf**

terraform { # The block to define Terraform settings

  required\_version = ">= 0.12" # Ensures that the Terraform version used is 0.12 or higher

}

provider "aws" { # Specifies the AWS provider block for interacting with AWS services

  region = var.aws\_region # Sets the AWS region using a variable (`var.aws\_region`) for flexibility

}

data "aws\_availability\_zones" "available" {} # Retrieves the list of available AWS availability zones in the chosen region

# Not required: currently used in conjunction with using

# icanhazip.com to determine local workstation external IP

# to open EC2 Security Group access to the Kubernetes cluster.

# See workstation-external-ip.tf for additional information.

provider "http" {} # Enables the HTTP provider for making HTTP requests

This configuration sets up the foundational elements for Terraform to interact with AWS and other services:

1. **Terraform Block**: Ensures the Terraform version is 0.12 or higher, ensuring compatibility with modern features.
2. **AWS Provider**: Configures the AWS provider to operate in a specified region, which is dynamically set using the var.aws\_region variable.
3. **Data Source (aws\_availability\_zones)**: Retrieves all available AWS availability zones in the specified region, useful for resource placement and high availability.
4. **HTTP Provider**: Although not directly used in this code, it enables the ability to perform HTTP requests, which can be leveraged by other scripts (e.g., to fetch the local IP for dynamic security configurations).

This setup provides the groundwork for deploying infrastructure on AWS while also enabling advanced configurations such as dynamic Security Group rules based on external IP addresses.

**variables.tf**

variable "aws\_region" { # Declares a variable named `aws\_region`

  default = "us-west-2" # Sets the default value of the AWS region to `us-west-2`

}

variable "cluster-name" { # Declares a variable named `cluster-name`

  default = "terraform-eks-demo" # Sets the default name for the EKS cluster to `terraform-eks-demo`

  type    = string # Explicitly defines the type of the variable as a string

}

This code defines two variables to make the Terraform configuration flexible and reusable:

1. **aws\_region**: Sets the AWS region for the infrastructure deployment. The default region is us-west-2 (Oregon), but it can be overridden as needed.
2. **cluster-name**: Specifies the name of the EKS cluster to be deployed. The default value is terraform-eks-demo, but users can customize it.

By using these variables, the Terraform code becomes dynamic, allowing changes to key configurations (region and cluster name) without modifying the main code. This approach promotes better maintainability and reuse of the Terraform scripts.

**workstaton-external-ip.tf**

#

# Workstation External IP

#

# This configuration is not required and is

# only provided as an example to easily fetch

# the external IP of your local workstation to

# configure inbound EC2 Security Group access

# to the Kubernetes cluster.

#

data "http" "workstation-external-ip" { # Defines a data source to make an HTTP GET request

  url = "http://ipv4.icanhazip.com" # The URL returns the external IP address of the caller

}

# Override with variable or hardcoded value if necessary

locals {

  workstation-external-cidr = "${chomp(data.http.workstation-external-ip.body)}/32"

}

This code dynamically fetches the external IP address of your local workstation using the public API http://ipv4.icanhazip.com and formats it in CIDR notation (<IP>/32). The local variable workstation-external-cidr can be used elsewhere in your Terraform configuration, such as defining inbound Security Group rules to allow access only from your workstation.

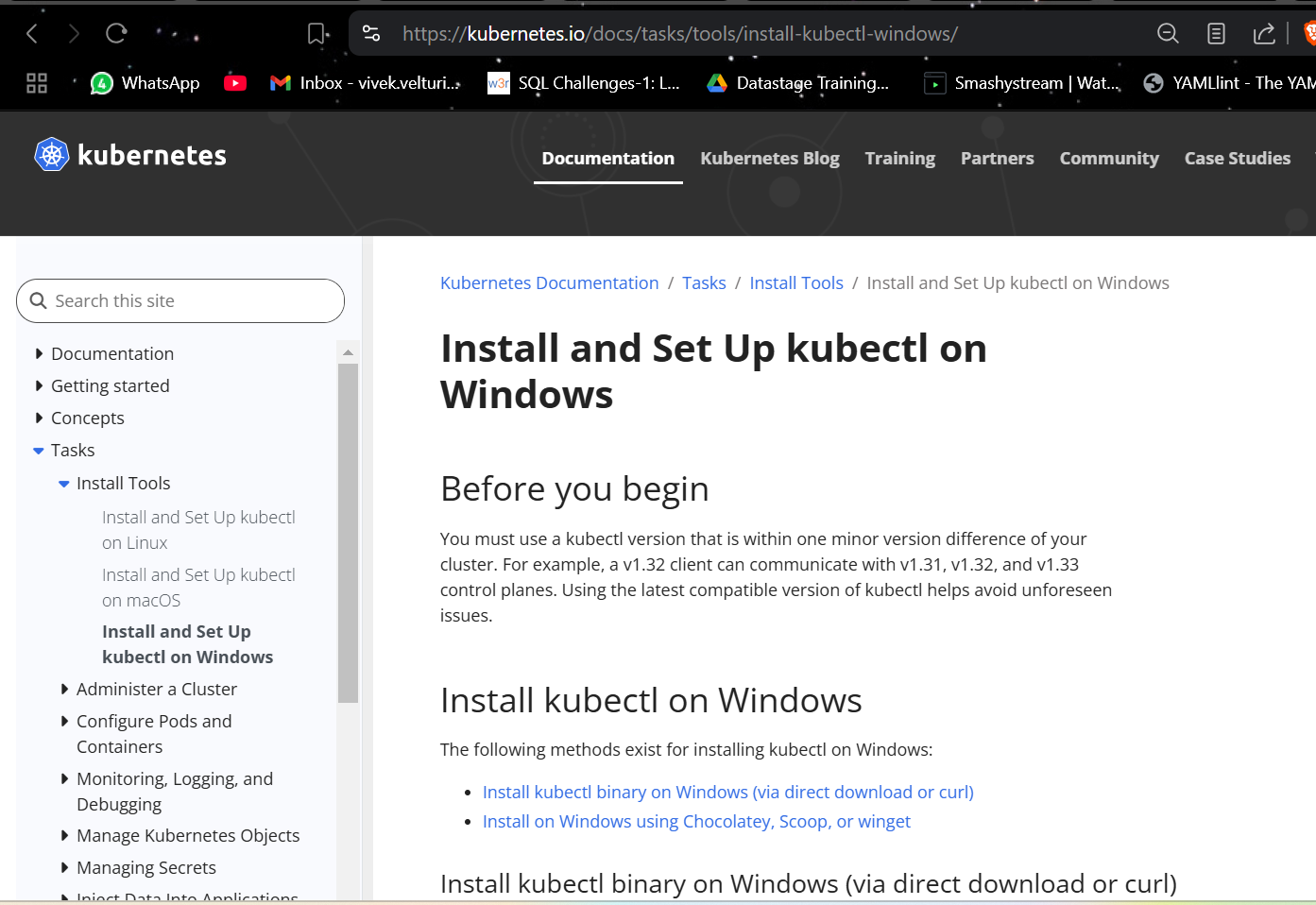
Key points:

1. **data.http.workstation-external-ip**: Makes an HTTP request to fetch the workstation's public IP.
2. **local.workstation-external-cidr**: Formats the IP with /32 to define a network allowing access for only that single IP.

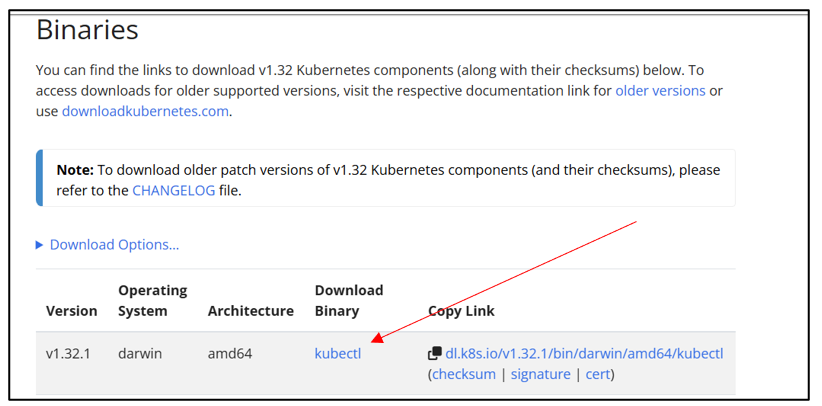
This configuration is optional and useful for securely restricting access to resources from your local machine.

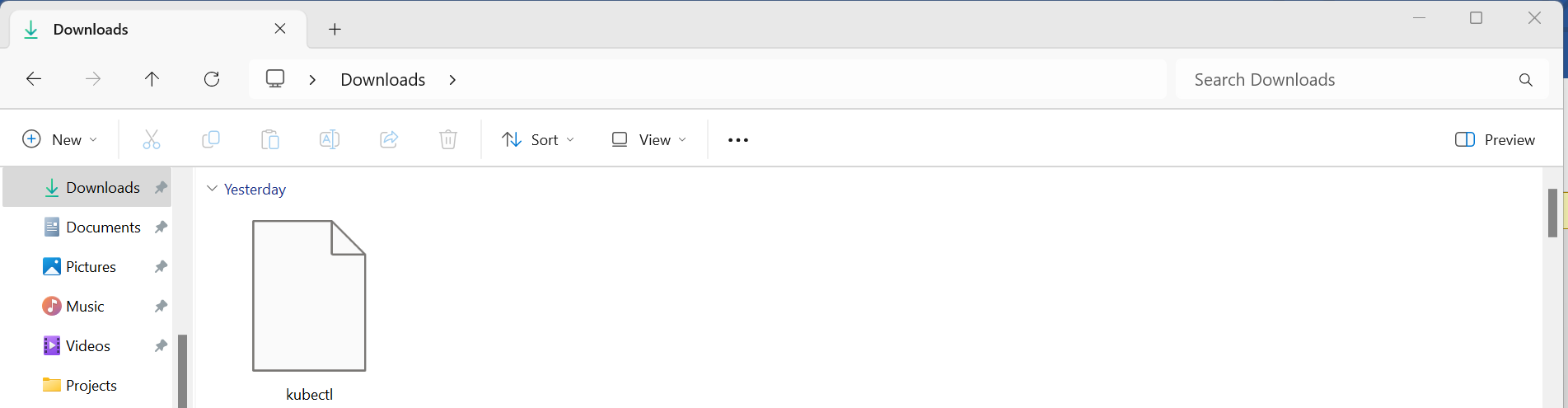
Before we can start working with the EKS cluster created using the Terraform script, we need a tool called kubectl installed on our local computer. Kubectl is a command-line utility that helps us manage and interact with Kubernetes clusters, like the EKS cluster. Since your computer runs on Windows, you'll first need to install kubectl on it. Once installed, you can use it to run commands and communicate with the cluster easily.

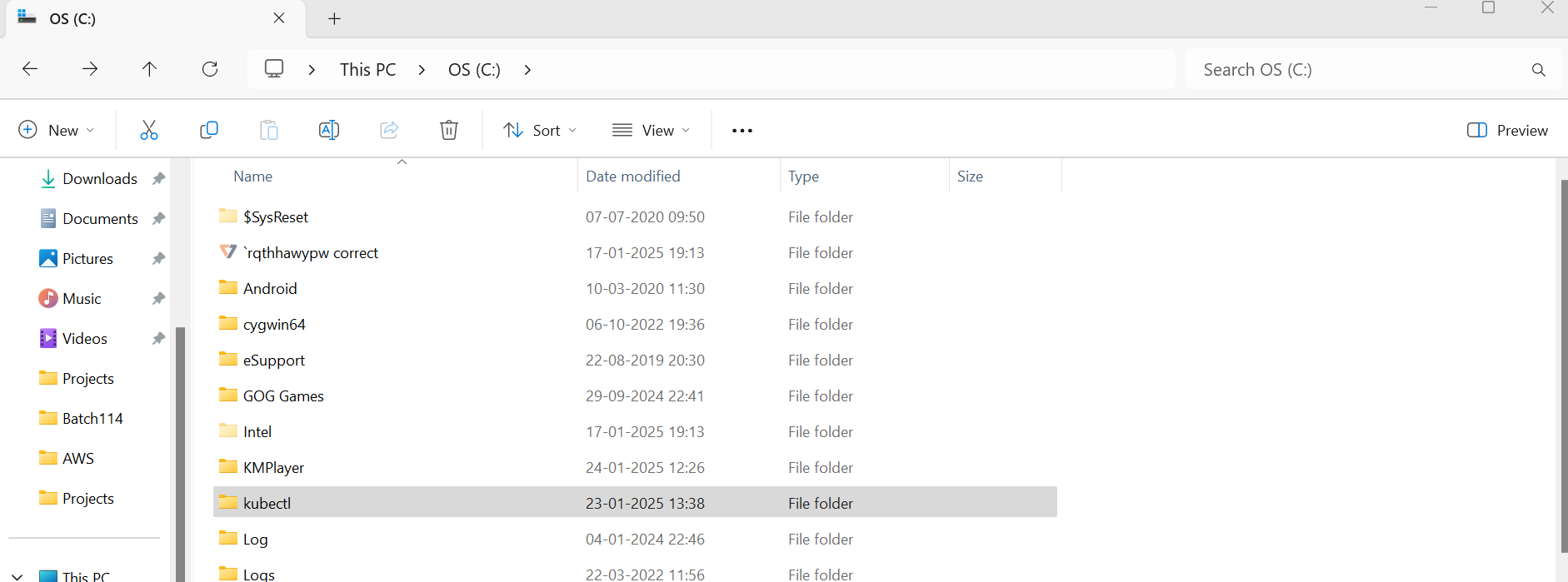
* To install kubectl on a Windows system, visit the official Kubernetes website at:  
  [**https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/**](https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/)
* **Under the "Install kubectl on Windows" section, click on "Install kubectl binary on Windows (via direct download or curl)".**
* Select the latest version of kubectl for Windows (amd64) and click to download it.
* After downloading the kubectl binary, locate the file in the Downloads folder on your local computer. Copy the file, navigate to the C: Drive, and create a new folder named "kubectl." Paste the copied file into this folder.
* Now go to **Computer Properties**, then click on **Advanced system settings**. In the System Properties window, select the **Environment Variables** button. In the Environment Variables window, find the **Path** variable under System variables, **click Edit, and add C:\kubectl** to the list.
* Now, we have successfully configured kubectl on our local Windows OS computer.

