2025

GITLAB CI/CD PIPELINES

HANDLING CI/CD PIPELENES

VISHAL MACHAN

LINKDEIN: VISHAL MACHAN | MAIL ID: vishalmachan15@gmail.com

How do you handle deployment rollbacks in GitLab CI/CD pipelines, and what steps would you take to troubleshoot a failed pipeline?

1. Version Control for Infrastructure & Application Code

1.1 Infrastructure Code (Terraform, CloudFormation, CDK)

A. Maintain Git branches (e.g., main, staging, rollback-<version>).
B. Use Terraform state locking (e.g., AWS S3 with DynamoDB locking).
C. Store state files in a remote backend (e.g., S3, Azure Blob).
D. Version Control for CloudFormation & AWS CDK
E. Best Practices

1.2 Application Code (Docker, ECS, Kubernetes):

1.2.1 Application Code Rollback (Docker, ECS, Kubernetes)

A. Using GitLab Releases/Tags to Track Deployments
B. Docker Rollback
C. Rollback in Kubernetes (Helm, kubectl)
D. Rollback in ECS with Terraform

1.2.2 Troubleshooting Rollback Failures

A. Rollback Failures in Docker/ECR:
B. Rollback Failures in Kubernetes:
C. Terraform Rollback Failures:

1.2.3 Best Practices for Rollback in GitLab CI/CD Pipelines

a. Version Control: Tagging and Release Management
b. Automate Rollbacks on Failure
c. Monitoring & Alerting for Deployment Failures
d. Post-Deployment Validation
e. Pipeline Checks & Quality Gates

1.2.4 different scenarios:

1. Docker + Kubernetes (EKS) Deployment with Rollback
2. AWS ECS Fargate Deployment with Rollback
3. Terraform-based AWS Infrastructure Deployment

1.2.5 GitLab CI/CD pipeline for deploying Terraform and AWS CDK infrastructure securely.

1. Terraform GitLab CI/CD Pipeline
2. AWS CDK GitLab CI/CD Pipeline

1.2.6 Conclusion

1. With this GitLab CI/CD pipeline, you can automate infrastructure deployment while ensuring validation, approvals, and rollback mechanisms.

1. AWS CodePipeline for Terraform
2. AWS Code Pipeline for AWS CDK
3. GitHub Actions for Terraform
4. GitHub Actions for AWS CDK

2. Configuring Terraform to Use S3 and DynamoDB
3. Initializing Terraform with the Backend
4. Ensuring State Locking Works with DynamoDB
5. Running Terraform with Remote Backend
6. Advanced Considerations
7. Final Notes

- **S3 + DynamoDB** backend is ideal for **team environments** where multiple users or CI/CD pipelines are working on the same infrastructure.
- For **single user** or **small projects**, local state can work, but using S3 for shared state and DynamoDB for locking is **highly recommended** for collaboration and stability.

BY VISHAL MACHAN

Step 1. Automating the Terraform Backend Setup (S3 + DynamoDB)

Step 1: Create S3 Bucket and DynamoDB Table Using CloudFormation

Step 2: Automate Terraform Deployment in CI/CD

Option 1: Using AWS CodePipeline

1. Create CodePipeline with Terraform

Steps in CodePipeline:

1. **Source Stage**: Pull Terraform code from **AWS CodeCommit** or **GitHub**.
2. **Build Stage**: Run Terraform commands through **AWS CodeBuild**.
3. **Deploy Stage**: Apply the Terraform plan via the build step.

Option 2: Using GitHub Actions for CI/CD

Step **3: Automate IAM Role and Policy for Terraform in CI/CD**

Step 4: Verify the Automation Flow

Conclusion:

By using **CloudFormation**, **AWS CodePipeline**, or **GitHub Actions**, you can fully automate the **S3 + DynamoDB backend setup** for Terraform.

This approach ensures:

- **State is stored remotely** (in **S3**).
- **State locking** is enabled via **DynamoDB**.
- Terraform actions (like apply) are automated in a CI/CD pipeline, reducing manual intervention.

