Introduction to EKS: Setting up an EKS cluster, Deploying applications on EKS. Writing basic Terraform scripts, Deploying infrastructure on AWS.

CI/CD for EKS using Jenkins/GitHub Actions, setting up Jenkins or GitHub Actions for Kubernetes deployments, Automating deployments to EKS

Introduction to Amazon EKS (Elastic Kubernetes Service)

1. Overview

Amazon Elastic Kubernetes Service (EKS) is a managed Kubernetes service provided by **AWS** that allows you to run Kubernetes clusters without managing the underlying control plane or master nodes.

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. With **EKS**, AWS handles the heavy lifting such as:

- Setting up and maintaining the Kubernetes control plane.
- Automatically patching and updating Kubernetes.
- Ensuring high availability and scalability.

EKS can run workloads across **AWS**, **on-premises (using EKS Anywhere)**, and even **hybrid environments**.

2. Key Components of EKS

- **Control Plane**: Managed by AWS (includes the API server, etcd, scheduler, and controller manager).
- **Worker Nodes**: EC2 instances or AWS Fargate tasks that run your application pods.
- **Node Groups**: A group of EC2 instances managed by an Auto Scaling Group.
- **Kubelet**: An agent running on worker nodes that communicates with the control plane.
- **kubectl**: Command-line tool used to interact with the Kubernetes API.
- **EKS Add-ons**: Optional components such as CoreDNS, kube-proxy, and Amazon VPC CNI plugin.

3. Advantages of EKS

Feature	Description
Fully Managed	AWS manages the Kubernetes control plane, including scalability and patching.
Scalable	Automatically scales based on workloads using Cluster Autoscaler and HPA.
Secure	Integrated with AWS IAM, VPC, and KMS for fine-grained access control and encryption.
Highly Available	EKS runs across multiple Availability Zones for resilience.
Integrations	Integrates with AWS services such as CloudWatch, IAM, ELB, and Route 53.

Setting up an Amazon EKS Cluster – Step-by-Step Explanation

Step 1: Prerequisites

Before you start, make sure you have the following:

- **AWS Account**
- **IAM Permissions** (AdministratorAccess or equivalent)
- **AWS CLI** installed
- aws --version
- **kubectl** installed (to manage Kubernetes)
- kubectl version --client
- **eksctl** installed (a simple CLI tool for EKS)
- eksctl version

Step 2: Create an IAM Role for EKS

EKS requires IAM roles for both the **EKS control plane** and **worker nodes**.

1. **Go to the AWS Management Console → IAM → Roles → Create Role**
2. Choose **EKS** as the use case → Select **EKS Cluster** → Attach policy:
    - AmazonEKSClusterPolicy
3. Name the role (e.g., eksClusterRole) and create it.

You will later create another role for **node groups** (worker nodes).

Step 3: Create a VPC for EKS

You can either:

- Use the **default VPC**, or
- Create a **new VPC** with public and private subnets.

Using eksctl, you can let it create one automatically, or you can manually define it in AWS VPC console.

EKS needs at least **two subnets** in different Availability Zones for high availability.

Step 4: Create the EKS Cluster (using eksctl)

The easiest method to create an EKS cluster is with the **eksctl** tool.

Command:

```
eksctl create cluster \
  --name my-eks-cluster \
  --region us-east-1 \
  --nodegroup-name my-nodes \
  --node-type t3.medium \
  --nodes 2 \
  --nodes-min 1 \
  --nodes-max 3 \
  --managed
```

Explanation of parameters:

- --name: Name of your EKS cluster
- --region: AWS region
- --node-type: EC2 instance type for worker nodes
- --nodes: Desired number of worker nodes
- --managed: Creates a managed node group (AWS manages lifecycle of nodes)

This command automatically:

- Creates the EKS cluster control plane
- Creates VPC, subnets, and security groups
- Launches EC2 worker nodes
- Connects nodes to the cluster

Step 5: Configure kubectl for the Cluster

After creation, configure kubectl to connect to the new EKS cluster:

```
aws eks --region us-east-1 update-kubeconfig --name my-eks-cluster
```

To verify the cluster connection:

```
kubectl get svc
```

If successful, you'll see the default Kubernetes services (like kubernetes in the default namespace).

Step 6: Verify Node Group

Check that your EC2 nodes (worker nodes) are registered and ready:

```
kubectl get nodes
```

You should see your nodes in the **Ready** state.