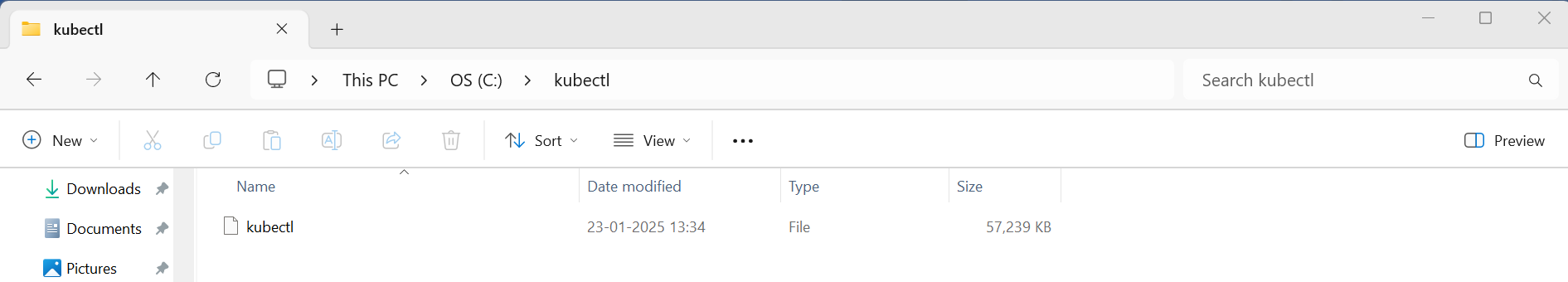


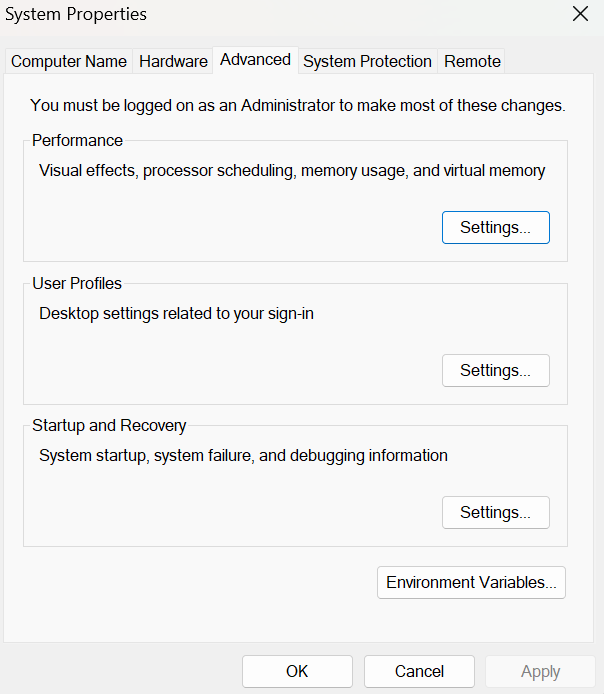


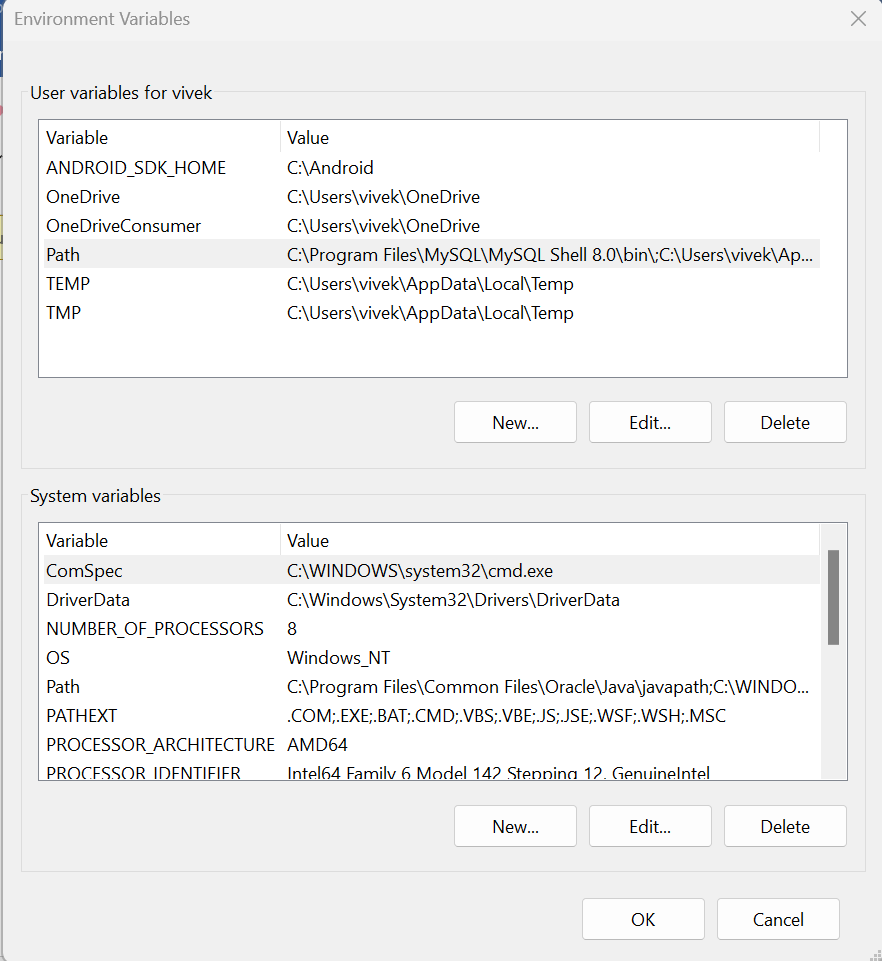


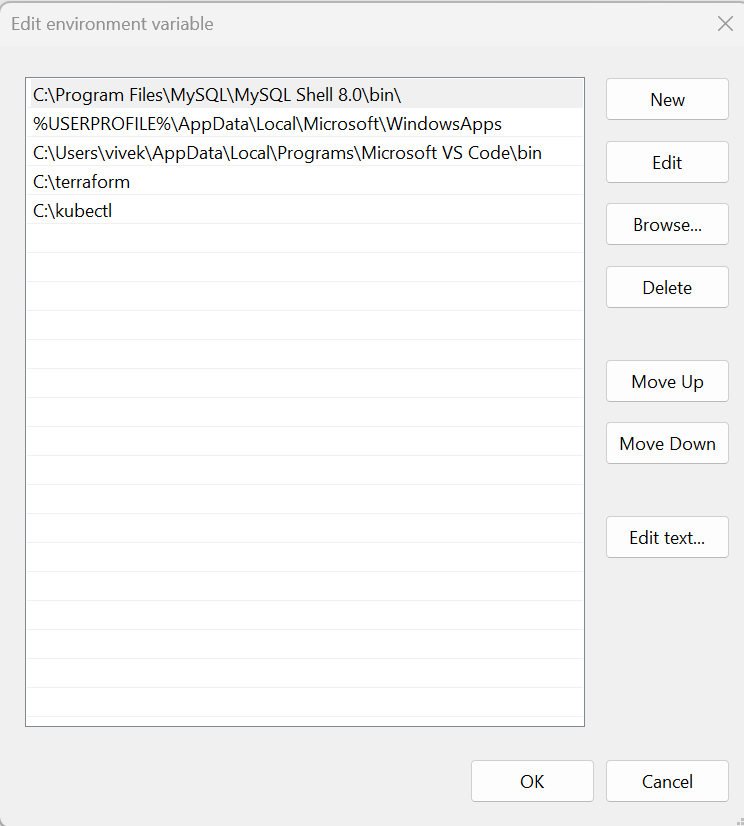




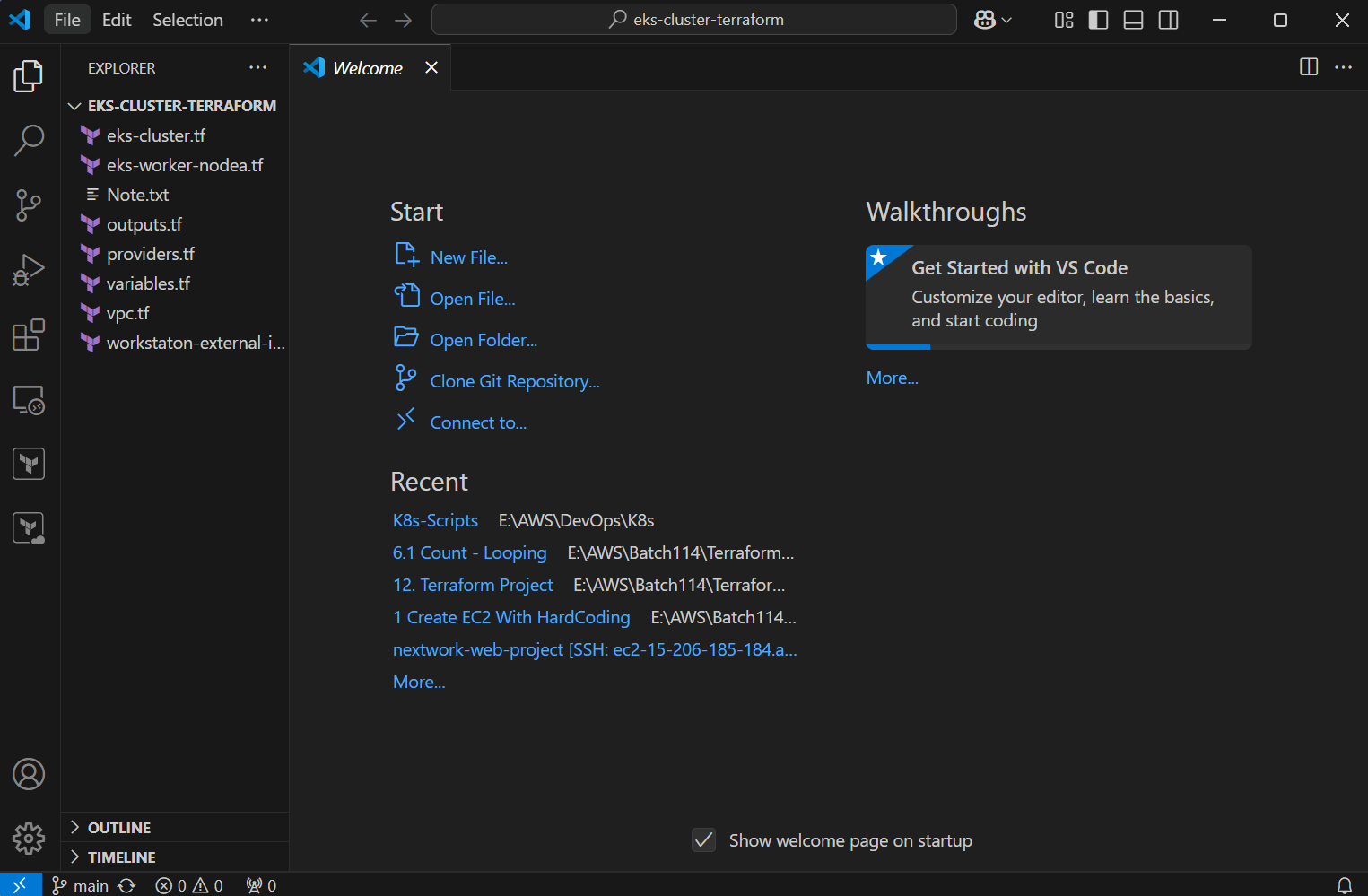




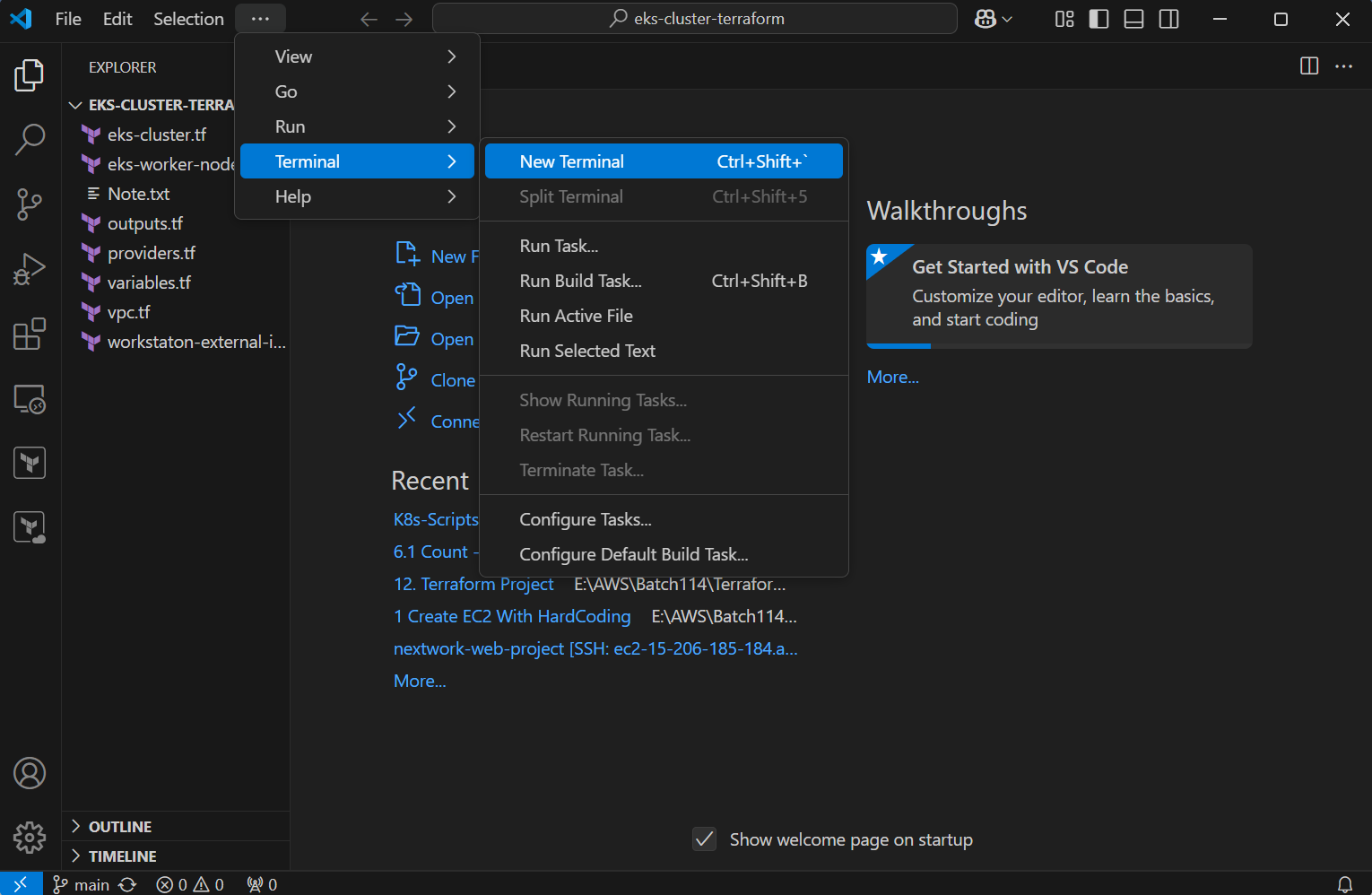




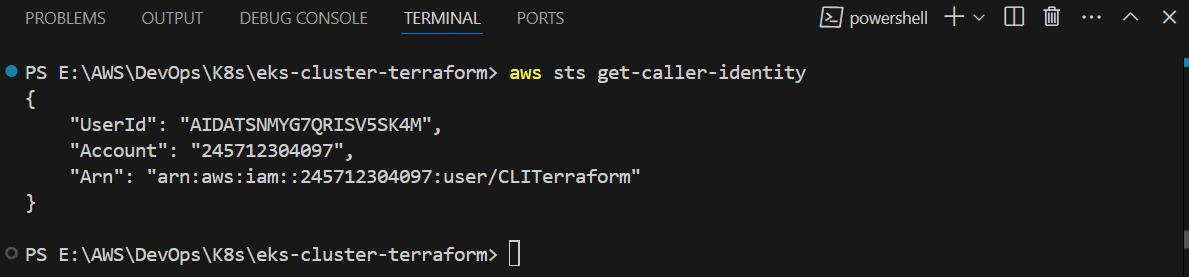
* Now that kubectl is set up on our computer, let's open Visual Studio Code and go to the folder where we cloned the Terraform scripts from GitHub.



* Open a new terminal.



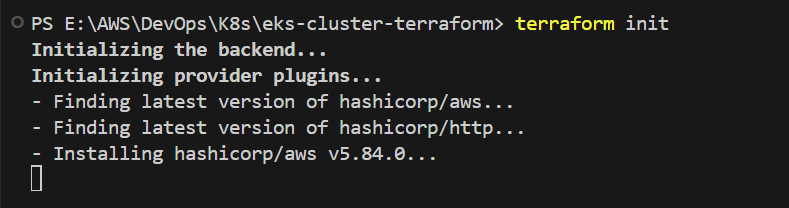
* To check if Visual Studio is connected to our AWS account, run the command: aws sts get-caller-identity.

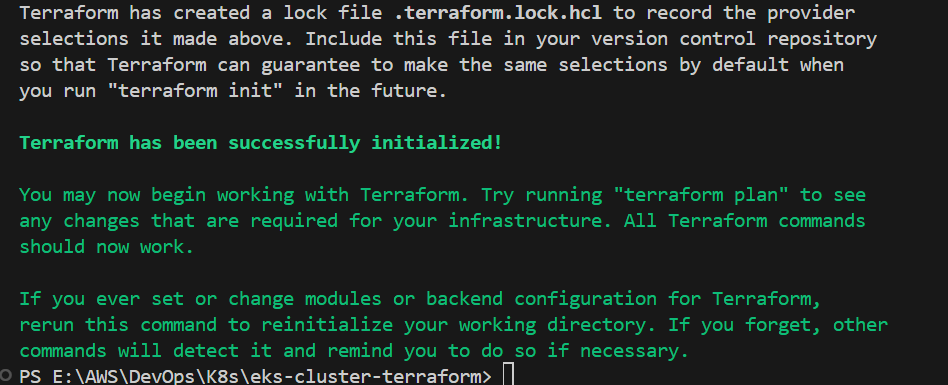


**Note:** **If Visual Studio is not connected to AWS**, follow these steps:

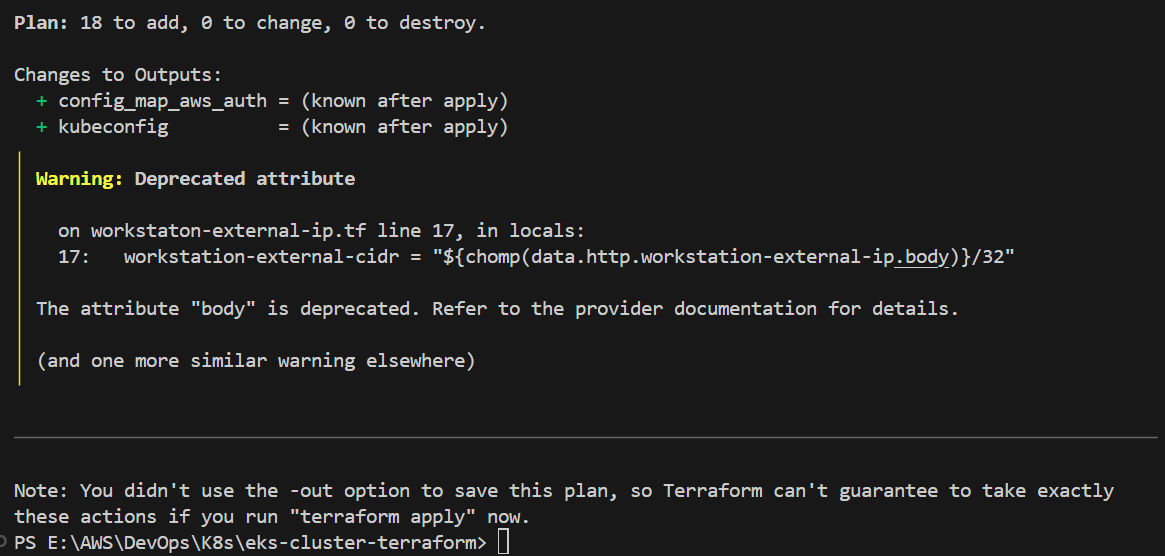
**Verify AWS CLI Configuration**: Ensure the AWS CLI is installed and configured properly. You can configure it by running the command aws configure in your command prompt or terminal, and then **enter**ing **your AWS Access Key, Secret Key, region, and output format.**

* Once your AWS is configured, initialize the Terraform script by executing the command: terraform init. This will set up the necessary dependencies and prepare the environment for running Terraform.



