Ria Chaudhari
D15A07

# Kubernetes Cluster Management with Terraform
# Case Study

● Concepts Used: Kubernetes, Terraform, AWS CloudShell.
● Problem Statement: "Use Terraform to provision a Kubernetes cluster on AWS. Then,
use AWS CloudShell IDE to deploy a sample application on the cluster using kubectl."
● Tasks:
○ Write a Terraform script to create a Kubernetes cluster on AWS.
○ Use AWS CloudShell to configure kubectl for the newly created cluster.
○ Deploy a simple application (e.g., a Python Flask app) on the Kubernetes cluster and verify its deployment.

## Introduction

### Case Study Overview:

This case study focuses on the integration of Kubernetes and Terraform to manage and deploy cloud-based infrastructure on AWS. The goal is to explore how Terraform can be used to provision a Kubernetes cluster on AWS, and subsequently use AWS CloudShell to deploy a sample application on the cluster using kubectl. This process highlights the power of Infrastructure as Code (IaC) tools like Terraform and container orchestration platforms like Kubernetes for seamless, automated cloud infrastructure management.

### Key Feature and Application:

The unique feature of this case study is the use of Terraform to automate the provisioning of a Kubernetes cluster on AWS, streamlining the often complex task of setting up cloud infrastructure. The practical application of this feature lies in its ability to simplify infrastructure management, allowing DevOps engineers and cloud professionals to manage, scale, and deploy applications efficiently. By using Terraform and Kubernetes together, it ensures that the infrastructure is version-controlled, reproducible, and easy to maintain, while Kubernetes provides the scalability and reliability needed for modern application deployments.

## Step-by-Step Explanation

Prerequisites:

1.AWS Account with proper permissions

2.AWS CLI and kubectl installed in AWS CloudShell

3.Set up Terraform on your local machine or CloudShell

Step 1: Set up Terraform

1.Install Terraform on your local machine or use AWS CloudShell, which has Terraform pre-installed,also aws cli and kubectl are already preinstalled on CloudShell.

Verify terraform installation:

```
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$ terraform -v
Terraform v1.9.8
on linux_amd64
+ provider registry.terraform.io/hashicorp/aws v5.72.1
+ provider registry.terraform.io/hashicorp/cloudinit v2.3.5
+ provider registry.terraform.io/hashicorp/kubernetes v2.33.0
+ provider registry.terraform.io/hashicorp/time v0.12.1
+ provider registry.terraform.io/hashicorp/tls v4.0.6
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$
```

Verify aws installation:

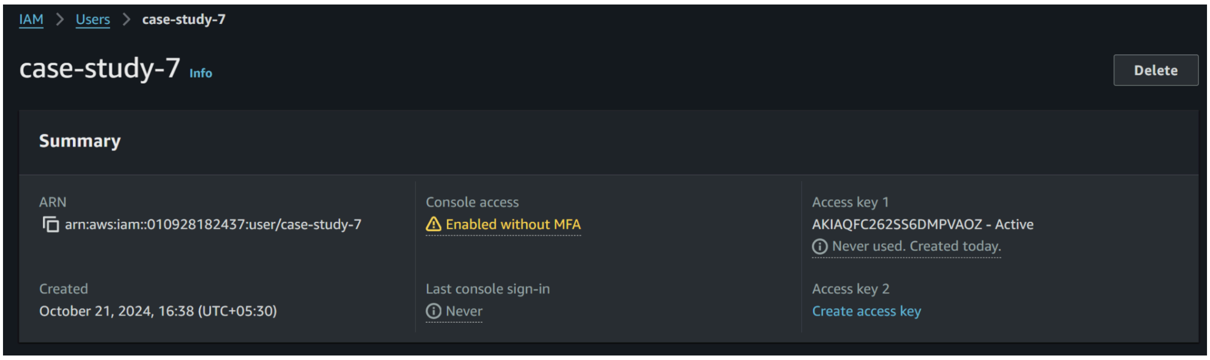
```
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$ aws --version
aws-cli/2.18.5 Python/3.12.6 Linux/6.1.109-118.189.amzn2023.x86_64 exec-env/CloudShell exe/x86_64.amzn.2023
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$
```

Verify kubectl installation:

```
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$ kubectl version --client
Client Version: v1.30.2-eks-1552ad0
Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3
```