Step 7: Deploy a Test Application

Let's test the cluster with a sample Nginx deployment.

Create a deployment:

kubectl create deployment nginx --image=nginx

Expose it as a service:

kubectl expose deployment nginx --port=80 --type=LoadBalancer

Check the service:

kubectl get svc

After a few minutes, you'll see an **EXTERNAL-IP** — open it in your browser to see the Nginx welcome page.

Step 8: (Optional) Manage the Cluster Using AWS Console

- Go to **AWS Console → EKS → Clusters → my-eks-cluster**
- You can view **nodes, workloads, networking, and logs**
- You can also integrate with **CloudWatch**, **ECR**, or **IAM roles for service accounts**

Step 9: Clean Up Resources (Optional)

To avoid unnecessary charges:

eksctl delete cluster --name my-eks-cluster --region us-east-1

This deletes the EKS cluster and all related AWS resources.

Deploying Applications on Amazon EKS

Once your EKS cluster is set up and configured, you can deploy applications to it using **Kubernetes manifests (YAML files)** or **Helm charts**. Below is a detailed, real-world example of deploying a containerized application on EKS.

Step 1: Prerequisites

Before deploying, ensure you have:

- A running **EKS cluster**

- **kubectl** configured and connected to the cluster
- A **Docker image** of your application (stored in **Amazon ECR** or Docker Hub)

Check connection to EKS:

kubectl get nodes

If you see worker nodes in the **Ready** state, you're good to go.

Step 2: Create a Docker Image and Push to ECR

1. **Build the Docker image:**
2. docker build -t myapp .
3. **Tag the image for ECR:**
4. docker tag myapp:latest <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/myapp:latest
5. **Login to ECR:**
6. aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com
7. **Push image to ECR:**
8. docker push <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/myapp:latest

Now your application image is stored in ECR and ready for deployment.

Step 3: Create a Kubernetes Deployment File

Create a file called **deployment.yaml** that defines how your application should run in the cluster.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
spec:
  replicas: 2
  selector:
    matchLabels:
```

```
    app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/myapp:latest
        ports:
        - containerPort: 8080
```

Explanation:

- replicas: 2 → Runs 2 pods for high availability.
- image: → Refers to your Docker image in ECR.
- containerPort: → Exposes port 8080 inside the container.

Step 4: Create a Kubernetes Service File

To expose the application, create **service.yaml**:

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: LoadBalancer
  selector:
    app: myapp
  ports:
  - port: 80
    targetPort: 8080
```

Explanation:

- type: LoadBalancer → Automatically provisions an AWS ELB (Elastic Load Balancer).
- selector: → Links the service to pods with the same label.
- port and targetPort → Define external and internal ports.

Step 5: Deploy to EKS

Apply the YAML files to your cluster:

kubectl apply -f deployment.yaml

kubectl apply -f service.yaml

Check the deployment status:

kubectl get deployments

kubectl get pods

Check the service to get the public endpoint:

kubectl get svc

After a few minutes, you'll see an **EXTERNAL-IP** under the service — copy that IP or DNS name into your browser to access your app.

Step 6: Verify Deployment

- Check if pods are running:
- kubectl get pods
- Describe pod logs:
- kubectl logs <pod-name>
- Access the application using the **LoadBalancer URL** shown earlier.

You should see your application running successfully on EKS.

Step 7: Update or Roll Out a New Version

To deploy a new version of your app:

1. Update your code → build and push a new image version to ECR.
2. Update the image tag in deployment.yaml.
3. Apply the changes:
4. kubectl apply -f deployment.yaml

Kubernetes performs a **rolling update** — ensuring zero downtime by updating pods gradually.

Step 8: Scaling the Application

You can scale your app manually or automatically.

Manual Scaling:

kubectl scale deployment myapp-deployment --replicas=4

Auto Scaling (Optional):
Enable Horizontal Pod Autoscaler (HPA):

kubectl autoscale deployment myapp-deployment --min=2 --max=6 --cpu-percent=70

Step 9: Monitor and Manage the Application

- Use **CloudWatch** and **Prometheus** for metrics and logs.
- Use kubectl describe pod and kubectl top pods for resource monitoring.
- Integrate with **AWS CloudWatch Container Insights** for deeper visibility.