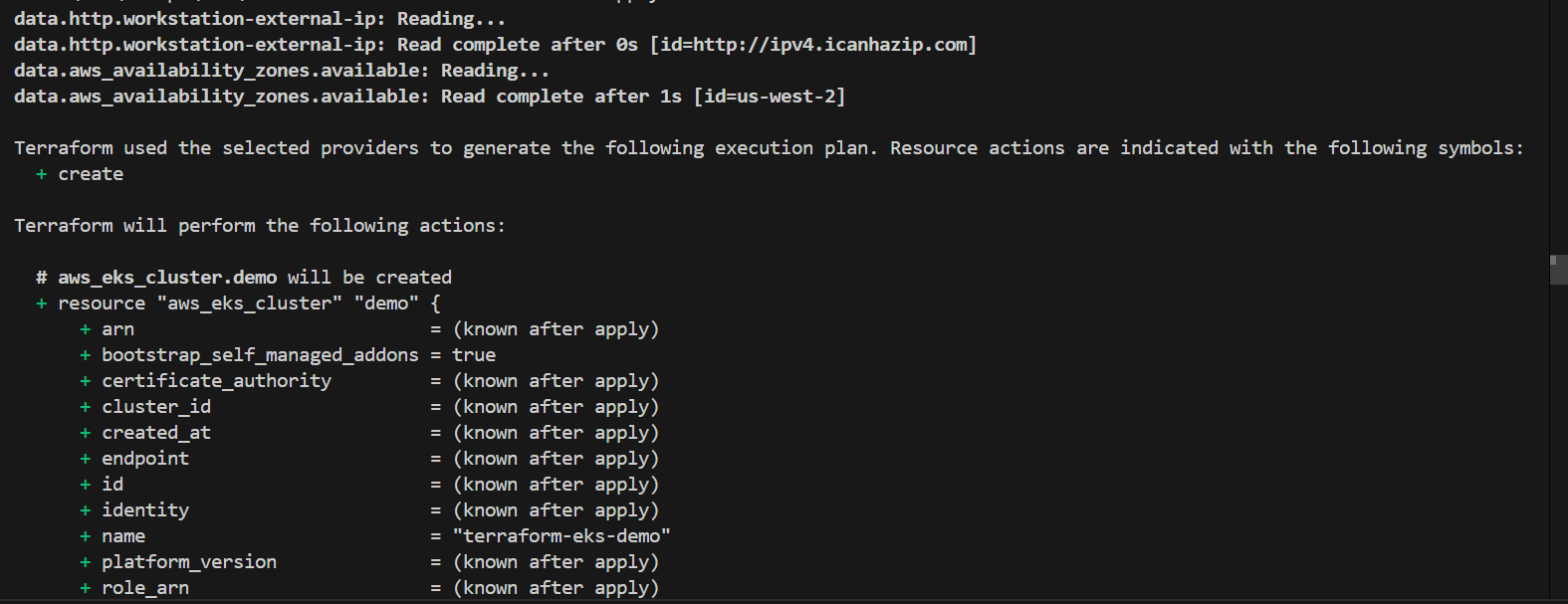


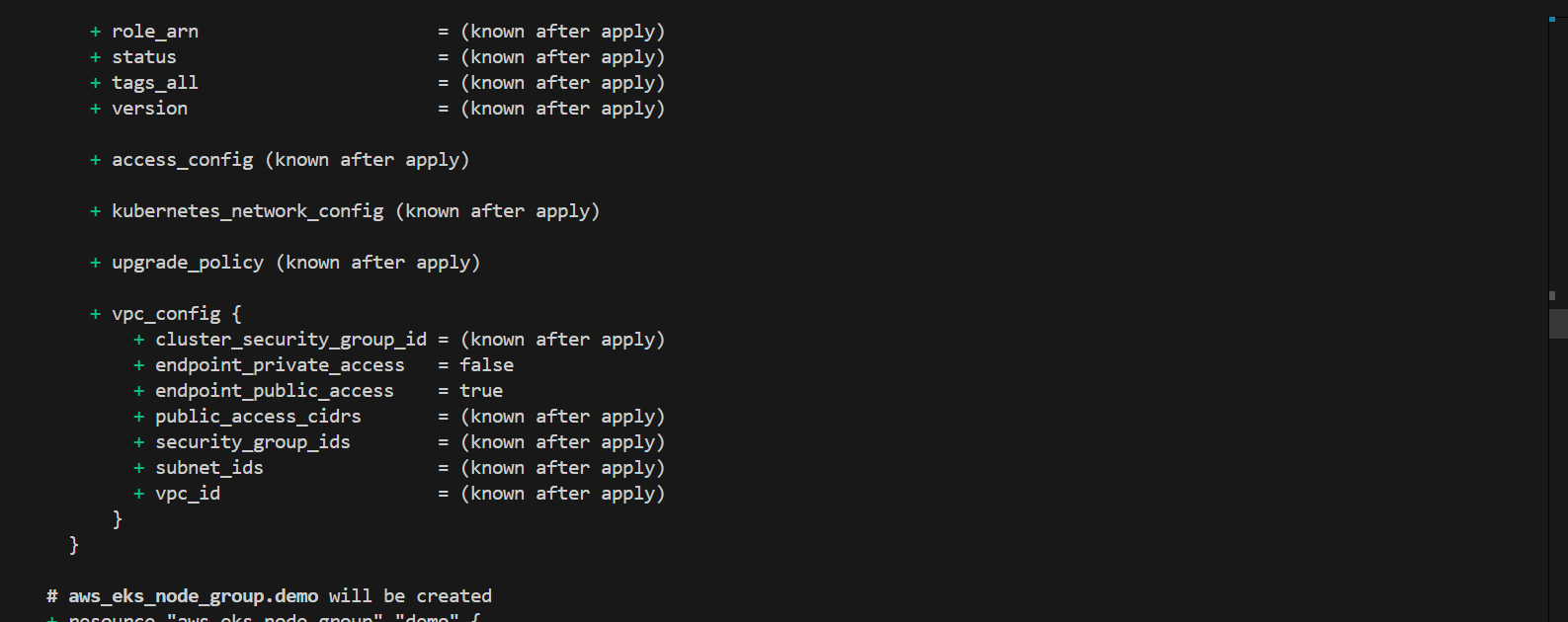
* After running terraform init, Terraform creates a lock file named **.terraform.lock.hcl** to record the provider selections it made. This file should be included in your version control system to ensure consistent provider versions are used in future runs. Terraform initialization is now complete, and you can begin working with your infrastructure. If you modify modules or backend configurations, you should rerun terraform init to reinitialize your working directory.
* Now Execute terraform plan command. The terraform plan command is used to create an execution plan for your infrastructure. It compares the current state of your infrastructure (as defined in your configuration files) with the existing state in your cloud provider (like AWS). The command shows what actions Terraform will take to align the infrastructure with your configuration, such as creating, modifying, or deleting resources. It doesn't make any changes to your infrastructure; it simply provides a preview of what will happen when you run terraform apply.

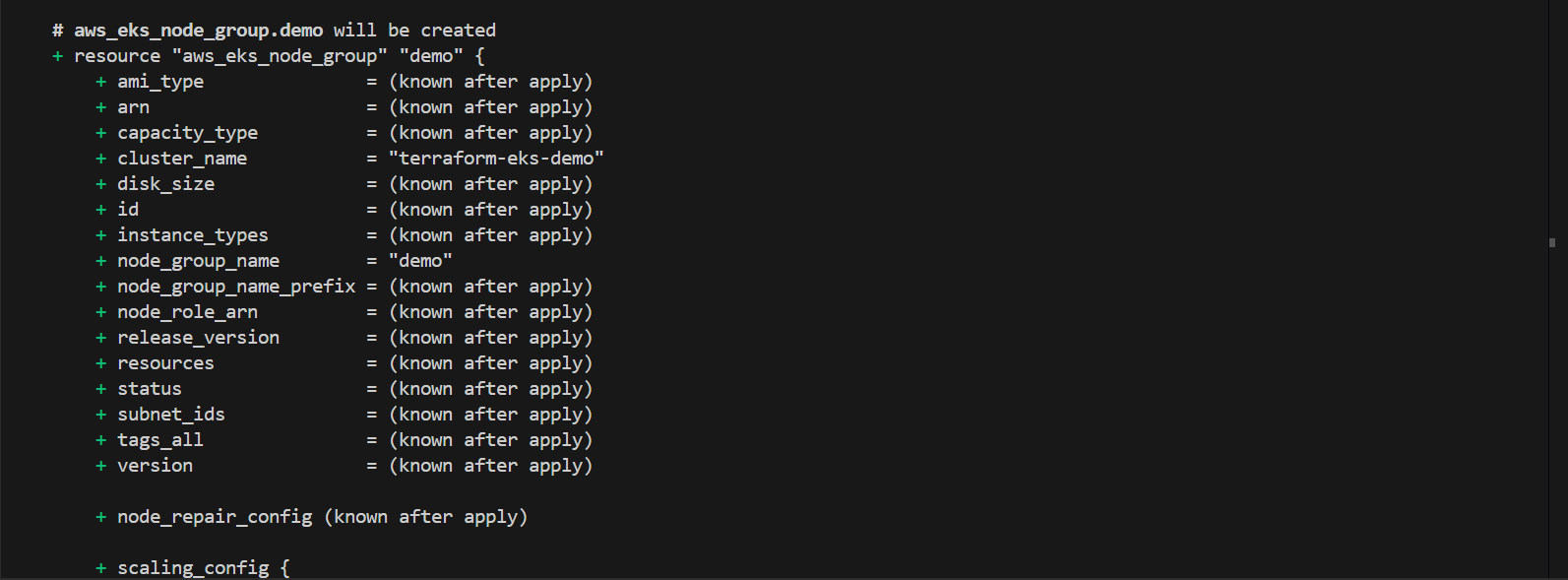


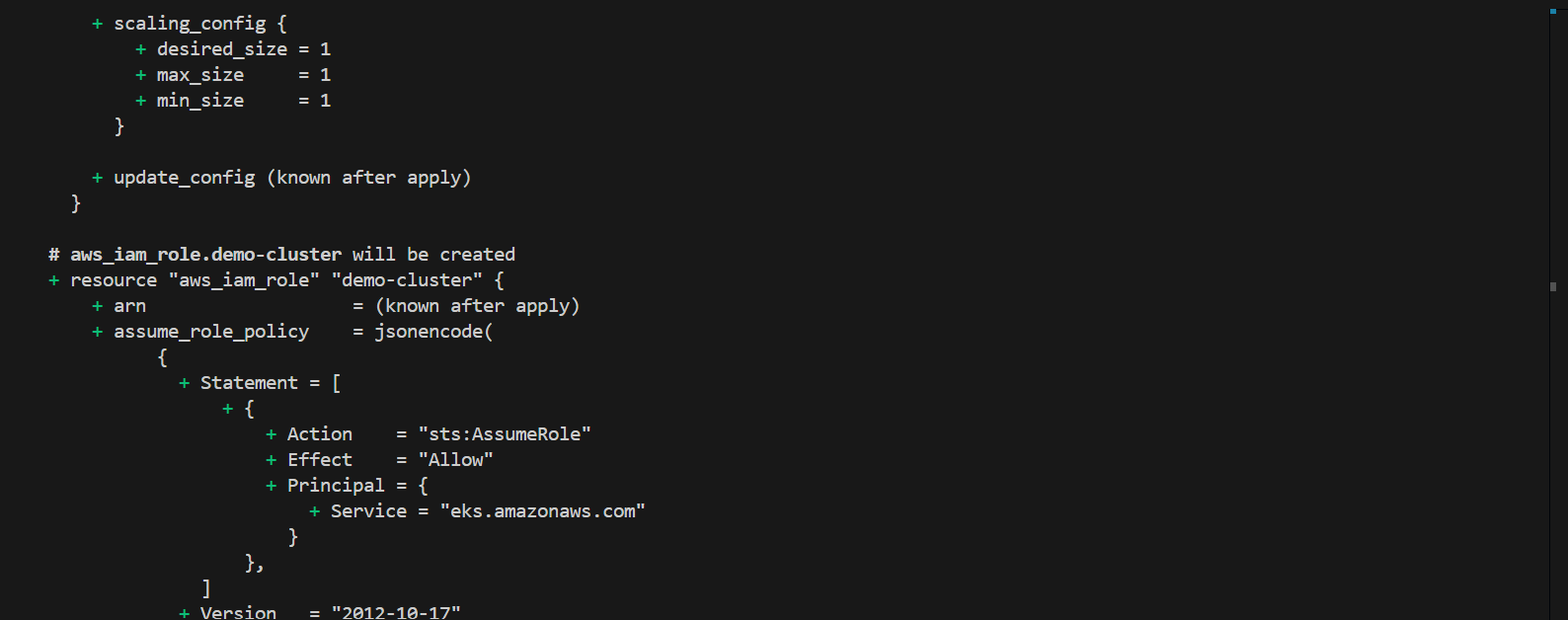
The terraform plan output **shows that Terraform will create a total of 18 resources** in your AWS environment, including an **EKS cluster** (aws\_eks\_cluster.demo), **node group** (aws\_eks\_node\_group.demo), **IAM roles, security groups, a VPC with subnets, route tables, and an internet gateway.** It also lists 0 resources to be modified or destroyed. The plan outlines the exact resources Terraform will add, along with any associated configurations, such as IAM role policy attachments and subnet associations. There are warnings about deprecated attributes, indicating that some of the configuration elements are outdated and may need to be updated in the future. This plan provides a detailed preview of what will be created when you apply the configuration, ensuring you can review the changes before proceeding.

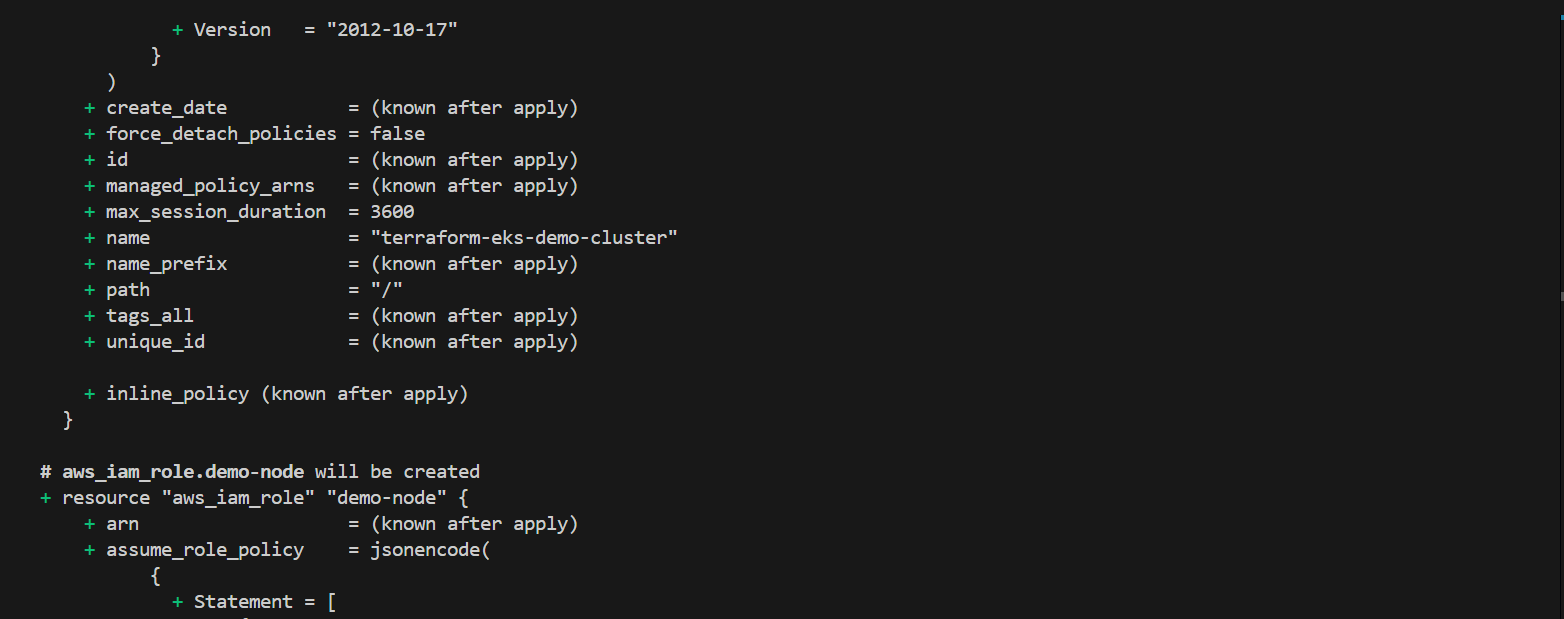
* Next, we need to use the terraform apply command to create the resources outlined in the terraform plan output.