Handling Deployment Rollbacks in GitLab CI/CD Pipelines

Rollback strategies in GitLab CI/CD depend on the deployment method (e.g., Kubernetes, Docker, Terraform, or traditional virtual machines). Here's how to implement rollbacks effectively:

1. Version Control for Infrastructure & Application Code

1.1 Infrastructure Code (Terraform, CloudFormation, CDK)

- A. Maintain **Git branches** (e.g., main, staging, rollback-<version>).
- B. Use **Terraform state locking** (e.g., AWS S3 with DynamoDB locking).
- C. Store state files in a remote backend (e.g., S3, Azure Blob).

Version Control for Infrastructure & Application Code

Managing infrastructure and application code effectively requires version control, state management, and rollback strategies. Below is a structured approach with **detailed scenarios and examples**.

For infrastructure as code (IaC), tools like Terraform, CloudFormation, and AWS CDK are used. Proper version control ensures **collaboration, rollback, and auditing**.

A. Git Branching Strategy

- **Main Branch**: Stable, production-ready infrastructure.
- **Staging Branch**: Used for testing infrastructure changes before merging to main.
- **Feature Branches** (feature-xyz): Used for adding new infrastructure features or modifications.
- **Rollback Branches** (rollback-<version>): Created to quickly restore previous infrastructure versions in case of failure.

Example: Terraform Version Control & Branching

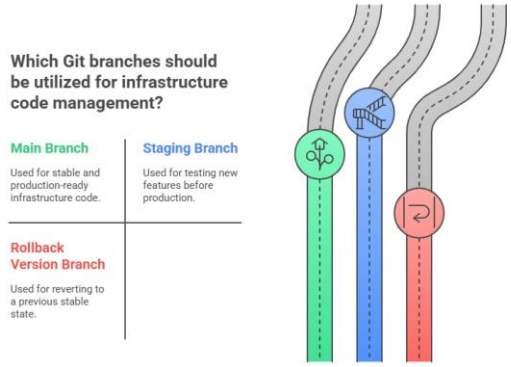
Feature Implementation

- Developer creates a branch:

git checkout -b feature-new-vpc

- Modifies Terraform code to add a new VPC

```
resource "aws_vpc" "main" {
 cidr_block = "10.0.0.0/16"
 tags = {
  Name = "MainVPC"
 }
}
```



- Pushes changes to the remote repository:

  git add .

  git commit -m "Added new VPC"

  git push origin feature-new-vpc

Testing in Staging

- Create a staging environment branch:

  git checkout -b staging

- Merge feature branch into staging:

  git merge feature-new-vpc

- Apply Terraform in staging:

  terraform init

  terraform apply

- If successful, merge into main for production.

Rollback Strategy

- If the new VPC causes issues in production, create a rollback branch:

  git checkout -b rollback-previous-vpc main~1

- Deploy the previous version:

  terraform apply

- Delete the problematic feature branch.

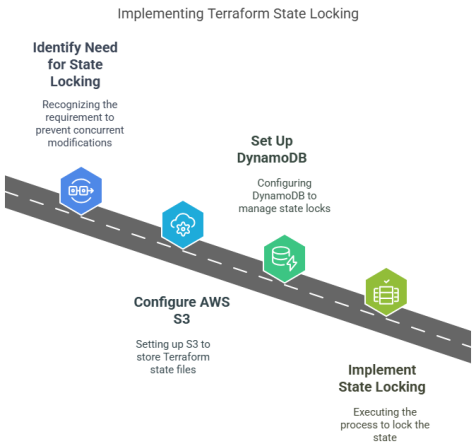
B. Terraform State Locking

Since **Terraform maintains a state file**, concurrent deployments can cause conflicts. To prevent this, **state locking** is used.