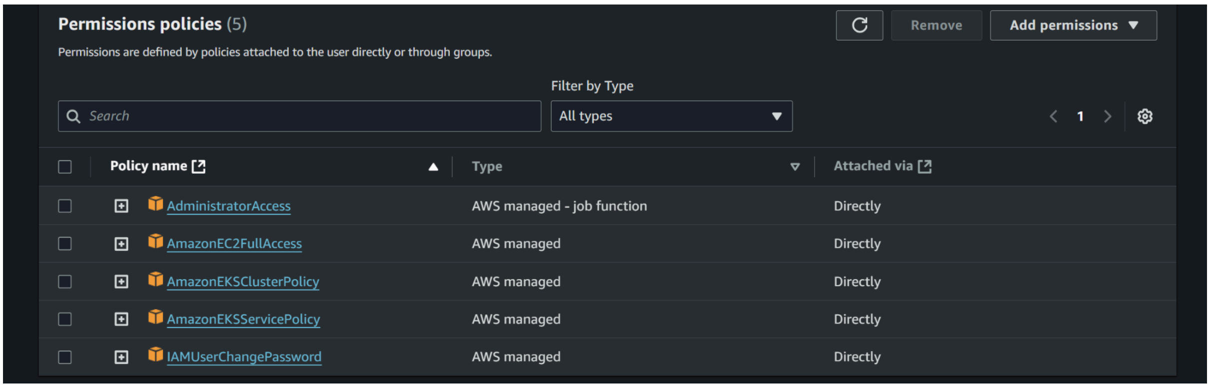
Step 2:Configure AWS credentials:
If using local machine: Set up AWS CLI and run `aws configure`
If using CloudShell: Credentials are automatically configured