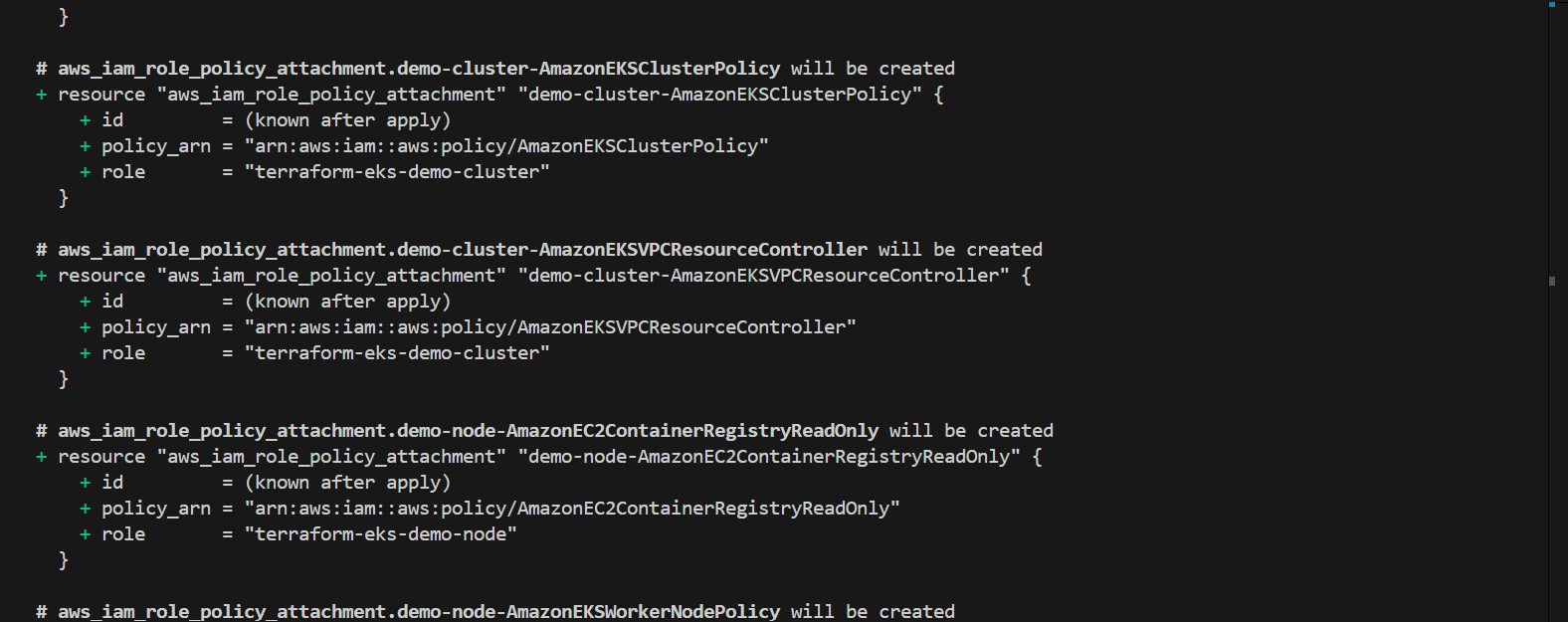


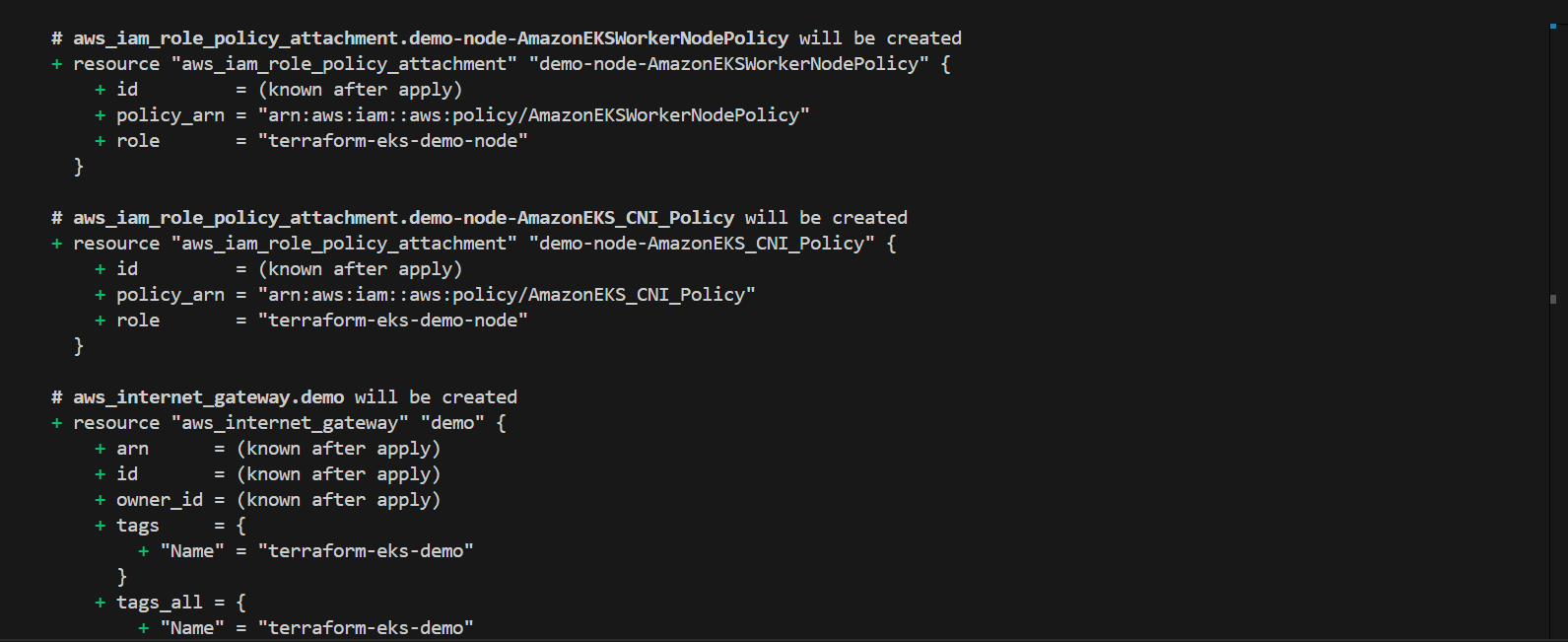


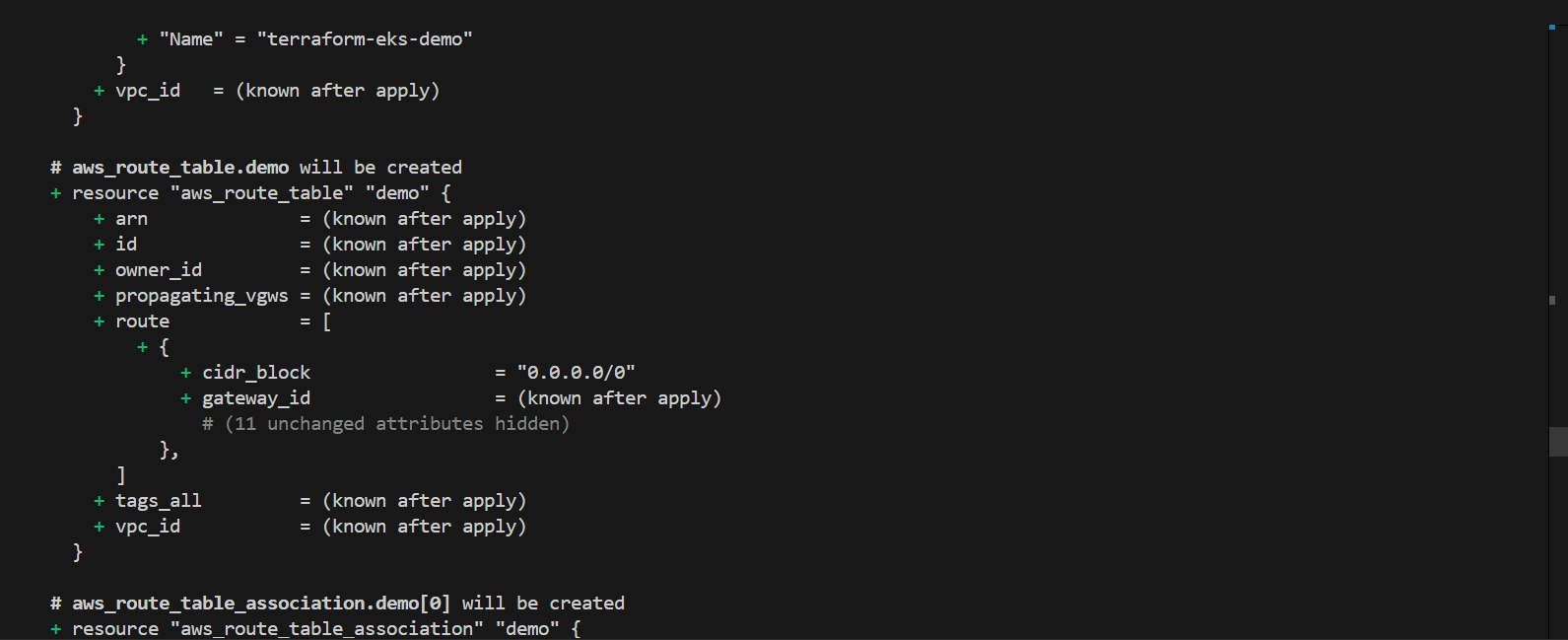


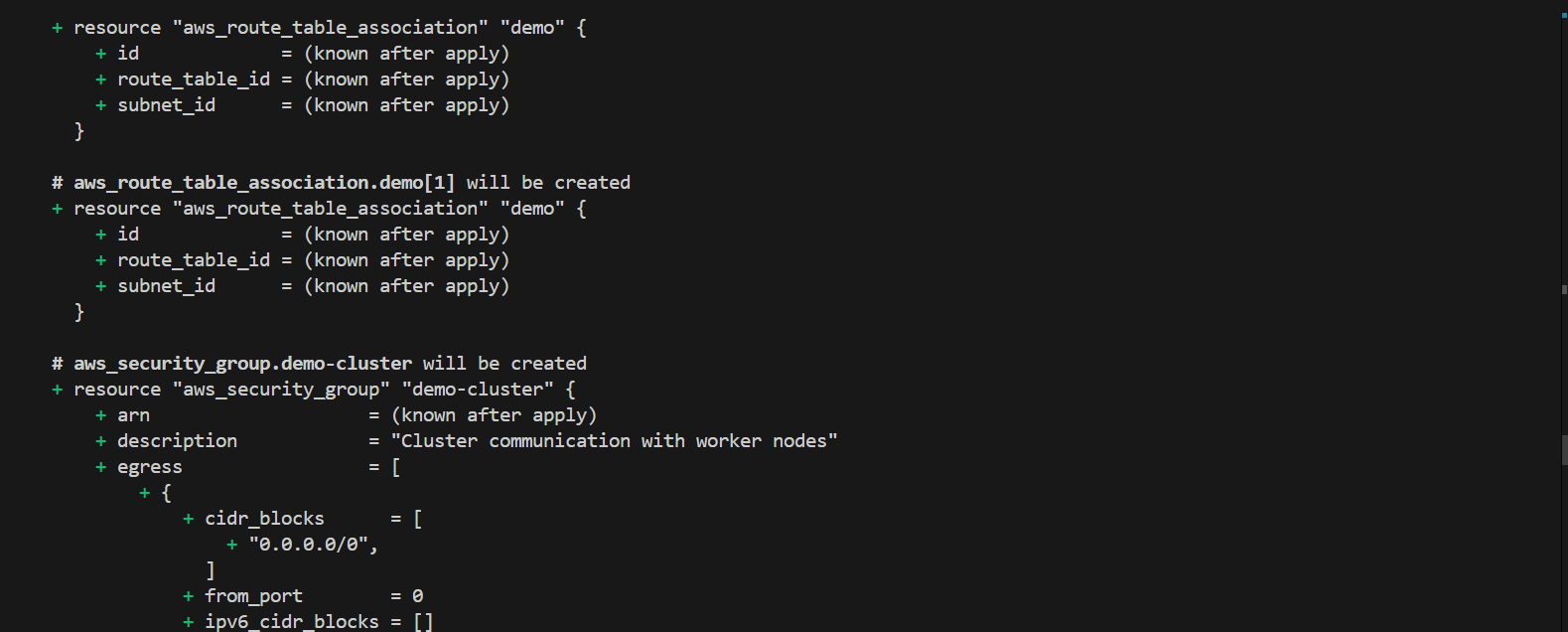


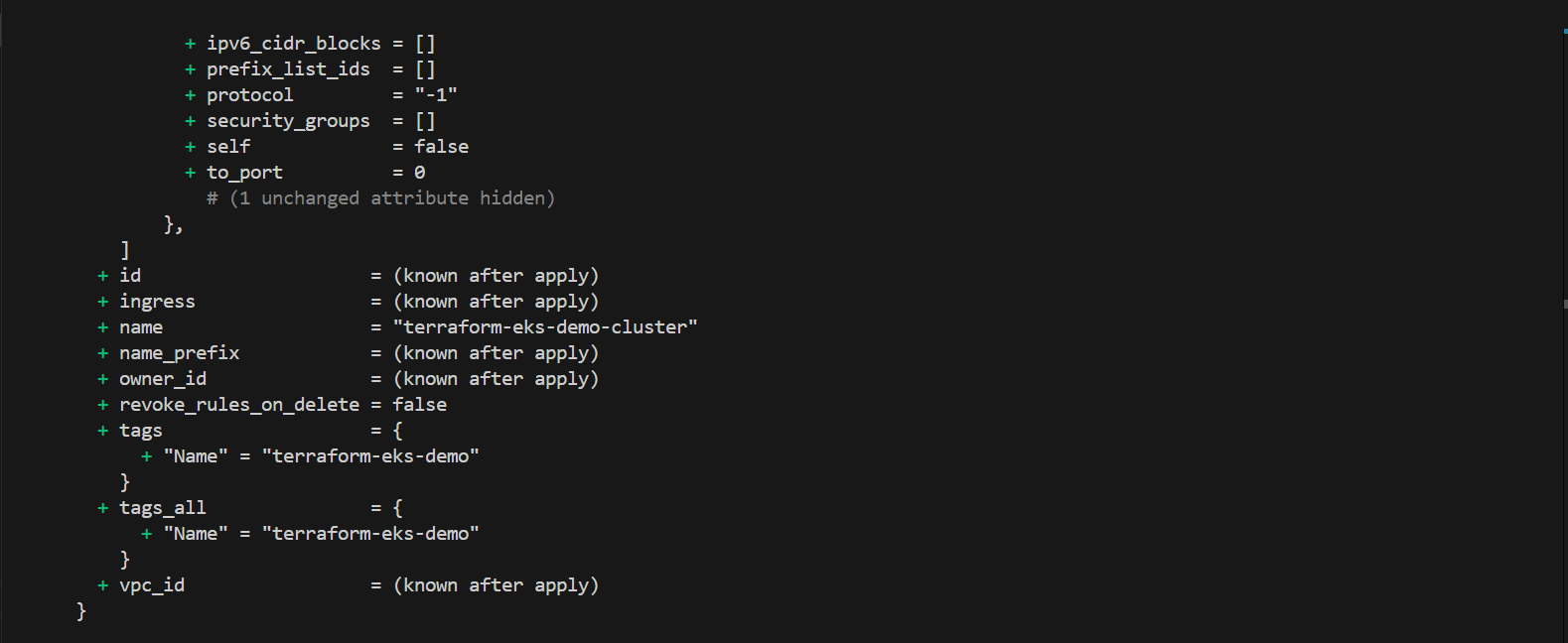


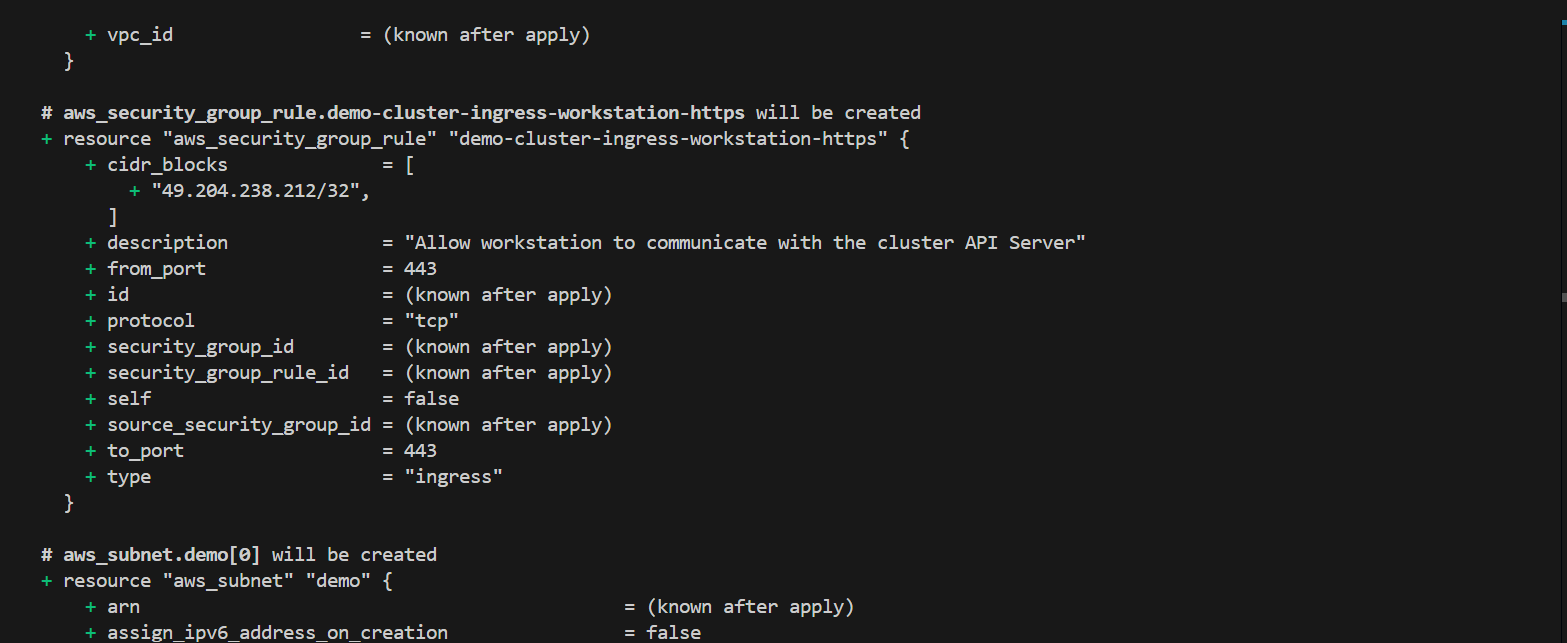


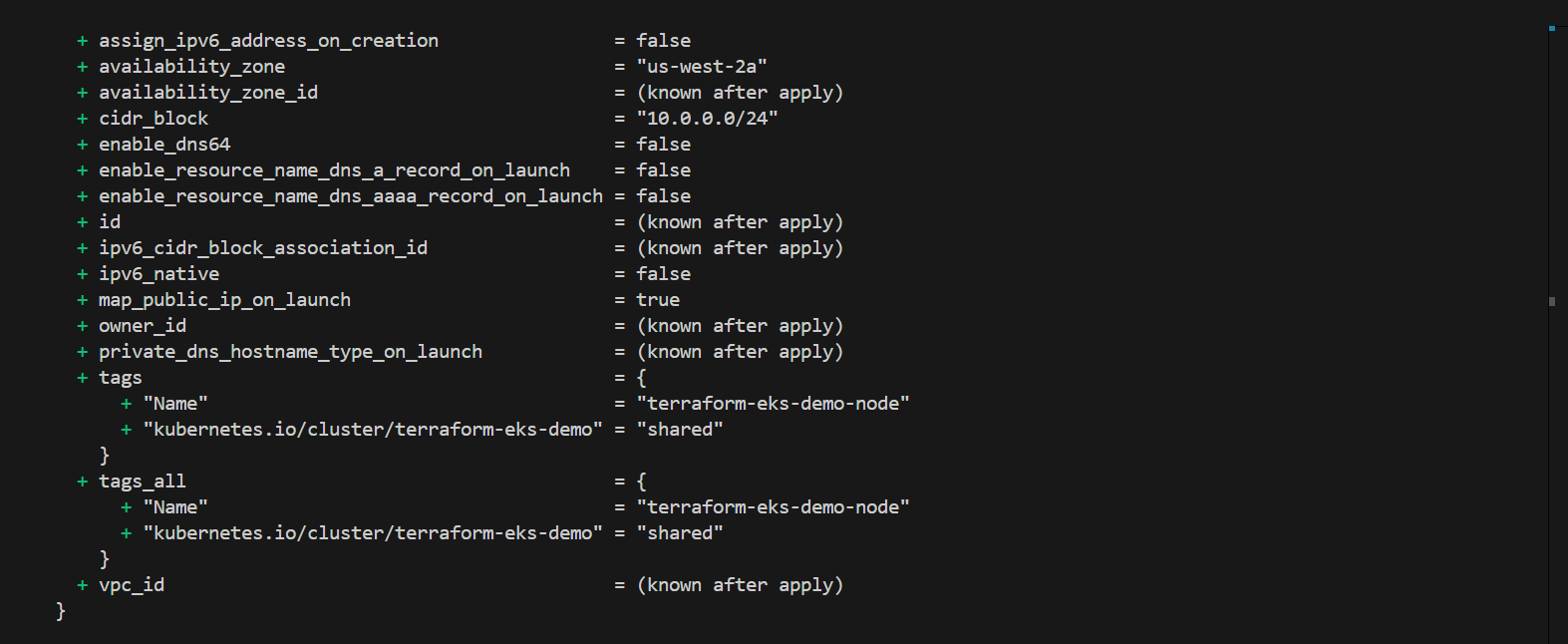




