# **IaC-EKS-Pipeline Deployment Runbook**

## 

## 

## **Project Overview**

This project automates the provisioning of Amazon EKS clusters using Terraform. It supports:

* Multi-environment deployment (dev, test, prod)
* Infrastructure as Code using Terraform
* Remote state management using S3 and DynamoDB
* CI/CD using GitLab with:
  + Plan validation
  + Breaking change detection
  + Infracost for cost estimation
  + Terraform-Compliance for policy validation
  + Manual approval for production apply
  + Git version tagging

## 

## 

## 

## 

## 

## 

## **Repository Structure**

IaC-EKS-Pipeline/

├── .gitlab-ci.yml

├── infracost.yml

├── infracost-usage.yml

├── modules/

│ └── eks/

│ ├── main.tf

│ ├── variables.tf

│ ├── outputs.tf

├── environments/

│ ├── dev/

│ │ ├── main.tf

│ │ ├── terraform.tfvars

│ ├── test/

│ │ ├── main.tf

│ │ ├── terraform.tfvars

│ ├── prod/

│ │ ├── main.tf

│ │ ├── terraform.tfvars

└── tests/

└── terraform-compliance/

## 

## 

## 

## 

## 

## **Explanation of Key Files**

### **1. .gitlab-ci.yml**

Defines the CI/CD pipeline stages:

* **format:** Linting and formatting Terraform code using terraform fmt and tflint
* **init**: Initializes Terraform
* **validate**: Validates the code syntax
* **terraform\_compliance**: Enforces policies using terraform-compliance
* **infracost**: Checks cost difference in changes
* **plan**: Detects breaking changes and creates plan artifacts
* **test**: Runs optional validation tests
* **apply**: Manual apply step
* **tag**: Tags Git with pipeline version

### 

### 

### 

### **2. infracost.yml and infracost-usage.yml**

* Used by Infracost to estimate and compare infrastructure costs before applying changes.

## **infracost.yml**

This is the **Infracost configuration file**, which tells Infracost which Terraform directories or modules to analyze and which usage file to include.

Defines:

* Which Terraform files to parse
* Which usage data to apply (optional)
* How to group resources (projects/modules)

## **infracost-usage.yml**

This is an **optional file** that provides usage estimates for **usage-based AWS resources**, such as:

Improve accuracy of cost estimates, especially for resources where cost depends on usage patterns.

* EKS or EC2 usage hours
* S3 storage and requests
* NAT Gateway data processed

### **3. modules/eks/**

Reusable EKS module that includes:

* **VPC** creation (public and private subnets, routing, DNS)
* **EKS cluster** setup
* **Node groups** for worker nodes
* **IAM roles** for cluster and node access
* Outputs for kubeconfig and related resources

Files:

* main.tf: Logic for VPC, EKS, IAM, and node group creation
* variables.tf: Parameters required to customize environments
* outputs.tf: Useful outputs like cluster endpoint and kubeconfig

### **4. environments/dev|test|prod/**

Each environment includes:

* main.tf: Calls the eks module with environment-specific settings

### 

### **5. terratest/terraform-compliance/**

Houses policy tests that verify compliance rules like:

* Ensuring encryption
* Preventing public access
* Tagging resources

## **Pre-Requisites**

* AWS CLI with credentials configured
* Terraform v1.3.5+
* GitLab Runner (if self-hosted CI)
* S3 bucket and DynamoDB table for state locking

## **Deployment Steps**

### **1. Clone the Repository**

git clone https://github.com/Jokimsteny/IaC-EKS-Pipeline.git

cd IaC-EKS-Pipeline

### **2. Configure Backend**

Ensure the backend.tfvars is correctly populated with:

bucket = "your-s3-bucket"

key = "eks/dev/terraform.tfstate"

region = "us-east-1"

dynamodb\_table = "terraform-locks"

### 

### **3. Customize Variables**

Edit the terraform.tfvars files under each environment folder:

cluster\_name = "dev-cluster"

region = "us-east-1"

vpc\_cidr = "10.0.0.0/16"

...

### **4. Run Terraform Manually (Optional)**

For local testing:

cd environments/dev

terraform init -backend-config=../../backend.tfvars

terraform plan -var-file=terraform.tfvars

terraform apply -var-file=terraform.tfvars

### **5. GitLab CI/CD Workflow**

Push your changes to GitLab. The pipeline will:

* Run validation and planning
* Detect any breaking changes
* Estimate cost with Infracost
* Validate compliance rules
* Await manual approval for apply stage
* Tag the release

## **Best Practices Followed**

* Remote state and locking
* CI/CD with manual gating
* Cost awareness via Infracost
* Policy compliance testing
* Modular Terraform code
* Reusable and consistent environments

## **Rollback Strategy**

In case of a failed deployment:

* Use Git revert to rollback code
* Manually reapply last successful plan using:

terraform apply tfplan

* Use Terraform state CLI to manipulate/restore state if needed

## **Maintenance Tips**

* Rotate IAM credentials used by the pipeline
* Monitor S3 and DynamoDB usage for state
* Regularly update Terraform and module versions
* Enforce linting and formatting using tflint or terraform fmt