Step 10: Clean Up Resources (Optional)

When done, delete the application:

kubectl delete -f service.yaml

kubectl delete -f deployment.yaml

Or delete all resources in one go:

kubectl delete all -l app=myapp

Writing Basic Terraform Scripts

1. What is Terraform?

Terraform is an **Infrastructure as Code (IaC)** tool created by **HashiCorp**.
It allows you to **create and manage cloud resources** (like AWS EC2 instances, VPCs, S3 buckets, etc.) using simple configuration files written in **HCL (HashiCorp Configuration Language)**.

Instead of clicking through the AWS Console, Terraform automates infrastructure creation using code.

2. Basic Steps in Using Terraform

Step	Command	Description
1	terraform init	Initialize Terraform and download provider plugins
2	terraform plan	Preview the resources Terraform will create
3	terraform apply	Apply the configuration and create resources
4	terraform destroy	Delete the created resources

3. Basic Structure of a Terraform Script

A Terraform project usually has these files:

- **provider.tf** → Defines the cloud provider (e.g., AWS, Azure, GCP).
- **main.tf** → Contains the resources to create (like EC2 instances).
- **outputs.tf** → Displays useful outputs (like public IP).
- **variables.tf** → (Optional) Stores input variables.

4. Example: Create an EC2 Instance on AWS

Here's a **simple example** of a Terraform script that creates a virtual machine (EC2 instance) on AWS.

provider.tf

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

This tells Terraform to use **AWS** and deploy resources in the **US East (N. Virginia)** region.

main.tf

```
resource "aws_instance" "my_ec2" {
  ami           = "ami-0c55b159cbfafe1f0"   # Amazon Linux 2 AMI
  instance_type = "t2.micro"

  tags = {
    Name = "TerraformExampleInstance"
```

```
  }
}
```

✅ This code creates a **t2.micro EC2 instance** using an **Amazon Linux 2** image.
Tags help identify the instance in the AWS console.

outputs.tf *(optional)*

```
output "instance_public_ip" {
  value = aws_instance.my_ec2.public_ip
}
```

✅ This prints the **public IP address** of your instance after creation.

5. Commands to Run

1. **Initialize Terraform:**
2. terraform init
3. **Preview the Plan:**
4. terraform plan
5. **Apply the Configuration:**
6. terraform apply

Type **yes** when prompted. Terraform will now create your EC2 instance.

7. **Check Output:**
   After the apply finishes, Terraform will show:
8. Outputs:
9. instance_public_ip = "3.95.120.12"
10. **Destroy Resources (when no longer needed):**
11. terraform destroy

6. Explanation of Key Terms

- **Provider:** Defines the platform (AWS, Azure, etc.) where resources will be created.
- **Resource:** A specific infrastructure component (like EC2, S3, VPC).
- **State File:** Terraform keeps track of resources it manages in a terraform.tfstate file.
- **Declarative Language:** You describe *what* you want, and Terraform figures out *how* to create it.

7. Advantages of Terraform

- Automates cloud resource creation.
- Ensures consistent setups across environments.
- Can version-control infrastructure code (e.g., via Git).
- Easy to scale and manage.

Deploying Infrastructure on AWS – Simple Explanation

1. What is Infrastructure Deployment?

Deploying infrastructure on AWS means **creating and setting up cloud resources** — such as servers, networks, storage, and databases — on **Amazon Web Services** to run your applications.

Instead of building physical servers, AWS allows you to **provision everything virtually** using the AWS Management Console, CLI, or automation tools like Terraform and CloudFormation.

2. Ways to Deploy Infrastructure on AWS

You can deploy AWS infrastructure in **three main ways:**

1. **AWS Management Console:**
    - A web-based interface for manually creating and managing resources.
    - Example: Launch an EC2 instance by selecting an AMI, instance type, and network settings.
2. **AWS Command Line Interface (CLI):**
    - Allows you to create and manage resources using terminal commands.
    - Example:
    - aws ec2 run-instances --image-id ami-0c55b159cbfafe1f0 --instance-type t2.micro
3. **Infrastructure as Code (IaC) Tools:**
    - Tools like **Terraform**, **AWS CloudFormation**, and **Ansible** let you define infrastructure using code files.
    - This approach is **automated, repeatable, and scalable**.