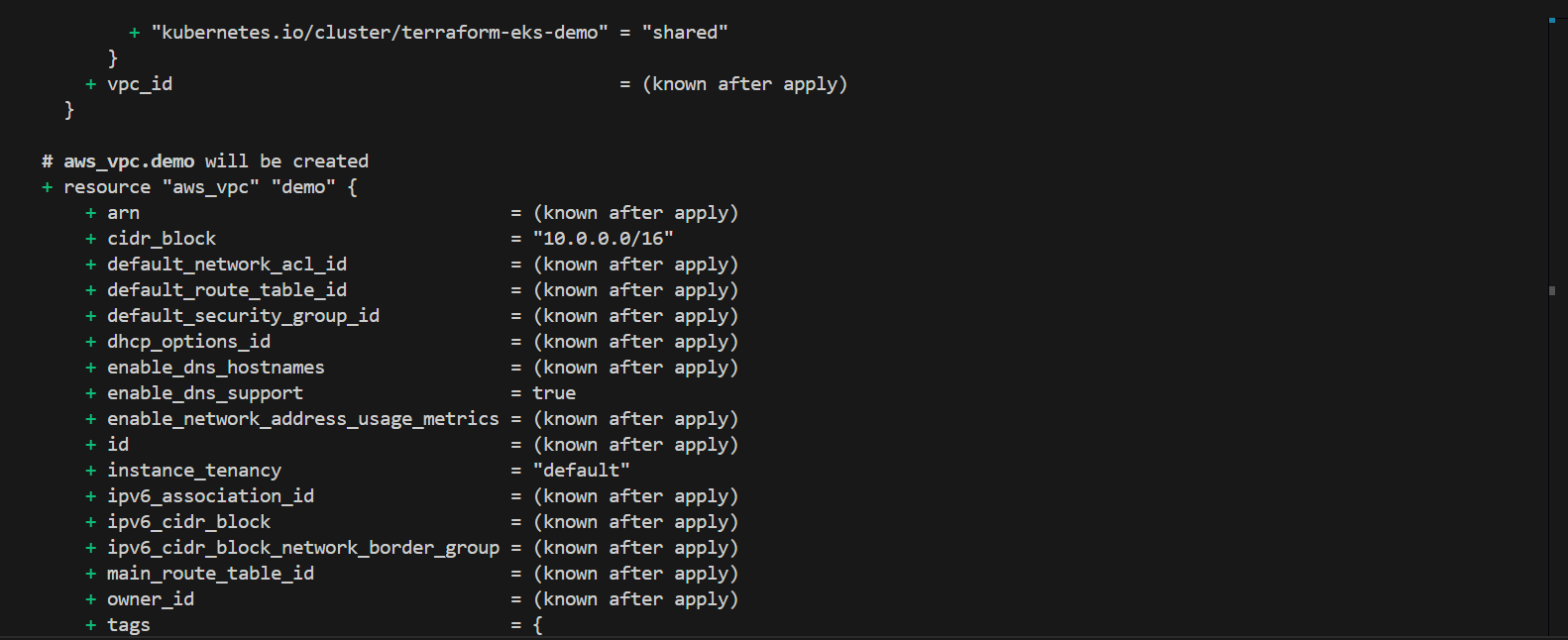


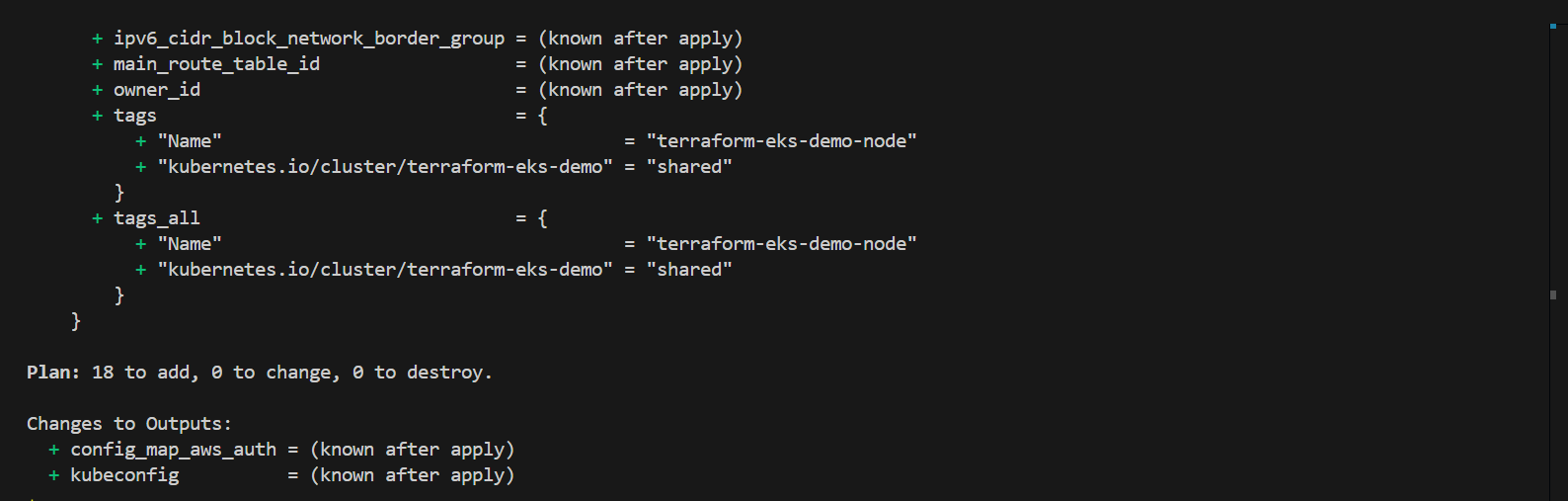






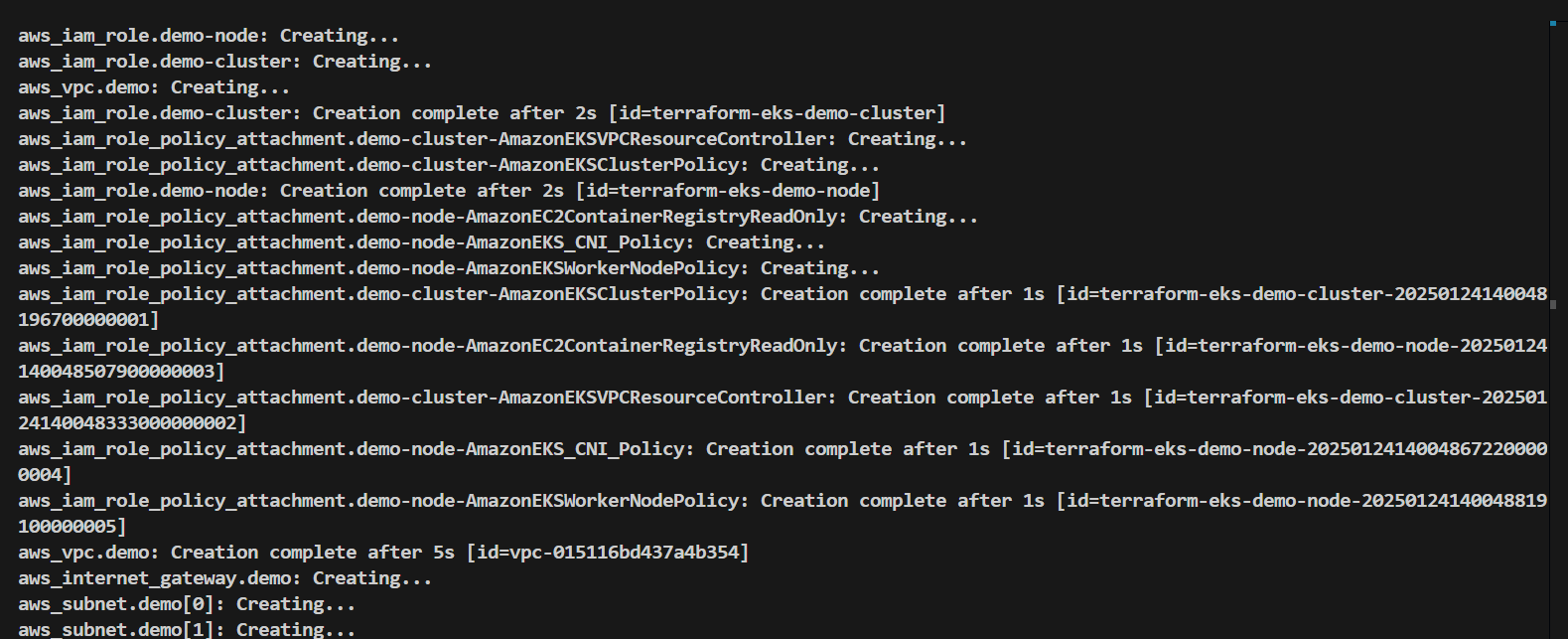


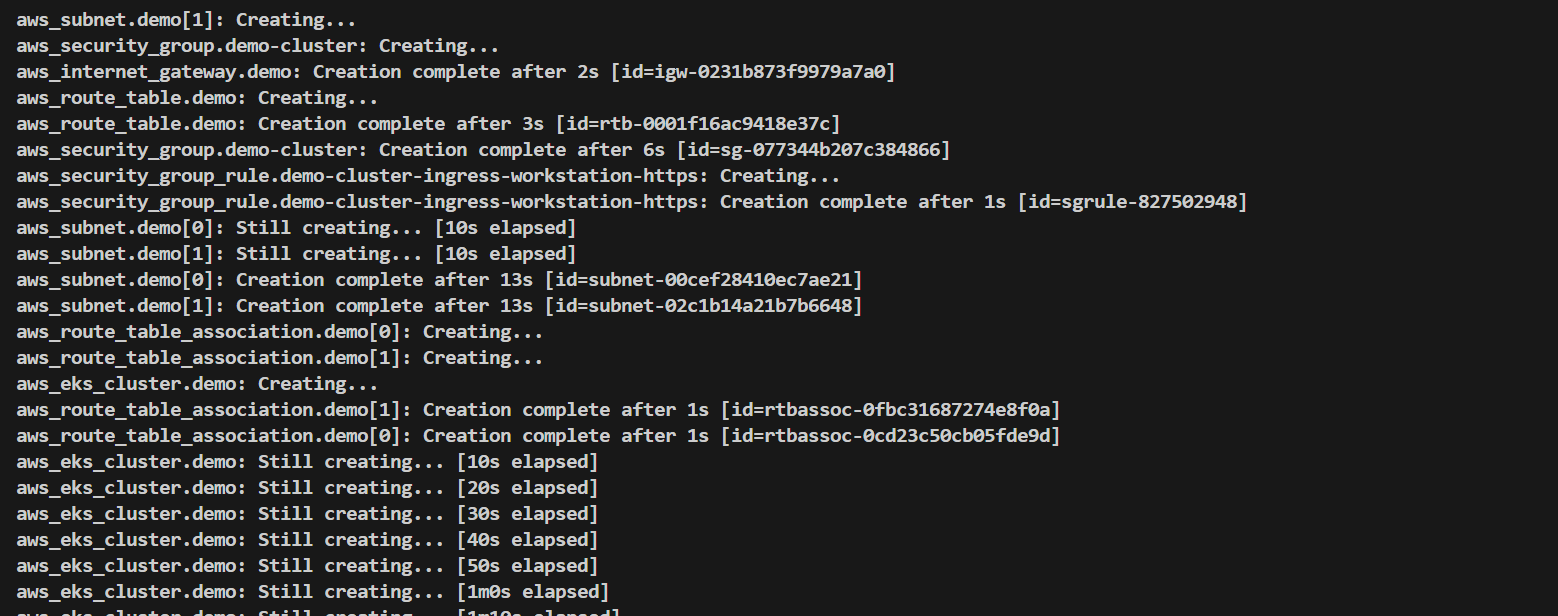


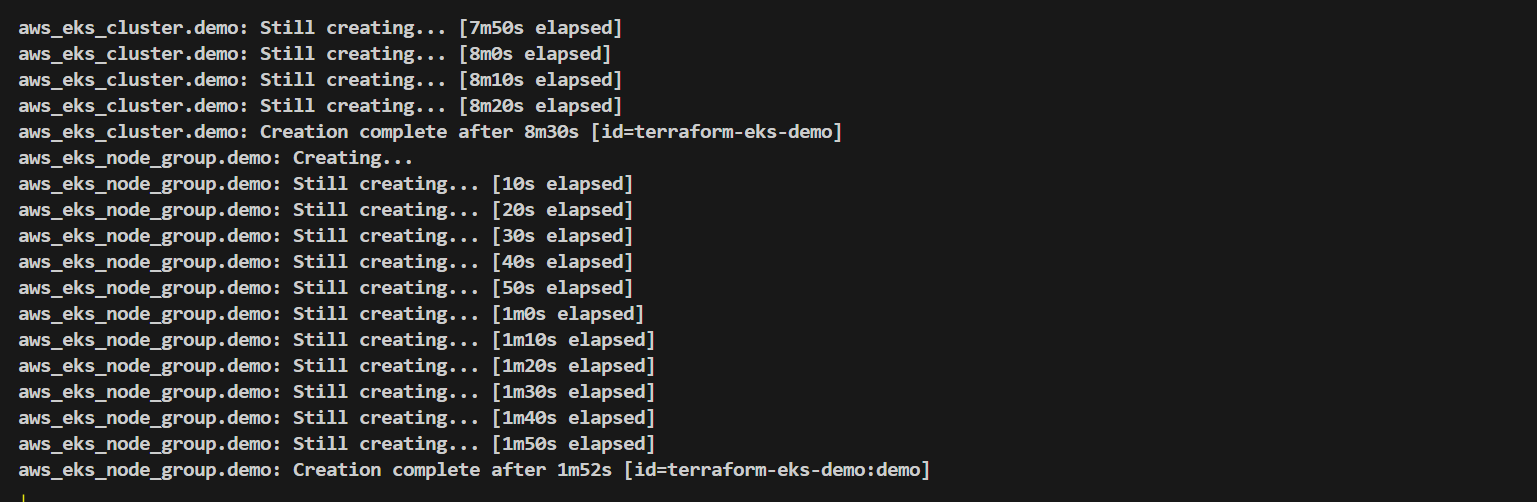


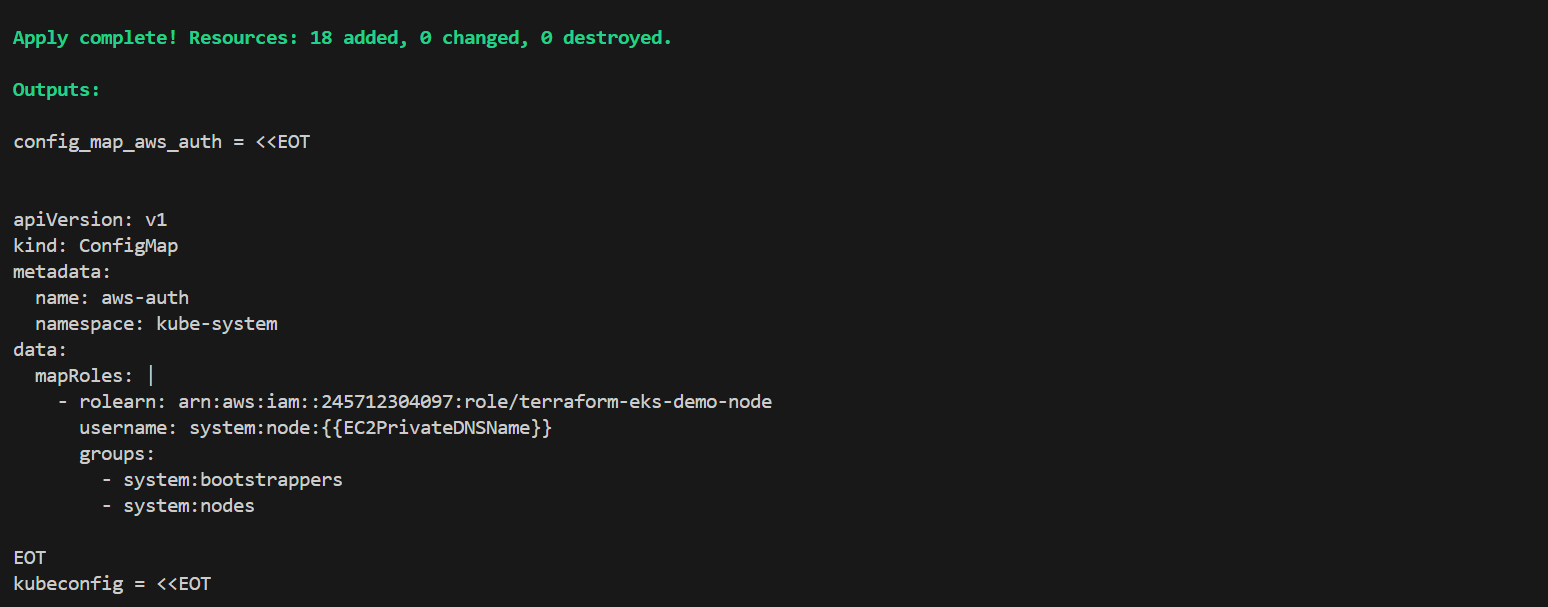


* Enter yes to confirm the creation of cluster in your AWS.