To get the access key and secret access key,first create an IAM User

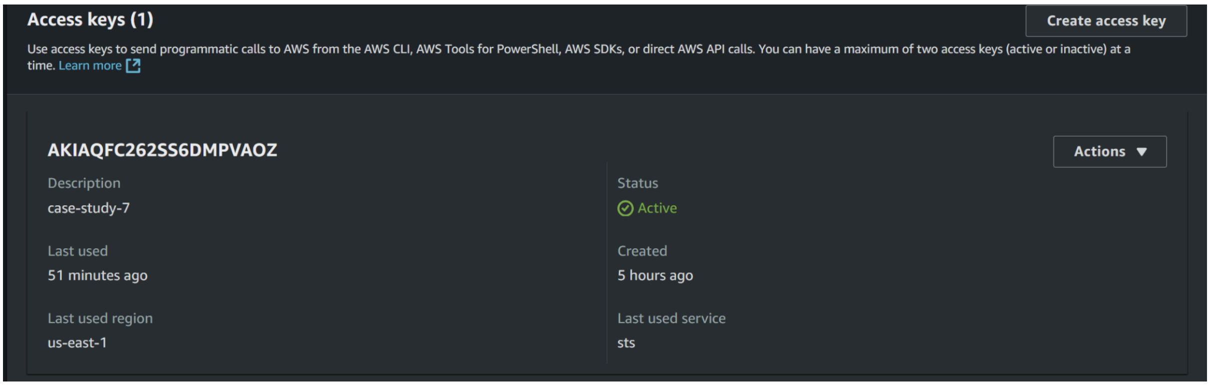



Then give the following permissions to the User

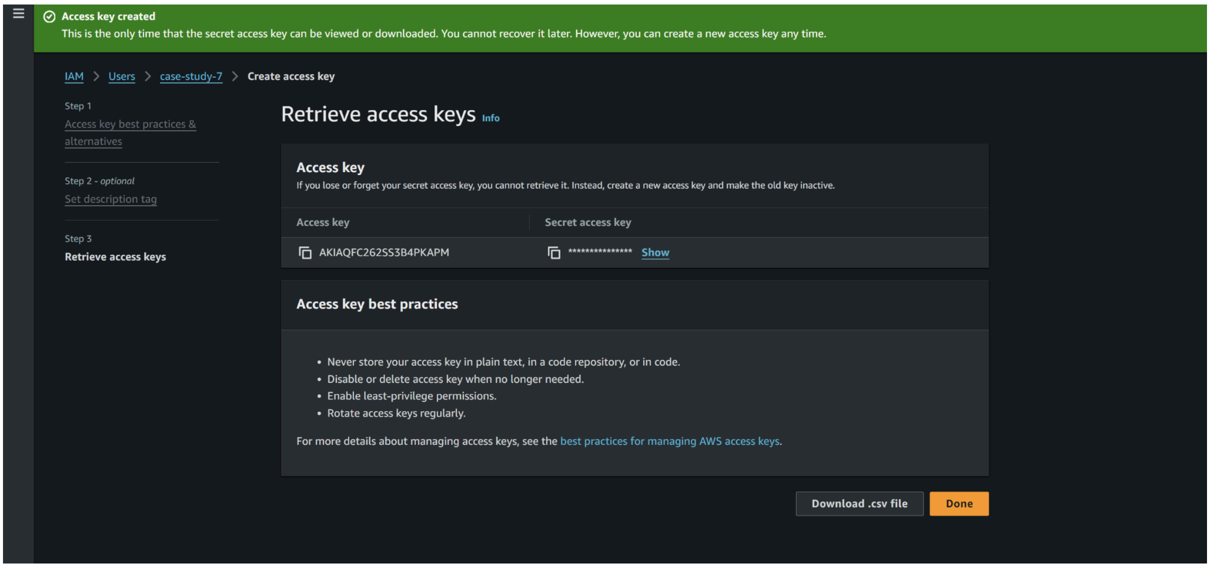
- AdministratorAccess
- AmazonEC2FullAccess
- AmazonEKSClusterPolicy
- AmazonEKSServicePolicy




Then click on create access key



You can download .csv file which contains both access key and secret access key




| Access key ID | Secret access key |
|---|---|
| AKIAQFC262SS6DMPVAOZ | /9o0lGY2ujdyekwyTYz8iOqPk4Lx2Oygh9QtABMz |

For aws configure enter your access key,secret access key,default region name(us-east-1) and default output format(json)

```
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$ aws configure
AWS Access Key ID [****************VAOZ]:  AKIAQFC262SS6DMPVAOZ
AWS Secret Access Key [****************ABMz]: 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
Default region name [us-east-1]: us-east-1
Default output format [json]: json
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$
```

Step 3: Create a new directory for your Terraform project:
mkdir k8s-terraform && cd k8s-terraform

```
[cloudshell-user@ip-10-130-74-195 ~]$ mkdir k8s-terraform
mkdir: cannot create directory ‘k8s-terraform’: File exists
[cloudshell-user@ip-10-130-74-195 ~]$ cd k8s-terraform
```

Step 4:Create a file named `main.tf` with the following content

```
  GNU nano 5.8                                main.tf
provider "aws" {
  region = "us-east-1"  # or your preferred region
}

module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "~> 19.0"

  cluster_name    = "my-eks-cluster"
  cluster_version = "1.27"

  vpc_id     = module.vpc.vpc_id
  subnet_ids = module.vpc.private_subnets
  cluster_endpoint_public_access = true

  eks_managed_node_group_defaults = {
    ami_type       = "AL2_x86_64"
    instance_types = ["t3.medium"]
  }

  eks_managed_node_groups = {
    example = {
      min_size     = 1
      max_size     = 3
      desired_size = 2
    }
  }
}

module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 4.0"

  name = "my-vpc"
  cidr = "10.0.0.0/16"

  azs             = ["us-east-1a", "us-east-1b", "us-east-1c"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  public_subnets  = ["10.0.4.0/24", "10.0.5.0/24", "10.0.6.0/24"]

  enable_nat_gateway   = true
```

```
  single_nat_gateway   = true
  enable_dns_hostnames = true
}

output "cluster_endpoint" {
  description = "Endpoint for EKS control plane"
  value       = module.eks.cluster_endpoint
}

output "cluster_name" {
  description = "Kubernetes Cluster Name"
  value       = module.eks.cluster_name
}
```