Remote Backend with AWS S3 & DynamoDB Locking :

1. Create an S3 bucket for state storage:

```
resource "aws_s3_bucket" "terraform_state" {
 bucket = "my-terraform-state-bucket"
 versioning {
  enabled = true
 }
 lifecycle {
  prevent_destroy = true
 }
}
```



2. Create a DynamoDB table for state locking:

```
resource "aws_dynamodb_table" "terraform_locks" {
 name        = "terraform-locks"
 billing_mode = "PAY_PER_REQUEST"
 hash_key    = "LockID"
 attribute {
  name = "LockID"
  type = "S"
 }
}
```

3. Configure Terraform backend in backend.tf

```
terraform {
 backend "s3" {
  bucket       = "my-terraform-state-bucket"
  key          = "infra/terraform.tfstate"
  region       = "us-east-1"
  dynamodb_table = "terraform-locks"
  encrypt      = true
 }
}
```

4. Terraform State Locking in Action

Developer A runs:

Terraform locks the state in DynamoDB, preventing other modifications.

If **Developer B** tries to apply changes simultaneously, they get:

Error: Error acquiring the state lock: Resource is locked

State unlocks automatically after **Developer A**'s operation completes.

C. Remote State Storage

Why store Terraform state remotely?

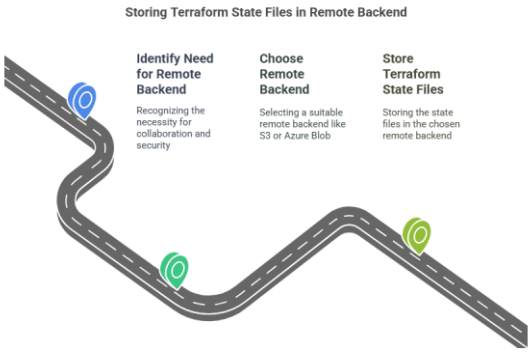
Ensure team collaboration.
Prevents loss of state files.
Enables versioning and rollback.

Example: Terraform Remote State in Azure Blob Storage

```
terraform {
 backend "azurerm" {
  resource_group_name  = "terraform-state-rg"
  storage_account_name = "terraformstatestorage"
  container_name       = "tfstate"
  key                  = "prod.terraform.tfstate"
 }
}
```

Using Remote State in Another Terraform Project:

```
data "terraform_remote_state" "network" {
  backend = "s3"
  config = {
    bucket = "my-terraform-state-bucket"
    key    = "infra/network.tfstate"
    region = "us-east-1"
  }
}

output "vpc_id" {
  value = data.terraform_remote_state.network.outputs.vpc_id
}
```



D. Version Control for CloudFormation & AWS CDK

CloudFormation and AWS CDK also require versioning best practices.

AWS CDK Git Branching & Rollback:

1. Deploying via CDK

git checkout -b feature-new-lambda

Modify CDK app (Python example):

```
from aws_cdk import aws_lambda as _lambda
from aws_cdk import core

class MyStack(core.Stack):
    def __init__(self, scope: core.Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        _lambda.Function(
            self, "MyLambdaFunction",
            runtime=_lambda.Runtime.PYTHON_3_8,
            handler="index.handler",
            code=_lambda.Code.from_asset("lambda")
        )
```

Deploy:
cdk deploy

2. Rollback a CloudFormation Stack

Check deployment history:

aws cloudformation describe-stacks --stack-name my-stack

Rollback to the last working version:

aws cloudformation rollback-stack --stack-name my-stack

E. Best Practices

Use Git Tags for Versioning:

Tagging stable releases:

git tag -a v1.0 -m "Stable infrastructure version 1.0"
git push origin v1.0

Rolling back:

git checkout v1.0
git push origin v1.0:main

Use CI/CD Pipelines

Terraform: Use GitLab CI/CD to validate changes before applying.
AWS CDK: Automate CDK deployment with GitHub Actions.