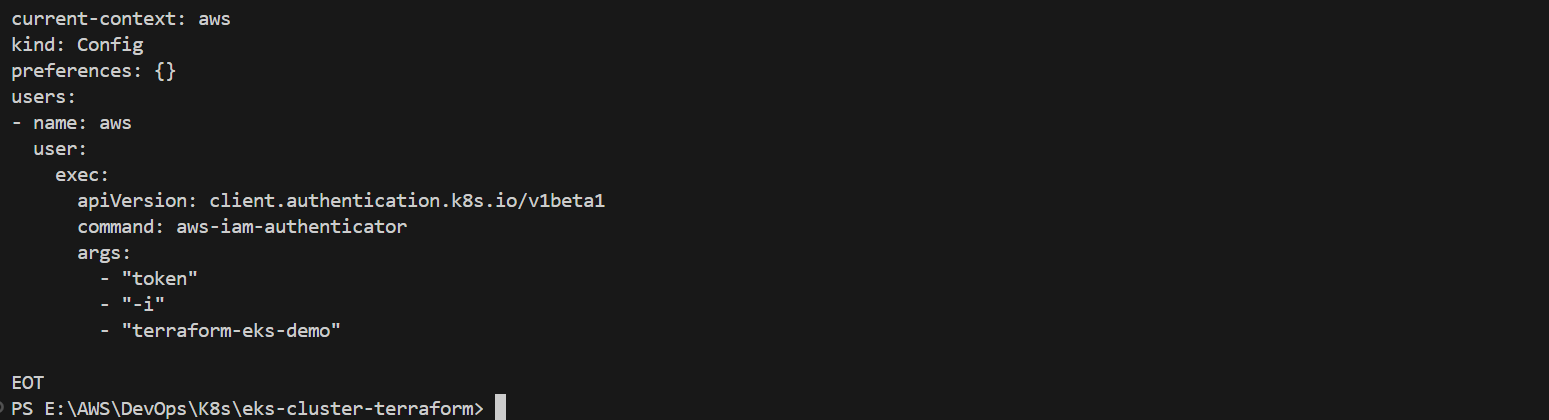




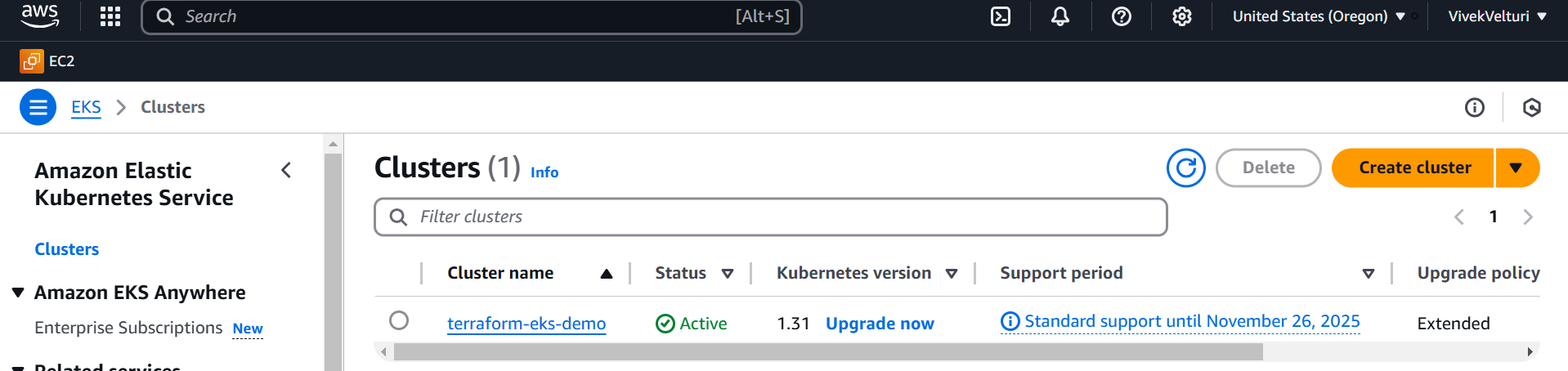


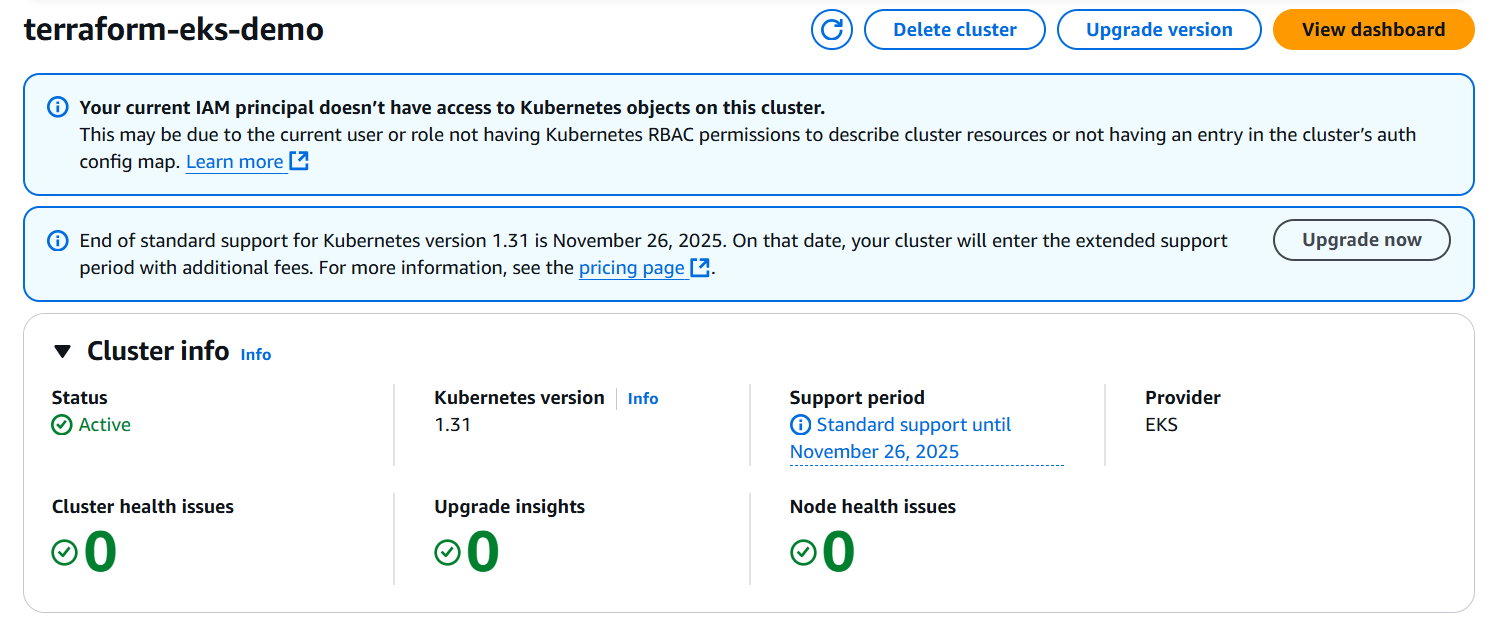


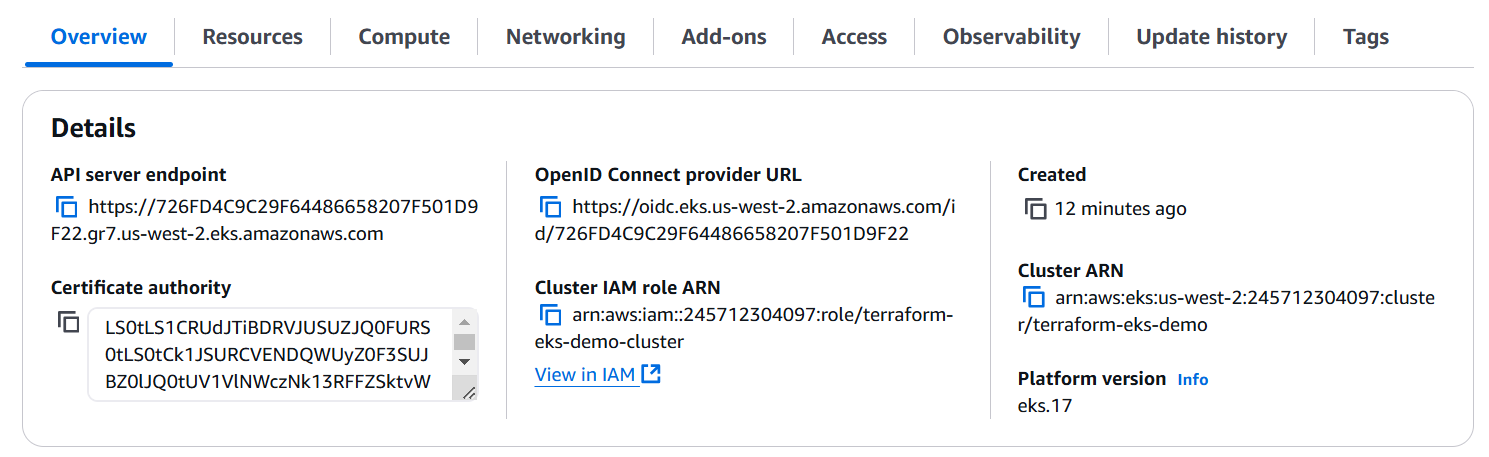


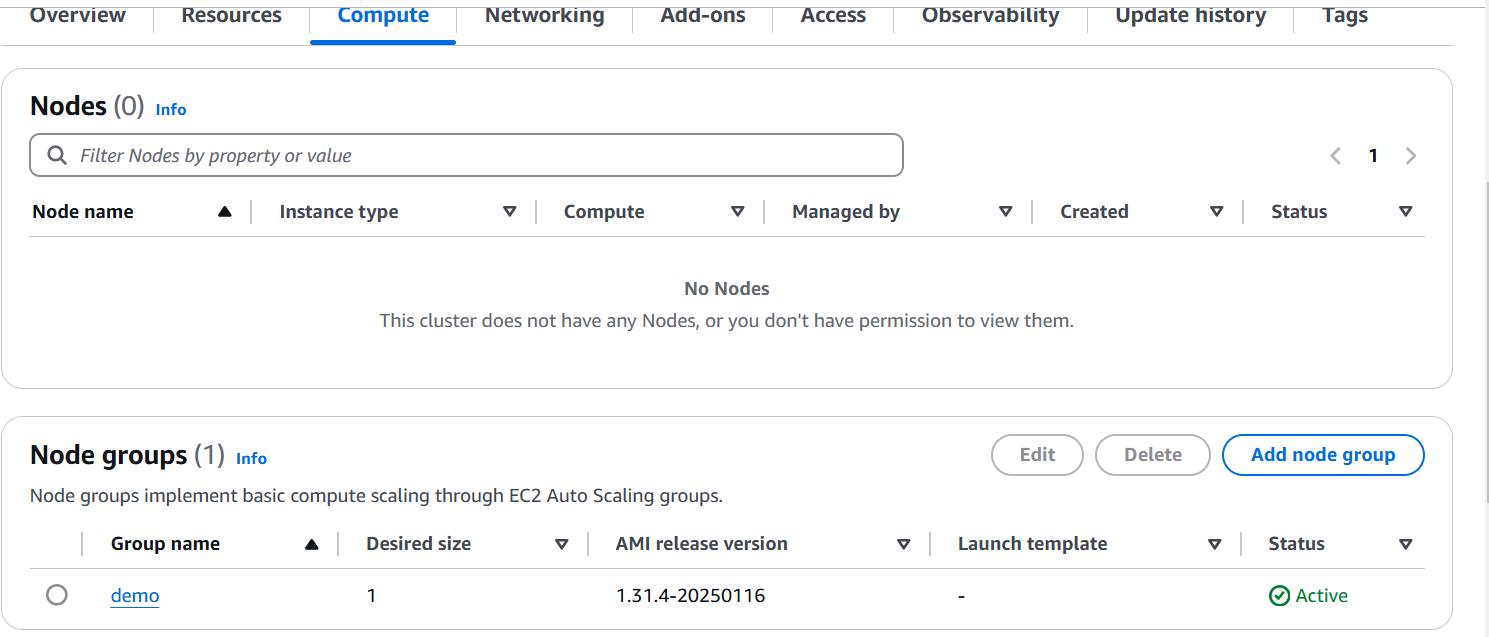


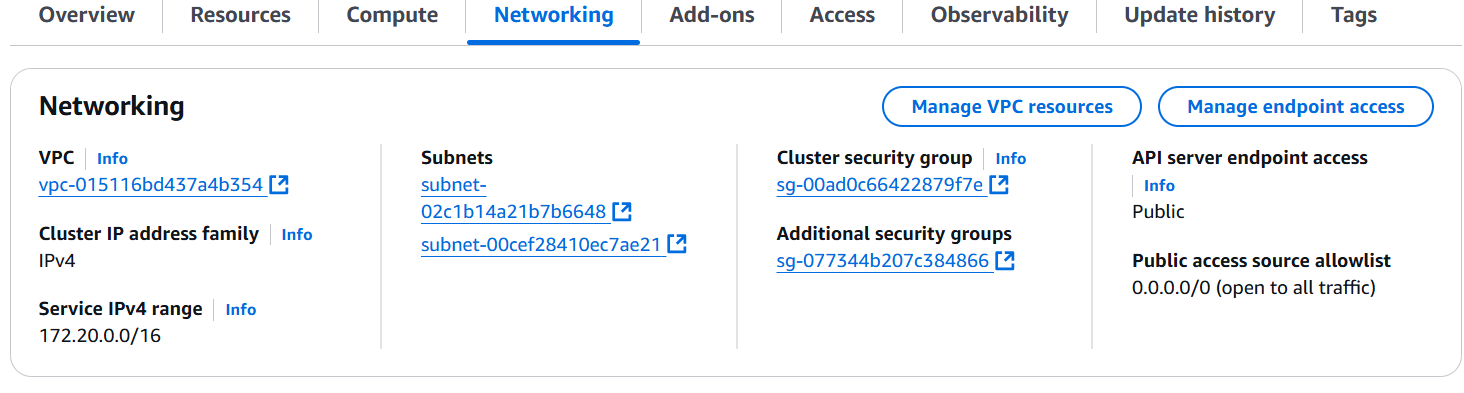
* After running the terraform apply command, log in to your AWS Management Console and navigate to the region where the resources were deployed using the Terraform script. Verify that an EKS cluster has been created and check the EC2 dashboard to confirm that an instance has been launched for the node.