## Step 5: Initialize and Apply Terraform Configuration

```
[cloudshell-user@ip-10-134-33-25 k8s-terraform]$ terraform init
Initializing the backend...
Initializing modules...
Initializing provider plugins...
- Reusing previous version of hashicorp/tls from the dependency lock file
- Reusing previous version of hashicorp/kubernetes from the dependency lock file
- Reusing previous version of hashicorp/time from the dependency lock file
- Reusing previous version of hashicorp/cloudinit from the dependency lock file
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/tls v4.0.6
- Using previously-installed hashicorp/kubernetes v2.33.0
- Using previously-installed hashicorp/time v0.12.1
- Using previously-installed hashicorp/cloudinit v2.3.5
- Using previously-installed hashicorp/aws v5.72.1

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
[cloudshell-user@ip-10-134-33-25 k8s-terraform]$
```

```
module.eks.data.tls_certificate.this[0]: Read complete after 0s [id=99d41e43229a4cdaf4141f3e8310e6d95c31dab9]
module.eks.aws_iam_openid_connect_provider.oidc_provider[0]: Creating...
module.eks.aws_iam_openid_connect_provider.oidc_provider[0]: Creation complete after 0s [id=arn:aws:iam::010928182437:oidc-provider/oidc.eks.us-east-1.amazonaws.com/id/102A4A2BA146EEB20F86461453A362A8]
module.eks.time_sleep.this[0]: Still creating... [10s elapsed]
module.eks.time_sleep.this[0]: Still creating... [20s elapsed]
module.eks.time_sleep.this[0]: Still creating... [30s elapsed]
module.eks.time_sleep.this[0]: Creation complete after 30s [id=2024-10-21T11:27:57Z]
module.eks.module.eks_managed_node_group["example"].aws_launch_template.this[0]: Creating...
module.eks.module.eks_managed_node_group["example"].aws_launch_template.this[0]: Creation complete after 1s [id=lt-0048309348a5ced5a]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Creating...
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [10s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [20s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [30s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [40s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [50s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [1m0s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [1m10s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [1m20s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [1m30s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [1m40s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [1m50s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [2m0s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [2m10s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [2m20s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [2m30s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Still creating... [2m40s elapsed]
module.eks.module.eks_managed_node_group["example"].aws_eks_node_group.this[0]: Creation complete after 2m48s [id=my-eks-cluster:example-20241021112757577100000011]

  Warning: Argument is deprecated

    with module.eks.aws_iam_role.this[0],
    on .terraform/modules/eks/main.tf line 293, in resource "aws_iam_role" "this":
   293: resource "aws_iam_role" "this" {

  The inline_policy argument is deprecated. Use the aws_iam_role_policy resource instead. If Terraform should exclusively manage all inline policy associations (the current behavior of this argument), use the
  aws_iam_role_policies_exclusive resource as well.

  (and one more similar warning elsewhere)

Apply complete! Resources: 53 added, 0 changed, 0 destroyed.
```

Verify whether the cluster is active or not:

aws eks describe-cluster –name my-eks-cluster –query cluster.status

```
ip-10-0-2-51.ec2.internal    Ready    <none>    4h38m   v1.27.16-eks-a737599
[cloudshell-user@ip-10-134-33-25 k8s-terraform]$ aws eks describe-cluster --name my-eks-cluster --query cluster.status
"ACTIVE"
[cloudshell-user@ip-10-134-33-25 k8s-terraform]$
```

View cluster info

Kubectl cluster-info

```
[cloudshell-user@ip-10-134-33-25 k8s-terraform]$ kubectl cluster-info
Kubernetes control plane is running at https://102A4A2BA146EEB20F86461453A362A8.yl4.us-east-1.eks.amazonaws.com
CoreDNS is running at https://102A4A2BA146EEB20F86461453A362A8.yl4.us-east-1.eks.amazonaws.com/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

Verify VPC configuration:
aws eks describe-cluster –name my-eks-cluster –query cluster.resourcesVpcConfig

```
[cloudshell-user@ip-10-134-33-25 k8s-terraform]$ aws eks describe-cluster --name my-eks-cluster --query cluster.resourcesVpcConfig
{
    "subnetIds": [
        "subnet-03380a7a38b20b144",
        "subnet-0dd85c613f7acd534",
        "subnet-0a1ad2bf44847168b"
    ],
    "securityGroupIds": [
        "sg-0af524cf736da28b1"
    ],
    "clusterSecurityGroupId": "sg-022fbf03ce96a2c14",
    "vpcId": "vpc-0781f7448c2eb48f2",
    "endpointPublicAccess": true,
    "endpointPrivateAccess": true,
    "publicAccessCidrs": [
        "0.0.0.0/0"
    ]
}
[cloudshell-user@ip-10-134-33-25 k8s-terraform]$
```

Step 6: Configure kubectl in AWS CloudShell
1.Open AWS CloudShell in the AWS Management Console.
2.Update the kubeconfig for your new cluster:

```
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$ aws eks update-kubeconfig --name my-eks-cluster --region us-east-1
Added new context arn:aws:eks:us-east-1:010928182437:cluster/my-eks-cluster to /home/cloudshell-user/.kube/config
```

3.Verify the connection
kubectl get nodes

```
[cloudshell-user@ip-10-134-33-25 k8s-terraform]$ kubectl get nodes
NAME                        STATUS   ROLES    AGE    VERSION
ip-10-0-1-152.ec2.internal  Ready    <none>   4h38m  v1.27.16-eks-a737599
ip-10-0-2-51.ec2.internal   Ready    <none>   4h38m  v1.27.16-eks-a737599
```

Step 7: Deploy a Sample Application
1.Create a file named `flask-app.yaml` with the following content

```
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$ nano flask-app.yaml
```

```
CloudShell                                                          Actions ▼
us-east-1  +
  GNU nano 5.8                         flask-app.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: flask-app
  template:
    metadata:
      labels:
        app: flask-app
    spec:
      containers:
      - name: flask-app
        image: tiangolo/uwsgi-nginx-flask:python3.8
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: flask-app-service
spec:
  selector:
    app: flask-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer

                                   [ Read 32 lines ]
```

2.Deploy the application
kubectl apply -f flask-app.yaml

```
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$ kubectl apply -f flask-app.yaml
deployment.apps/flask-app created
service/flask-app-service created
```

3.Check the deployment status
kubectl get deployments
kubectl get services

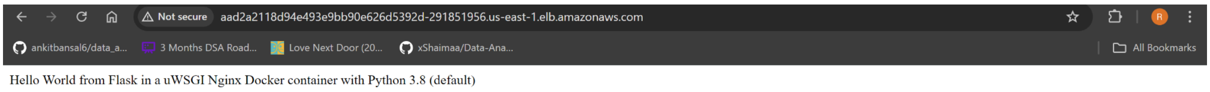
```
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$ kubectl get deployments
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
flask-app   2/2     2            2           34s
```

```
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$ kubectl get services
NAME                TYPE           CLUSTER-IP      EXTERNAL-IP                                                                PORT(S)        AGE
flask-app-service   LoadBalancer   172.20.154.66   aad2a2118d94e493e9bb90e626d5392d-291851956.us-east-1.elb.amazonaws.com   80:31647/TCP   46s
kubernetes          ClusterIP      172.20.0.1      <none>                                                                     443/TCP        29m
```

4.Once the LoadBalancer service is provisioned, get the external IP
kubectl get services flask-app-service

```
[cloudshell-user@ip-10-140-121-154 k8s-terraform]$ kubectl get services flask-app-service
NAME                TYPE           CLUSTER-IP      EXTERNAL-IP                                                                PORT(S)        AGE
flask-app-service   LoadBalancer   172.20.154.66   aad2a2118d94e493e9bb90e626d5392d-291851956.us-east-1.elb.amazonaws.com   80:31647/TCP   78s
```

Step 8: View the deployed application




## Guidelines: Best Practices for Kubernetes Cluster Management with Terraform

To ensure the Kubernetes cluster is provisioned and managed effectively, here are several best practices and additional guidelines that users should follow during the setup, deployment, and management process:

### 1. Infrastructure as Code (IaC) Best Practices

- **Version Control with Git:** Store your Terraform configuration files (e.g., `main.tf`, `variables.tf`) in a Git repository. This enables version control and allows you to track changes, collaborate with team members, and roll back to previous versions if necessary.

2. **Monitoring and Scaling**

- **Enable Cluster Monitoring:** Use CloudWatch and AWS Kubernetes Monitoring to track performance metrics and logs. This allows you to monitor the health of your cluster and take action if issues arise.
- **Auto-scaling for Worker Nodes:** Configure auto-scaling for the Kubernetes worker nodes to ensure your application can handle increased traffic without manual intervention. AWS Auto Scaling Groups can be used to automatically adjust the number of EC2 instances based on load.