3. Common AWS Services Used in Infrastructure Deployment

Service	Purpose
Amazon EC2	Virtual servers for running applications
Amazon S3	Object storage for data and backups
Amazon VPC	Networking and security configuration
Amazon RDS	Managed databases
Elastic Load Balancer (ELB)	Distributes traffic between servers
IAM	Access control and permissions

4. Example: Deploying an EC2 Instance (Simple Steps)

Here's a basic example of deploying a virtual machine on AWS:

1. **Login to AWS Console.**
2. **Go to EC2 Service.**
3. **Click "Launch Instance."**
4. **Choose an AMI (Amazon Linux 2).**
5. **Select Instance Type (t2.micro).**
6. **Configure Network (VPC, subnet).**
7. **Add Storage and Security Group (allow SSH and HTTP).**
8. **Launch and Connect** using SSH:
9. ssh -i mykey.pem ec2-user@<public-ip>

Your EC2 instance is now running — that means you've successfully deployed basic infrastructure on AWS.

5. Example: Deploying Infrastructure Using Terraform

You can also automate this using a Terraform script:

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

resource "aws_instance" "my_ec2" {
  ami           = "ami-0c55b159cbfafe1f0"
```

```
  instance_type = "t2.micro"
  tags = {
    Name = "TerraformInstance"
  }
}
```

Commands:

terraform init

terraform apply

✅ Terraform automatically deploys the EC2 instance on AWS.

6. Benefits of Deploying Infrastructure on AWS

- **Scalability:** Easily scale up or down as demand changes.
- **Cost Efficiency:** Pay only for what you use.
- **Automation:** Use IaC tools for faster and consistent deployments.
- **High Availability:** Use multiple regions and availability zones.
- **Security:** Manage permissions with IAM and private networks using VPC.

Creating an Amazon EKS Cluster using eksctl or AWS Management Console

1. What is Amazon EKS?

Amazon **EKS (Elastic Kubernetes Service)** is a managed service by AWS that makes it easy to run **Kubernetes** (an open-source container orchestration tool) on AWS.

With EKS, you can deploy, manage, and scale containerized applications without worrying about setting up or maintaining the Kubernetes control plane.

2. Two Common Ways to Create an EKS Cluster

You can create an EKS cluster in **two simple ways:**

1. Using **eksctl** (a command-line tool)
2. Using the **AWS Management Console** (a web interface)

🧰 Method 1: Creating an EKS Cluster using eksctl (Command Line Tool)

Step 1: Install Prerequisites

Make sure you have these installed on your system:

- **AWS CLI** – to connect to your AWS account
- **kubectl** – to manage Kubernetes resources
- **eksctl** – to create and manage EKS clusters

Check versions:

aws --version

kubectl version --client

eksctl version

Step 2: Create the Cluster

Use one simple command to create your EKS cluster:

```
eksctl create cluster \
  --name my-eks-cluster \
  --region us-east-1 \
  --nodegroup-name my-nodes \
  --node-type t3.medium \
  --nodes 2 \
  --managed
```

What this does:

- --name → Sets your cluster name
- --region → Chooses the AWS region
- --nodegroup-name → Names your worker node group
- --node-type → Chooses EC2 instance type for worker nodes
- --nodes → Number of nodes to create
- --managed → Uses AWS-managed node groups

This command automatically:

- Creates the EKS control plane
- Launches EC2 worker nodes
- Sets up networking (VPC, subnets, security groups)

Step 3: Configure kubectl

After the cluster is created, connect kubectl to it:

aws eks --region us-east-1 update-kubeconfig --name my-eks-cluster

Now check your cluster:

kubectl get nodes

You should see your worker nodes listed — meaning the EKS cluster is ready!

Setting Up CI/CD Pipeline for EKS using Jenkins

This pipeline automates building, testing, and deploying your application to an **Amazon EKS cluster**.

Prerequisites

1. **AWS Account** with EKS cluster already created.
2. **Jenkins installed** (can be on EC2, Docker, or local).
3. **kubectl installed** on Jenkins server.
4. **Docker installed** on Jenkins server.
5. **IAM user/role** with permissions for EKS, ECR, and S3.
6. **Git repository** with your application and Kubernetes manifest files.

Step 1: Install Jenkins Plugins

- Go to **Manage Jenkins → Manage Plugins**
- Install these plugins:
    1. **Pipeline**
    2. **Docker Pipeline**
    3. **Kubernetes CLI Plugin** (optional)
    4. **Git Plugin**

Step 2: Configure AWS Credentials in Jenkins

1. Go to **Manage Jenkins → Credentials → System → Global Credentials**.
2. Add AWS Access Key ID and Secret Access Key with an appropriate ID (e.g., aws-credentials).
3. This allows Jenkins to access **ECR** and **EKS** securely.

Step 3: Write a Jenkins Pipeline (Jenkinsfile)

Create a file named Jenkinsfile in your Git repository:

```
pipeline {
    agent any
```

```
    environment {
        AWS_REGION = 'us-east-1'
        ECR_REPO = '<aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/myapp'
        IMAGE_TAG = 'latest'
    }

    stages {
        stage('Checkout Code') {
            steps {
                git branch: 'main', url: 'https://github.com/username/myapp.git'
            }
        }

        stage('Build Docker Image') {
            steps {
                sh 'docker build -t $ECR_REPO:$IMAGE_TAG .'
            }
        }

        stage('Login to ECR') {
            steps {
                withCredentials([[$class: 'AmazonWebServicesCredentialsBinding', credentialsId: 'aws-credentials']]) {
                    sh 'aws ecr get-login-password --region $AWS_REGION | docker login --username AWS --password-stdin $ECR_REPO'
                }
            }
        }

        stage('Push Docker Image') {
            steps {
```

```
            sh 'docker push $ECR_REPO:$IMAGE_TAG'
        }
    }

    stage('Deploy to EKS') {
        steps {
            withCredentials([[$class: 'AmazonWebServicesCredentialsBinding', credentialsId: 'aws-credentials']]) {
                sh '''
                aws eks --region $AWS_REGION update-kubeconfig --name my-eks-cluster
                kubectl apply -f k8s/deployment.yaml
                kubectl rollout status deployment/myapp-deployment
                '''
            }
        }
    }
}
```

Explanation of stages:

1. **Checkout Code** → Pulls application code from Git.
2. **Build Docker Image** → Packages the app as a Docker container.
3. **Login to ECR** → Authenticates Jenkins to push Docker image to Amazon ECR.
4. **Push Docker Image** → Uploads the Docker image to ECR.
5. **Deploy to EKS** → Updates Kubernetes deployment in the EKS cluster.

Step 4: Create a Jenkins Pipeline Job

1. Go to **Jenkins Dashboard → New Item → Pipeline**.
2. Give it a name, e.g., EKS-CI-CD-Pipeline.
3. Under **Pipeline Definition**, choose **Pipeline script from SCM**.
4. Select Git and provide your repository URL.
5. Save and **Build Now**.

Step 5: Verify Deployment

After the pipeline runs successfully:

1. Check EKS cluster nodes:

kubectl get nodes

2. Check pods:

kubectl get pods

3. Ensure your application pods are running and updated with the latest Docker image.

Step 6: Optional Enhancements

- **Automated Triggers** → Trigger Jenkins pipeline on every Git commit.
- **Notifications** → Add Slack/email notifications for build status.
- **Multiple Environments** → Deploy to staging first, then production.
- **Helm Charts** → Use Helm for more advanced Kubernetes deployments.

Updates in Kubernetes (on EKS)

Updating an application typically means **changing the deployment spec**, often a **container image** or configuration.

A. Rolling Updates (Default)

- Kubernetes **Deployments** manage rolling updates by default.
- **Goal:** Update pods incrementally, ensuring zero downtime.

Steps:

1. Update the Deployment, e.g., change the container image:

kubectl set image deployment/my-app my-app-container=my-app:2.0

2. Kubernetes will:
    - Create new pods with the new image.
    - Gradually terminate old pods, respecting maxUnavailable and maxSurge.

Key Deployment Parameters:

```
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
```

maxUnavailable: 25%

maxSurge: 25%

- maxUnavailable: Maximum pods that can be down during update.
- maxSurge: Maximum additional pods created above desired count.

Check status:

kubectl rollout status deployment/my-app

B. Blue-Green or Canary Updates

EKS supports these strategies too (not built-in, but achievable with services like **Argo Rollouts** or **Istio**):

1. **Blue-Green:** Deploy a new version to a parallel environment, switch traffic via Service.
2. **Canary:** Gradually route a small portion of traffic to the new version before full rollout.

2. Rollbacks in Kubernetes (on EKS)

Sometimes updates fail or introduce bugs. Kubernetes allows **rolling back Deployments** to a previous version.

A. Using Deployment Rollback

1. Check rollout history:

kubectl rollout history deployment/my-app

2. Roll back to a previous revision:

kubectl rollout undo deployment/my-app

3. Roll back to a specific revision:

kubectl rollout undo deployment/my-app --to-revision=2

- This reverts the Deployment spec to the previous pod template.
- Kubernetes will recreate pods with the old configuration/image.

Check rollback status:

kubectl rollout status deployment/my-app

GitHub Actions for Kubernetes deployments

GitHub Actions is like a robot inside GitHub that does things automatically when you push code.

When you use it for **Kubernetes deployments**, it can:

1. **Build your app** (e.g., Docker image)

2. **Push it to a container registry** (like Docker Hub or ECR)
3. **Update your Kubernetes cluster** (like EKS) with the new version

Simple Flow

1. **Code changes** → pushed to GitHub.
2. **GitHub Actions workflow** starts automatically.
3. **Workflow steps**:
    - Build Docker image of your app.
    - Push the image to a registry.
    - Use kubectl to update your Kubernetes Deployment.

Example Workflow (simplified)