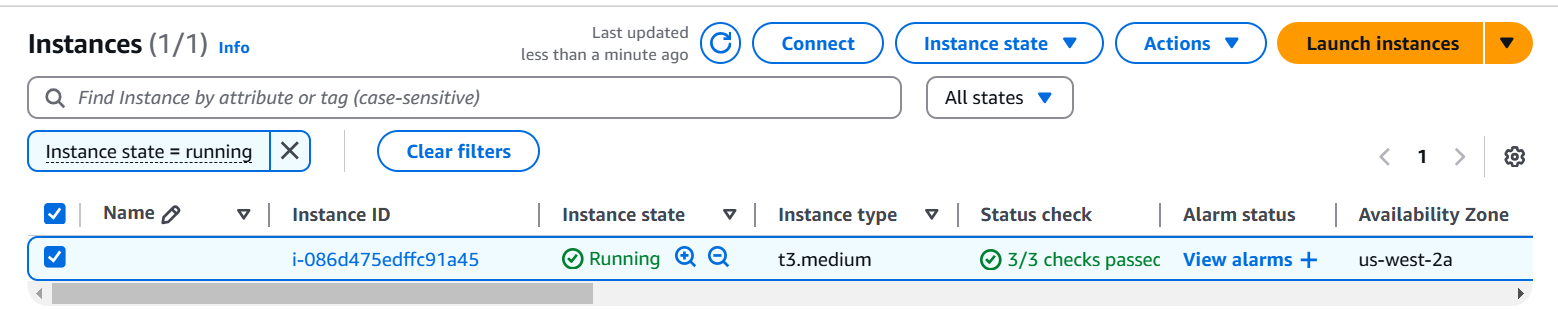


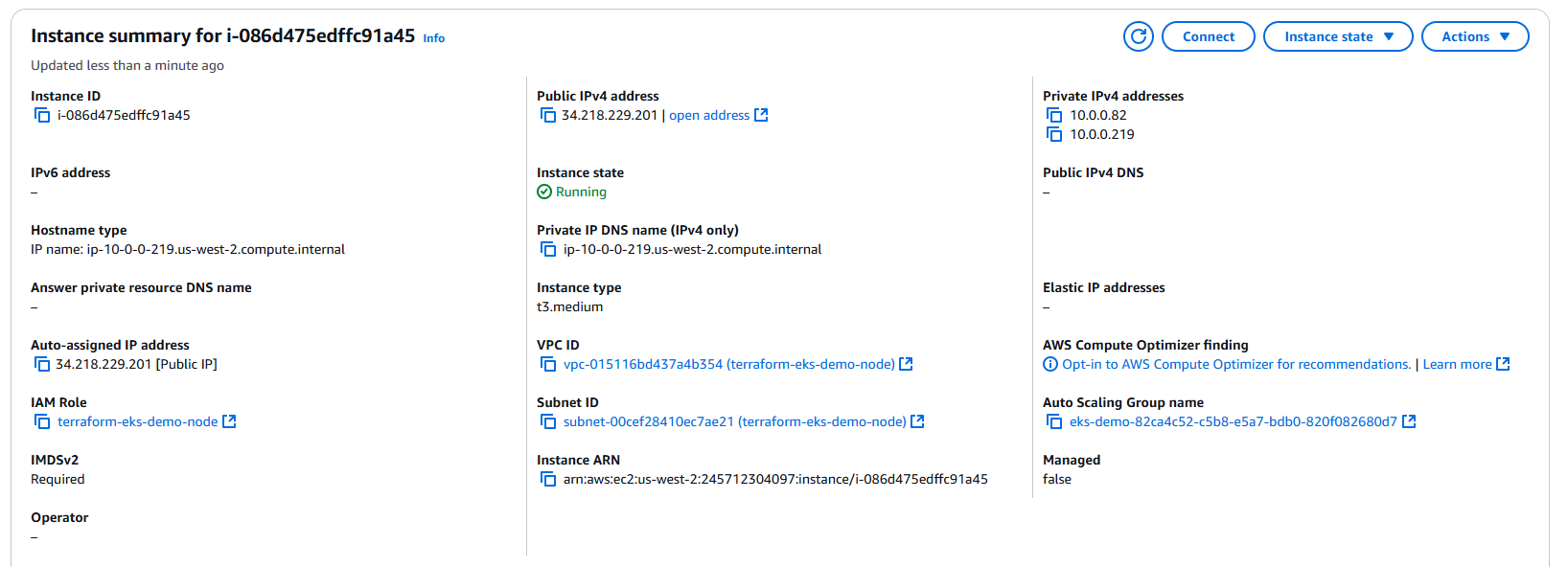


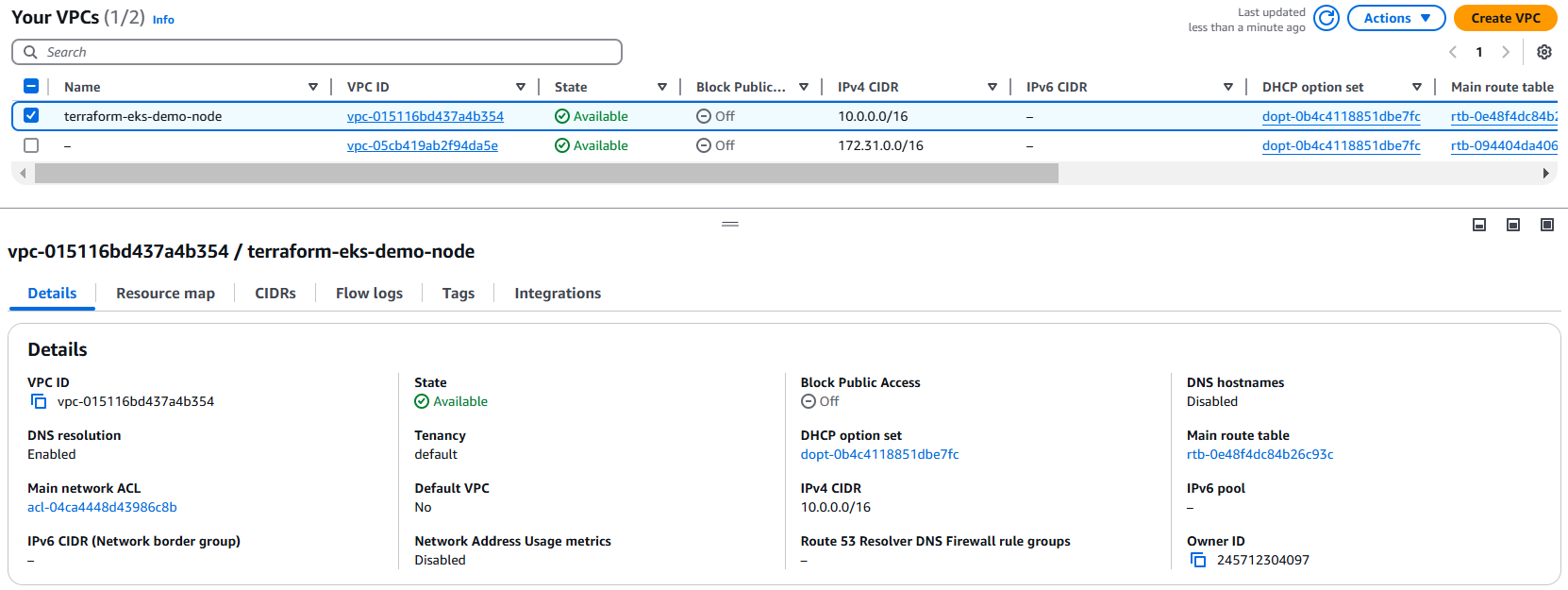


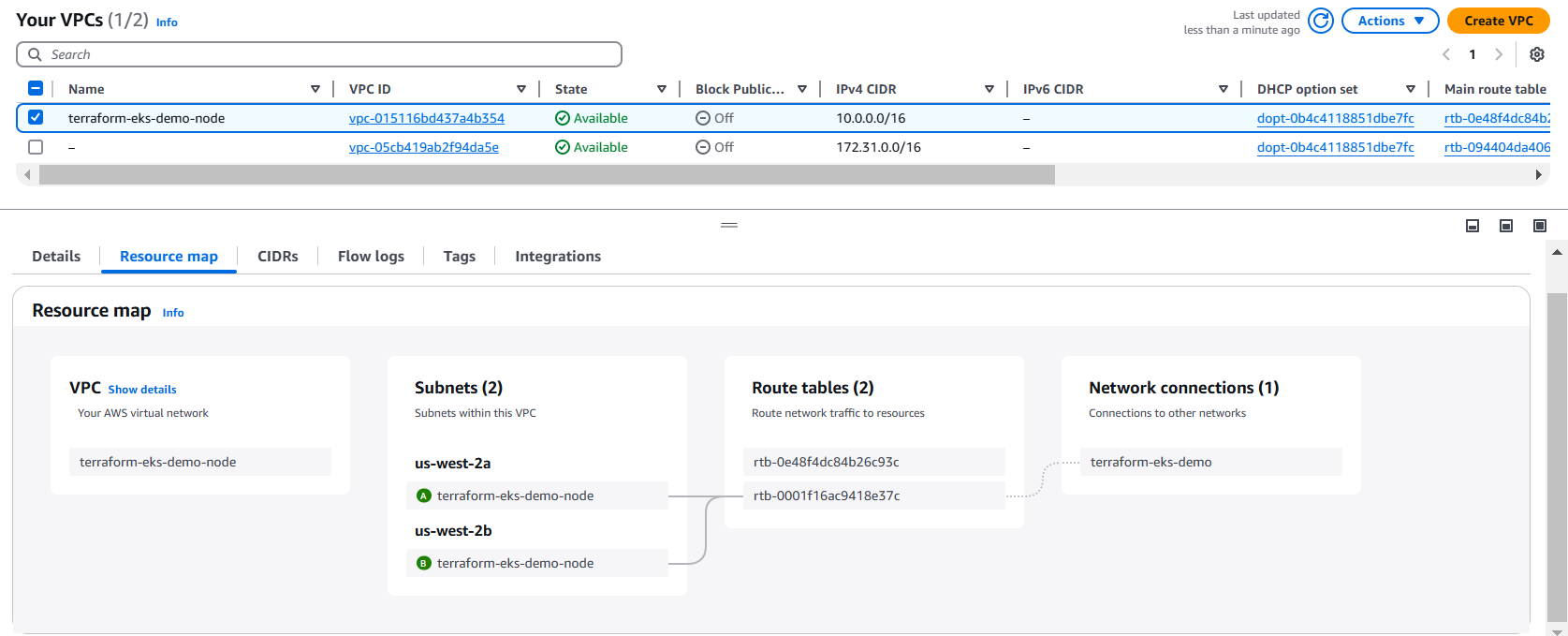


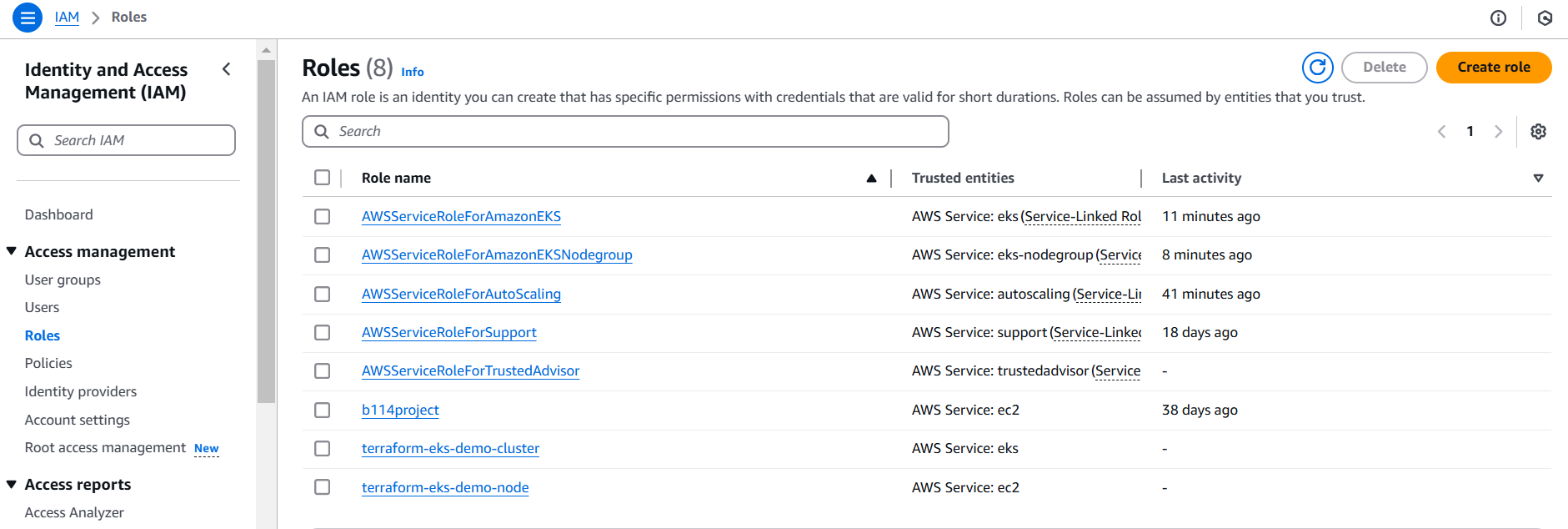




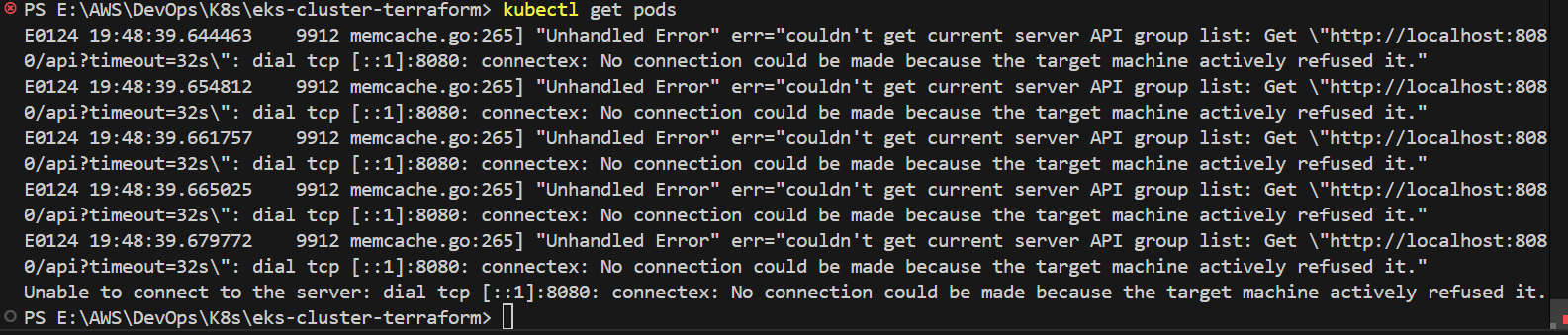








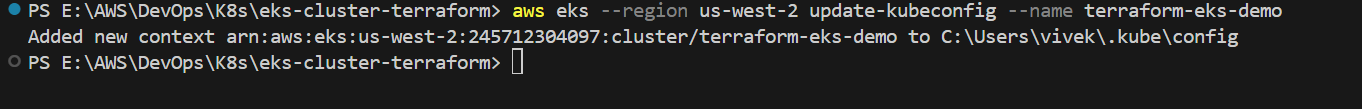
* The kubectl utility, which we installed earlier on our local computer, serves as the command-line interface to communicate with Kubernetes clusters. Once the EKS cluster has been successfully created in AWS, kubectl allows us to manage the cluster and its resources. This includes tasks such as deploying applications, scaling services, monitoring workloads, and managing configurations. By connecting kubectl to the cluster, we can execute commands directly from our local machine to interact with the Kubernetes environment running in AWS.
* To verify the connection between kubectl and the EKS cluster, use the following command: kubectl get pods



* If you encounter an error such as **unable to connect to the server**, it indicates that kubectl is not yet configured to interact with the cluster. To establish this connection, use the following command:

aws eks --region <region> update-kubeconfig --name <cluster-name>

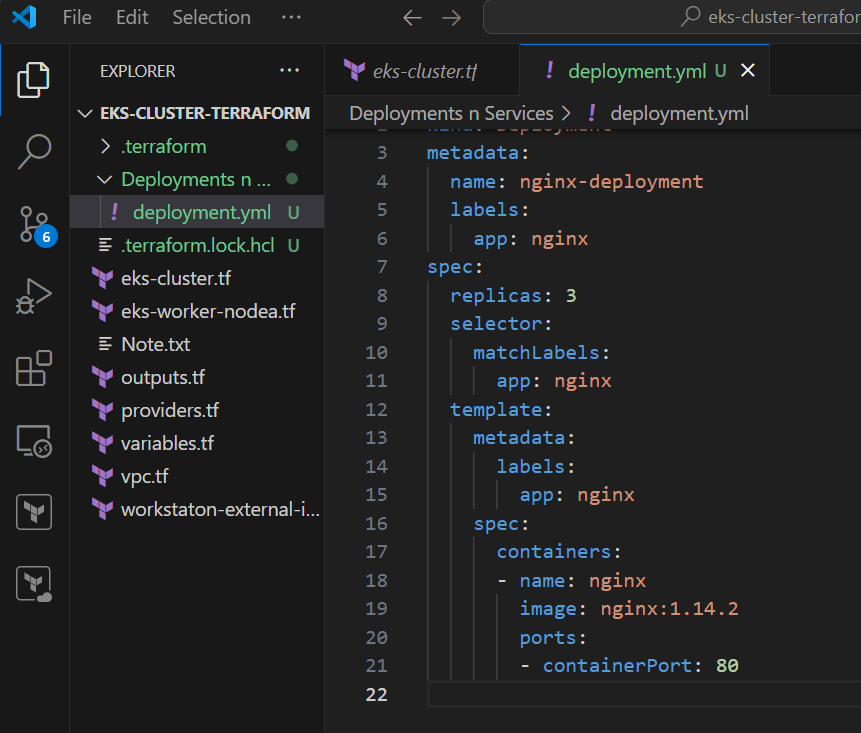
For example, if your cluster is in the us-west-2 region and its name is terraform-eks-demo, the command would be: aws eks --region us-west-2 update-kubeconfig --name terraform-eks-demo



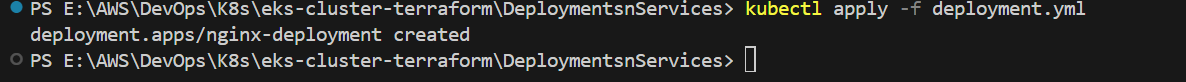
* Next, let’s set up a deployment. Begin by creating a folder named **Deployment n Services** inside the **eks-cluster-terraform** directory, where your Terraform scripts are stored. Inside this folder, create a YAML file called **deployment.yml**. Obtain the deployment script from the official Kubernetes website, paste it into the file, and save it using **Ctrl+S**. This YAML file will define a deployment configuration with a replica set of 3, which will create and manage pods within the cluster.

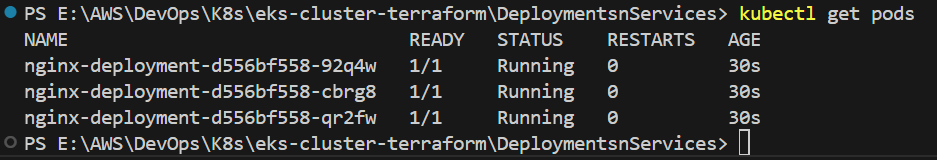


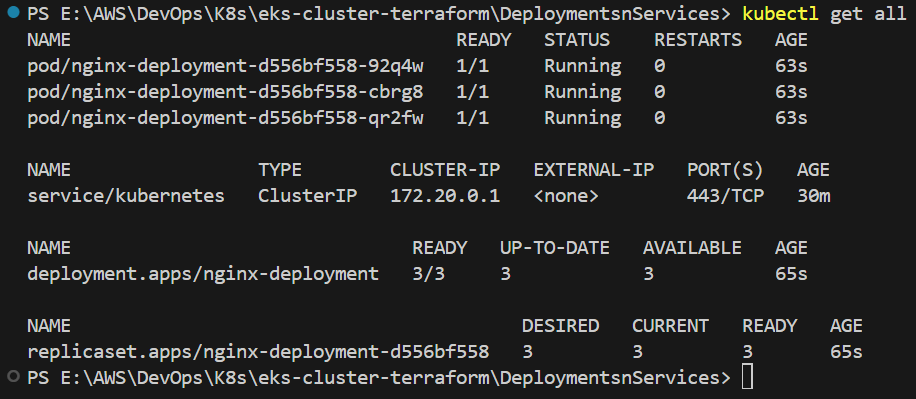




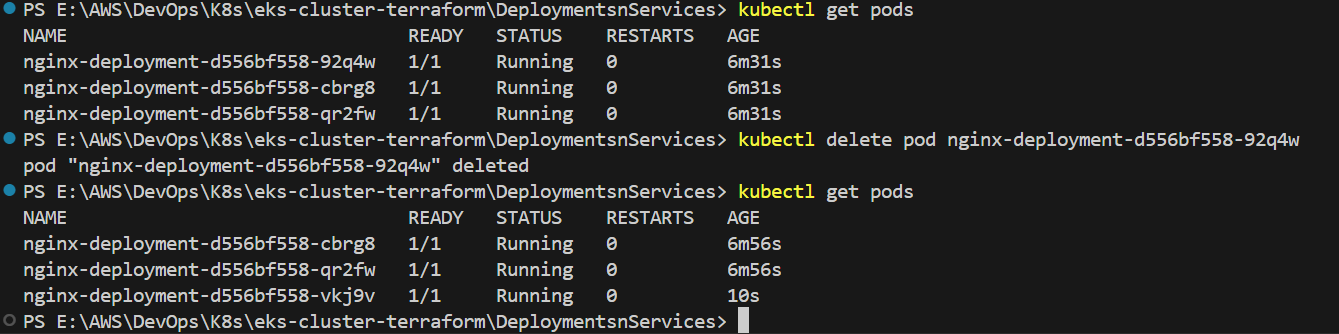
* After saving the deployment file, run the command kubectl apply -f deployment.yml to create the pods. Once the command is executed, you can verify the pods have been created by running kubectl get pods.







* If we delete any pods created through the deployment, new pods will automatically be created to maintain the specified replica set defined in the deployment configuration.



* Next, visit the official Kubernetes website and download the YAML script for creating a NodePort service. Create a new file named **nodeport.yml** inside the **Deployments n Services** folder and paste the downloaded script into it.



* In the NodePort script, you will notice the following selector:

selector:

app.kubernetes.io/name: MyApp

However, in our deployment, the pods are created with the label app: nginx. To ensure the NodePort service targets the pods created by our deployment,

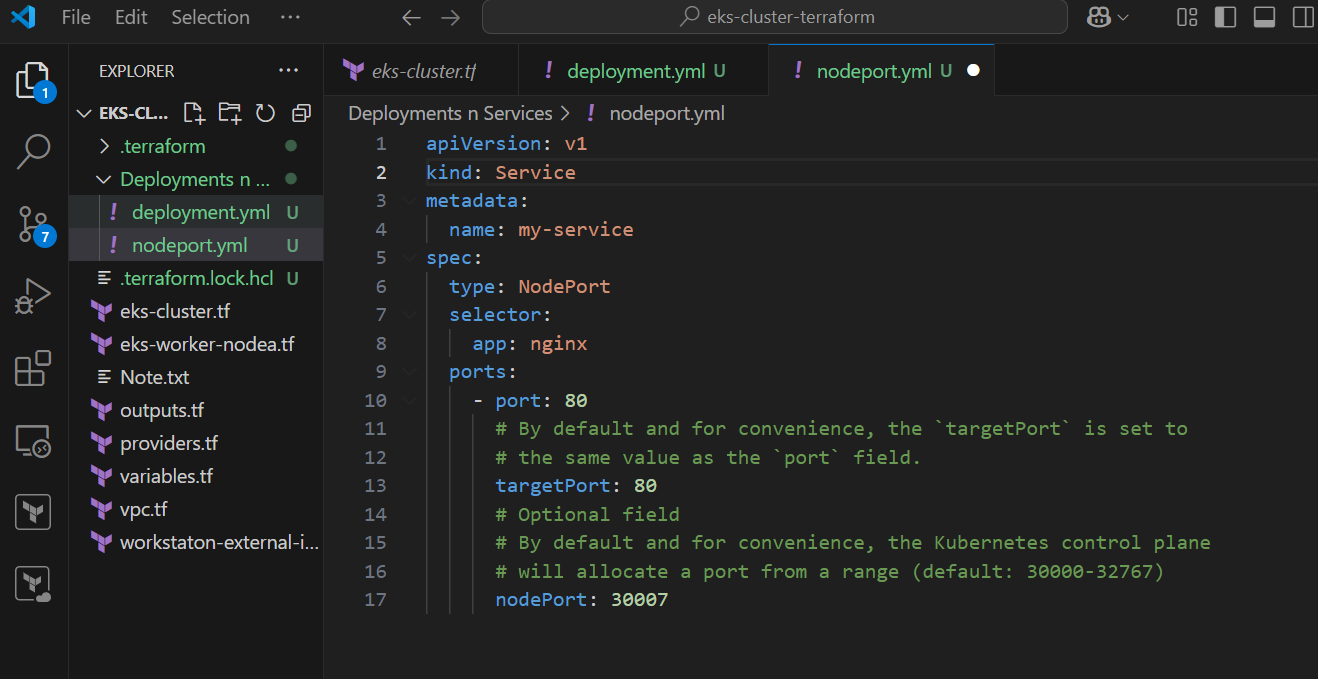
replace:

app.kubernetes.io/name: MyApp

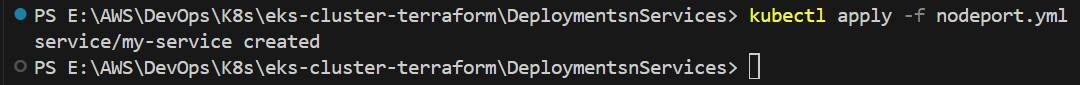
with:

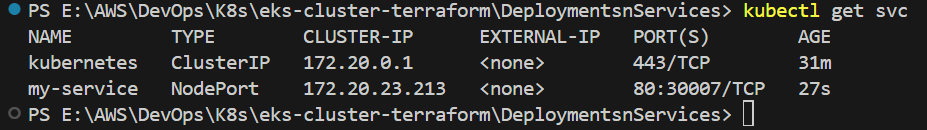
app: nginx

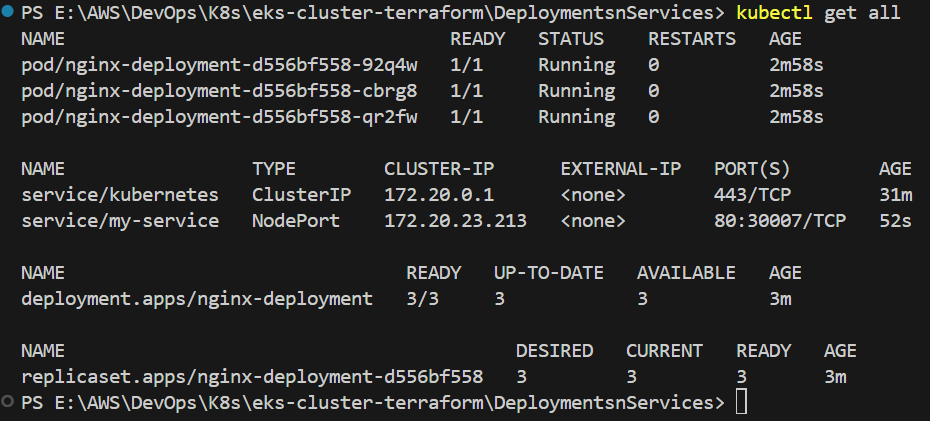
This change will allow the NodePort service to apply correctly to the pods managed by the deployment.