```hcl
scaling_config {
  desired_size = 2
  max_size     = 5
  min_size     = 1
}
```

**3.Cost Management**

- **Resource Cleanup:** After testing or deploying applications, ensure that unused or obsolete resources (e.g., EC2 instances, EKS clusters) are destroyed to avoid unnecessary costs. Always run `terraform destroy` when you no longer need the infrastructure.
- **Spot Instances:** For non-production workloads, consider using EC2 Spot Instances to reduce costs. However, be mindful of the volatility associated with Spot Instances.

**4.Post-Deployment Best Practices**

- **Regular Backups:** Periodically back up both your Terraform state and Kubernetes configuration files. This helps ensure that you can restore the infrastructure and configurations in case of failure or accidental changes.
- **Disaster Recovery Planning:** Set up disaster recovery protocols by replicating the cluster across multiple availability zones (AZs) within AWS. This will ensure high availability in case of AZ failure.
- **Test Changes in Staging:** Before applying major changes to production, test them in a staging environment. This helps catch issues early and prevents downtime in production environments

## Demonstration Preparation

### Key Points to Focus On During the Demonstration

1. **Introduction to the Project:**
    - Briefly introduce the purpose of the demonstration, outlining the goal of provisioning a Kubernetes cluster on AWS using Terraform and deploying a sample application.
2. **Environment Setup:**
    - Explain the choice of using AWS CloudShell for the experiment, highlighting its advantages (e.g., pre-installed tools, accessible interface).
    - Discuss the configuration of AWS credentials and the importance of having the correct permissions.
3. **Terraform Configuration:**
    - Walk through the main components of the Terraform scripts, including:
        - Provider configuration for AWS.
        - Resource definitions for the EKS cluster, VPC, subnets, and node groups.
        - Use of variables for flexibility and maintainability.
4. **Provisioning the Cluster:**
    - Demonstrate the `terraform init`, `terraform plan`, and `terraform apply` commands:
        - Explain what each command does and how it contributes to the overall provisioning process.
        - Highlight the importance of reviewing the execution plan before applying changes.
5. **Configuring kubectl:**
    - Show how to configure `kubectl` to interact with the newly created EKS cluster using AWS CLI commands.
    - Explain the significance of setting the Kubernetes context.
6. **Deploying the Sample Application:**
    - Provide a live demo of deploying a simple application (e.g., a Python Flask app) on the Kubernetes cluster using `kubectl`.
    - Emphasize how to verify the deployment, check pod status, and access application logs.
7. **Monitoring and Troubleshooting:**
    - Discuss methods for monitoring the cluster and application, including using `kubectl` commands to check resources and troubleshoot any issues.
8. **Resource Cleanup:**

    - Highlight the importance of cleaning up resources after the demonstration to avoid unnecessary costs, showcasing the `terraform destroy` command.

**Importance of Practicing the Demonstration**

- **Familiarity with the Process:** Practicing the demonstration multiple times allows you to become familiar with the workflow and troubleshoot any potential issues beforehand. This preparation can prevent interruptions during the live demonstration.
- **Timing and Pacing:** Practicing helps you gauge the timing of each section, ensuring you stay within the allotted time for the demonstration. This helps maintain audience engagement and keeps the presentation concise.
- **Confidence Building:** Rehearsing the demonstration boosts your confidence in presenting the material. Familiarity with the tools and processes allows you to present more effectively and handle questions with ease.
- **Refining Delivery:** Practicing helps you refine your explanations and delivery, making your points clearer and easier to understand for the audience. This ensures you can convey complex concepts in a digestible manner.
- **Simulating Real Scenarios:** By practicing, you can simulate real-world scenarios and issues that might arise during the demonstration. This allows you to prepare responses and solutions, enhancing your credibility and expertise.

## Conclusion

This case study demonstrates the process of provisioning a Kubernetes cluster on AWS using Terraform and deploying a sample Python Flask application using AWS CloudShell. It highlights the automation of cloud infrastructure management through Infrastructure as Code (IaC) with Terraform, the configuration of `kubectl` for cluster interaction, and the deployment of a containerized application, showcasing the synergy of these technologies for efficient and scalable application management in a cloud environment.