```
name: Deploy to Kubernetes

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Docker
        uses: docker/setup-buildx-action@v2

      - name: Login to ECR
```

```
  uses: aws-actions/amazon-ecr-login@v1

- name: Build and push Docker image
  run: |
    docker build -t my-app:latest .
    docker tag my-app:latest <AWS_ACCOUNT_ID>.dkr.ecr.<region>.amazonaws.com/my-app:latest
    docker push <AWS_ACCOUNT_ID>.dkr.ecr.<region>.amazonaws.com/my-app:latest

- name: Set up kubectl
  uses: azure/setup-kubectl@v3
  with:
    version: 'latest'
- name: Update Kubernetes Deployment
  run: |
    kubectl set image deployment/my-app my-app-container=<AWS_ACCOUNT_ID>.dkr.ecr.<region>.
```

Automating Deployments to EKS

Automation is typically done using **CI/CD pipelines**, with tools like **GitHub Actions, GitLab CI, Jenkins, Argo CD**, or **Flux**. The flow is usually:

A. CI/CD Pipeline Flow

1. **Code Push** → triggers the pipeline.
2. **Build Phase**:
    - Build the Docker image.
    - Run tests (unit, integration, etc.).
3. **Push Phase**:
    - Push Docker image to a container registry (ECR, Docker Hub, etc.).
4. **Deployment Phase**:
    - Update Kubernetes Deployment using kubectl, Helm, or GitOps tools.
5. **Post-Deployment Checks**:
    - Verify rollout status.

- Run smoke tests.
- Monitor logs or metrics.

Deployment Methods

1. kubectl / Helm

- Use kubectl apply or kubectl set image to update Deployments.
- Helm allows versioned releases and easy rollbacks.

Example (Helm upgrade):

helm upgrade my-app ./chart --set image.tag=2.0.1

2. GitOps Approach

- Tools like **Argo CD** or **Flux** sync a Git repository with your EKS cluster automatically.
- Any change to Git manifests triggers the deployment.

C. Rollbacks

- Always enable **rollback strategies**:
  - Kubernetes Deployment rollback: kubectl rollout undo.
  - Helm rollback: helm rollback <release> <revision>.

Best Practices for Automating EKS Deployments

A. CI/CD Best Practices

1. **Use versioned Docker images** (avoid latest tag).
2. **Run automated tests** before deployment.
3. **Keep secrets secure**:
   - AWS Secrets Manager or SSM Parameter Store.
   - Avoid storing secrets in GitHub directly.
4. **Automate rollbacks** for failed deployments.

B. Kubernetes Best Practices

1. **Use rolling updates** to avoid downtime.
2. **Set resource requests & limits** for pods.
3. **Monitor deployments**:
   - Use CloudWatch Container Insights, Prometheus, or Grafana.
4. **Separate environments**:
   - Have dev, staging, and production clusters or namespaces.

5. **Use liveness & readiness probes** to ensure healthy pods before sending traffic.
6. **Use Helm or Kustomize** for versioned, maintainable manifests.

C. GitOps Advantages

- Single source of truth (Git repo).
- Automatic drift detection and self-healing.
- Clear history of changes and easy rollback.

D. Security Best Practices

- Use **IAM roles for service accounts (IRSA)** instead of embedding AWS keys.
- Limit cluster permissions (RBAC) per service.
- Scan container images for vulnerabilities before deploying.

3. Simple Example Flow with GitHub Actions + EKS

[GitHub Push] → [CI Build & Test] → [Build & Push Docker] → [Update EKS Deployment] → [Monitor]

- Optional: Add **staging approval step** before production.