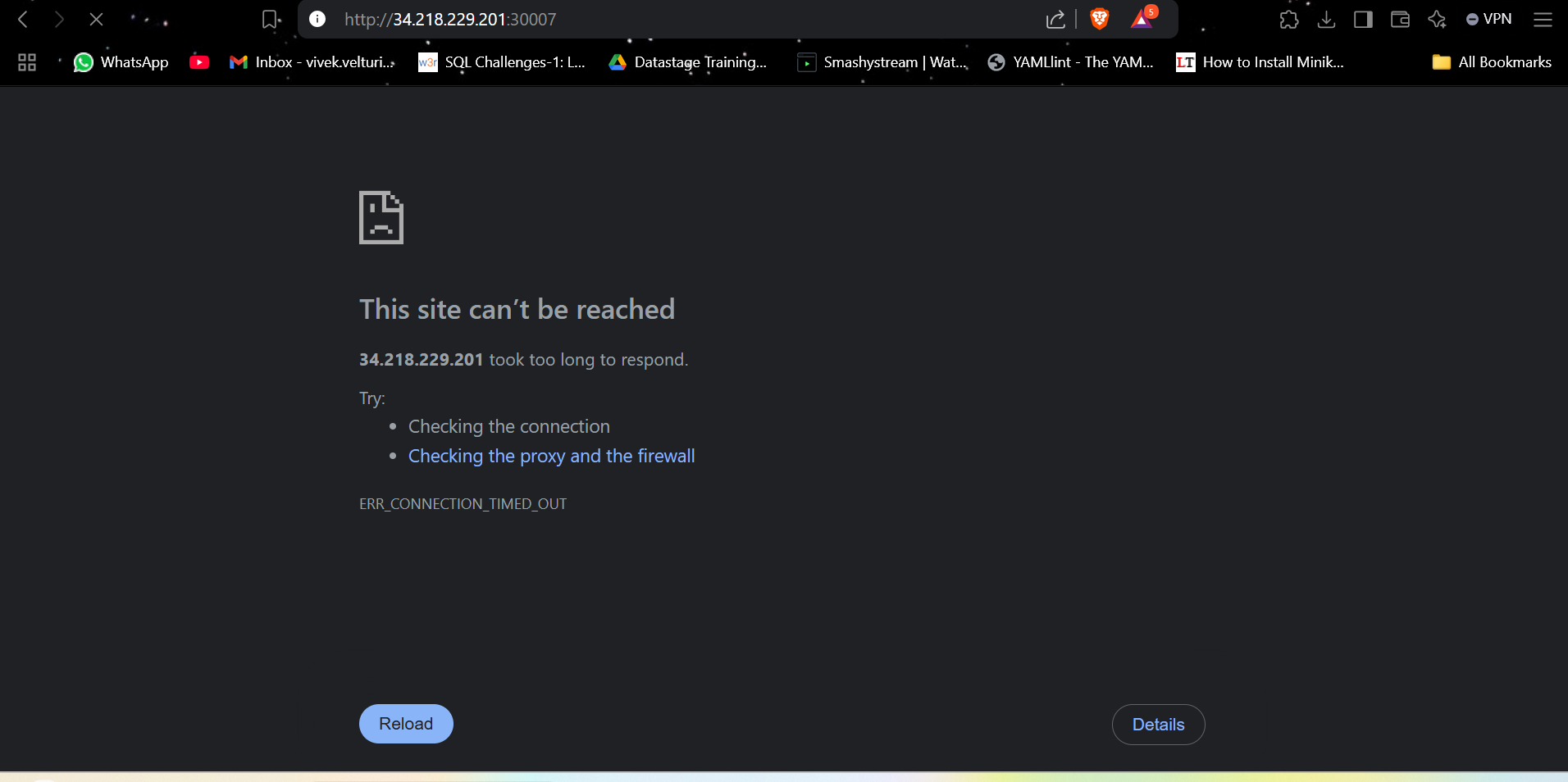
* After saving the nodeport.yml file, execute the command kubectl apply -f nodeport.yml to create the NodePort service. Once the service is successfully created, you can verify its status and details by using either kubectl get svc to list all services or kubectl get all to view all resources in the cluster, including the newly created service. These commands will confirm that the NodePort service has been applied and is ready to route traffic to the pods managed by your deployment.



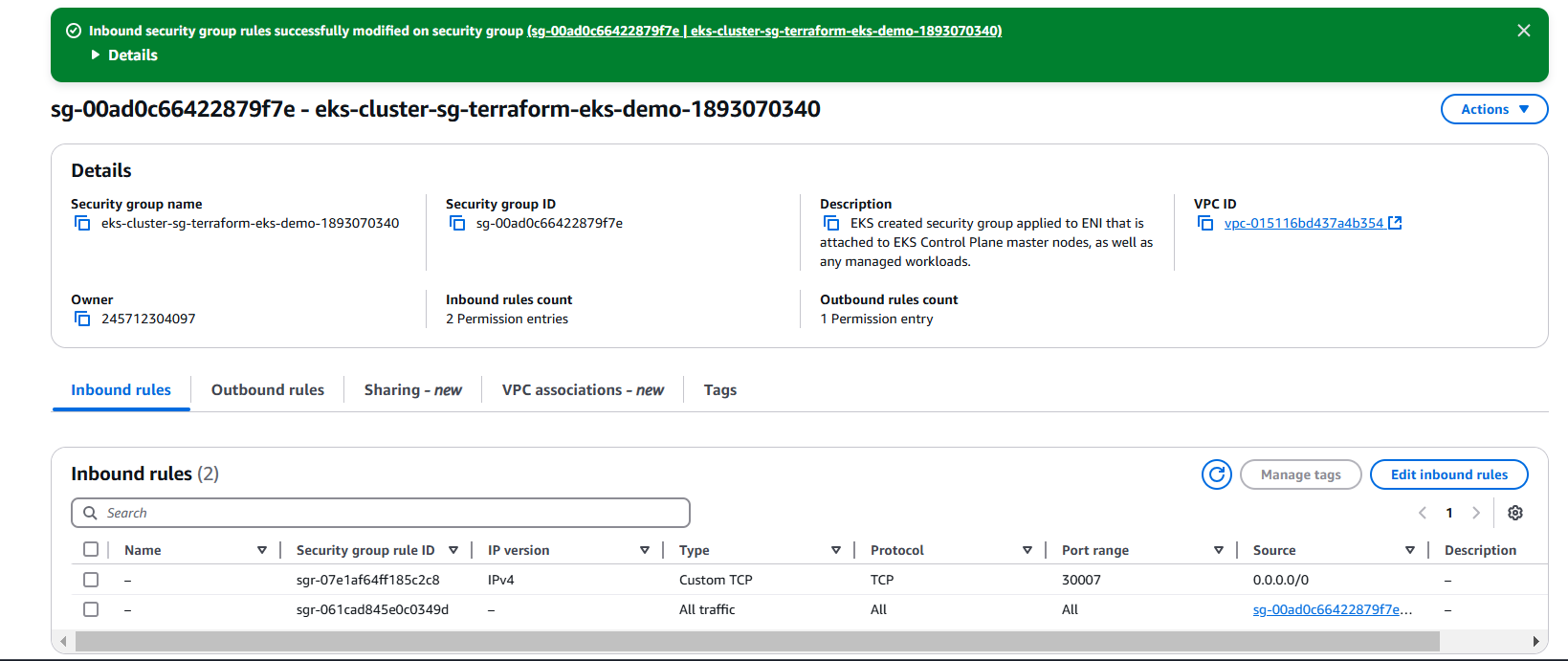




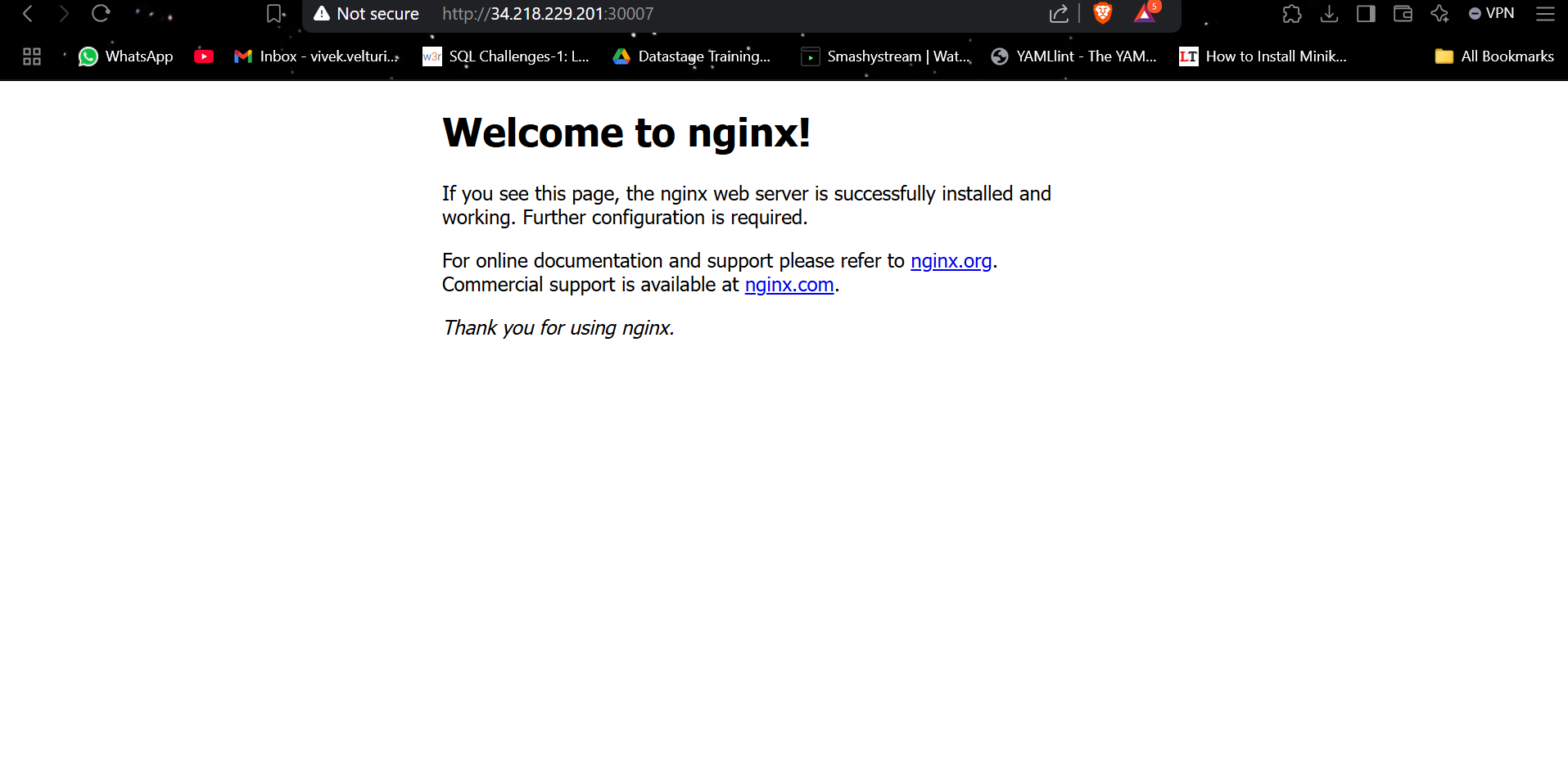
* In the NodePort script, we have specified port 80 for the host and port 30007 for the client. To verify the service, open your web browser and enter the public IP address of your node instance followed by :30007. For example, if your node instance's public IP is 34.218.229.201, you would enter http://34.218.229.201:30007 in the browser to check if the service is accessible.



* If the service is not accessible, it's likely because port 30007 has not been added to the security group associated with the node instance.
* Go to the AWS Management Console.
  + Open the **EC2 Dashboard** and locate the node instance created as part of your cluster.
* **Modify Security Group**:
* Select the **Security Group** attached to the node instance.
* Click on **Edit Inbound Rules** to modify the access rules.
* **Add Inbound Rule**:
* Add a new rule:
  + **Type**: Custom TCP
  + **Port Range**: 30007
  + **Source**: 0.0.0.0/0 (to allow access from any IP address).
* Save the changes.



* **Test the Service Again,** Open your browser and enter the public IP address of the node followed by :30007.  
  For example: http:// 34.218.229.201:30007.
* You should now be able to access the NGINX webpage.

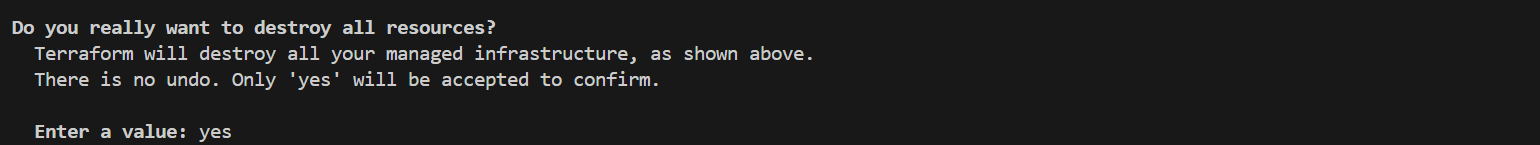


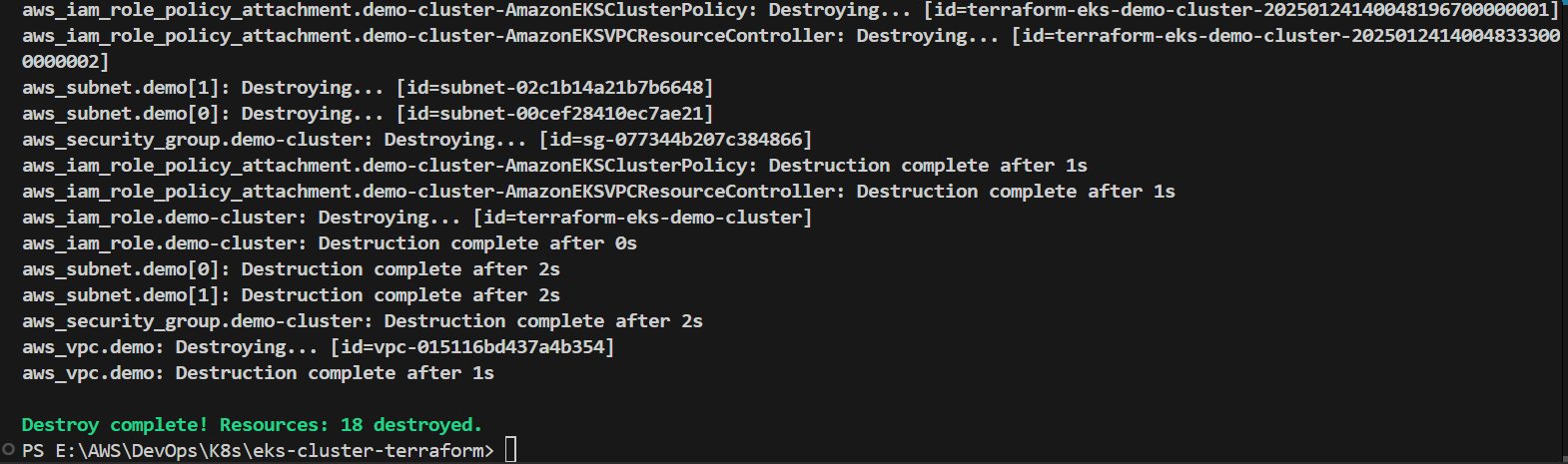
* To delete the EKS cluster and all associated resources created using Terraform, follow these steps:

1. Open the **terminal** in the same directory where you previously executed the Terraform commands (terraform init, terraform plan, and terraform apply).
2. Run the following command to start the deletion process: terraform destroy
3. Terraform will display a list of resources it plans to delete. Carefully review the plan, and when prompted, confirm the deletion by typing yes.

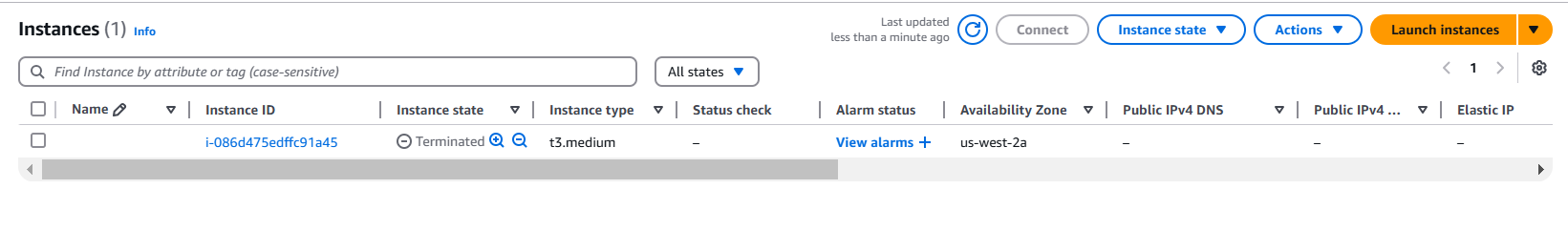


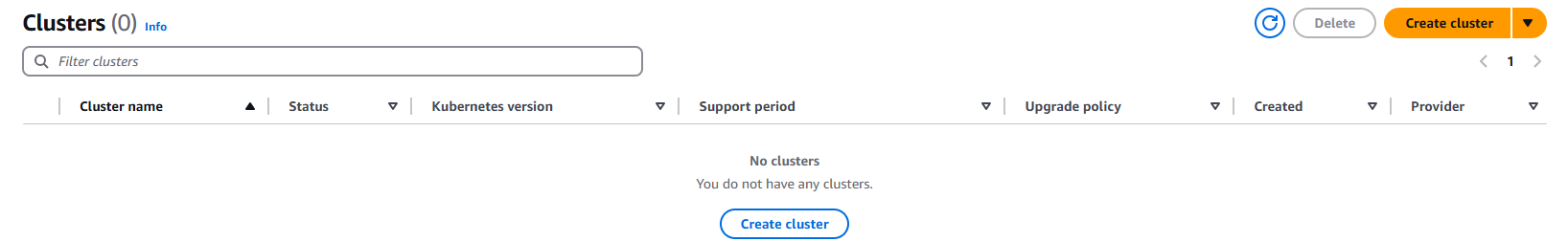






1. Wait for the process to complete. Terraform will remove all the resources it managed, including the EKS cluster, EC2 instances, and networking components.
2. After completion, you can verify the deletion by logging into the AWS Management Console and checking the **EKS** and **EC2** sections to ensure no resources remain.





This process ensures that all AWS resources created by your Terraform scripts are properly removed, avoiding unnecessary charges.