Enable State Encryption:

Encrypt Terraform state in S3 with:

encrypt = true

Secure state in Azure with:

```hcl
enable_blob_encryption = true
```

Automate Drift Detection:

Detect drift in Terraform:

terraform plan -detailed-exitcode

Detect drift in CloudFormation:
aws cloudformation detect-stack-drift --stack-name my-stack

1.2 Application Code (Docker, ECS, Kubernetes):

- Use GitLab **releases/tags** (v1.0, v1.1) to track deployments.
- Keep **previously deployed Docker images** in AWS ECR/GitLab Registry.

Handling deployment rollbacks in GitLab CI/CD pipelines is essential for ensuring that, in case of failure, you can revert to a previous stable state of application. Below are detailed strategies for implementing and troubleshooting rollbacks for Docker, ECS, Kubernetes, and traditional virtual machine deployments, with specific commands and steps.

1.2.1 Application Code Rollback (Docker, ECS, Kubernetes)

A. Using GitLab Releases/Tags to Track Deployments

Create a GitLab Release/Tag:

- Tags in GitLab provide a clear reference point to specific versions of your application.
- A tag can be created in GitLab CI/CD after a successful build, before deployment.

Command to Create a Tag:

```
git tag v1.0
git push origin v1.0
```

Using Tags for Rollback:

- Tags like v1.0, v1.1, etc., can help track versions of the application.
- In your pipeline, you can deploy specific tags to ensure that you are deploying stable versions.

B. Docker Rollback

If you're using Docker to deploy applications to AWS ECS or Kubernetes, you can store previous Docker images in AWS ECR or GitLab's Docker Registry and easily roll back by pulling a previous image.

1. Store Docker Images in ECR/GitLab Registry:

- Each build pushed to the registry should have a version/tag (e.g., v1.0, v1.1).

2. Rollback Steps:

- Step 1: Check Existing Docker Images: In your pipeline, before deployment, confirm the images that exist.

  aws ecr describe-images --repository-name my-app

- Step 3: Re-deploy with the Older Docker Image (ECS Example):
  If you're using ECS, you can update your ECS task definition to point to the older image.

  aws ecs update-service --cluster my-cluster --service my-service --force-new-deployment --image <aws_account_id>.dkr.ecr.<region>.amazonaws.com/my-app:v1.0

3. Example GitLab CI/CD Configuration for Docker Rollback:

```
stages:
 - build
 - deploy

variables:
 IMAGE_TAG: $CI_COMMIT_REF_NAME

build:
 stage: build
 script:
  - docker build -t <aws_account_id>.dkr.ecr.<region>.amazonaws.com/my-app:$IMAGE_TAG .
  - docker push <aws_account_id>.dkr.ecr.<region>.amazonaws.com/my-app:$IMAGE_TAG

deploy:
 stage: deploy
 script:
  - aws ecs update-service --cluster my-cluster --service my-service --force-new-deployment --image
<aws_account_id>.dkr.ecr.<region>.amazonaws.com/my-app:$IMAGE_TAG
```

C. Rollback in Kubernetes (Helm, kubectl)

1. Rollback Deployment using kubectl:

- Step 1: Check Available Deployments and History:
  kubectl get deployments
  kubectl rollout history deployment/my-app

- Step 2: Rollback to a Specific Revision (e.g., v1.0):
  kubectl rollout undo deployment/my-app --to-revision=1

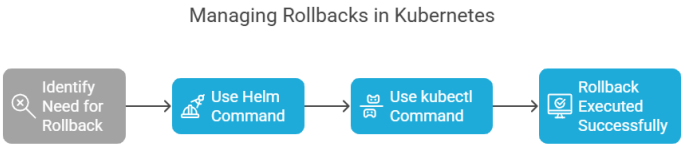
- Step 3: Check the Rollback Status:
  kubectl rollout status deployment/my-app

2. Rollback Using Helm:

If you use Helm to deploy your Kubernetes applications, you can rollback to a previous release version.

- Step 1: Check Release History:
  helm history my-app

- Step 2: Rollback to Previous Release:
  helm roll back my-app 1



D. Rollback in ECS with Terraform

If you are using Terraform to manage your ECS deployments, rollbacks can be done by simply applying the previous configuration or modifying it to a previous state.

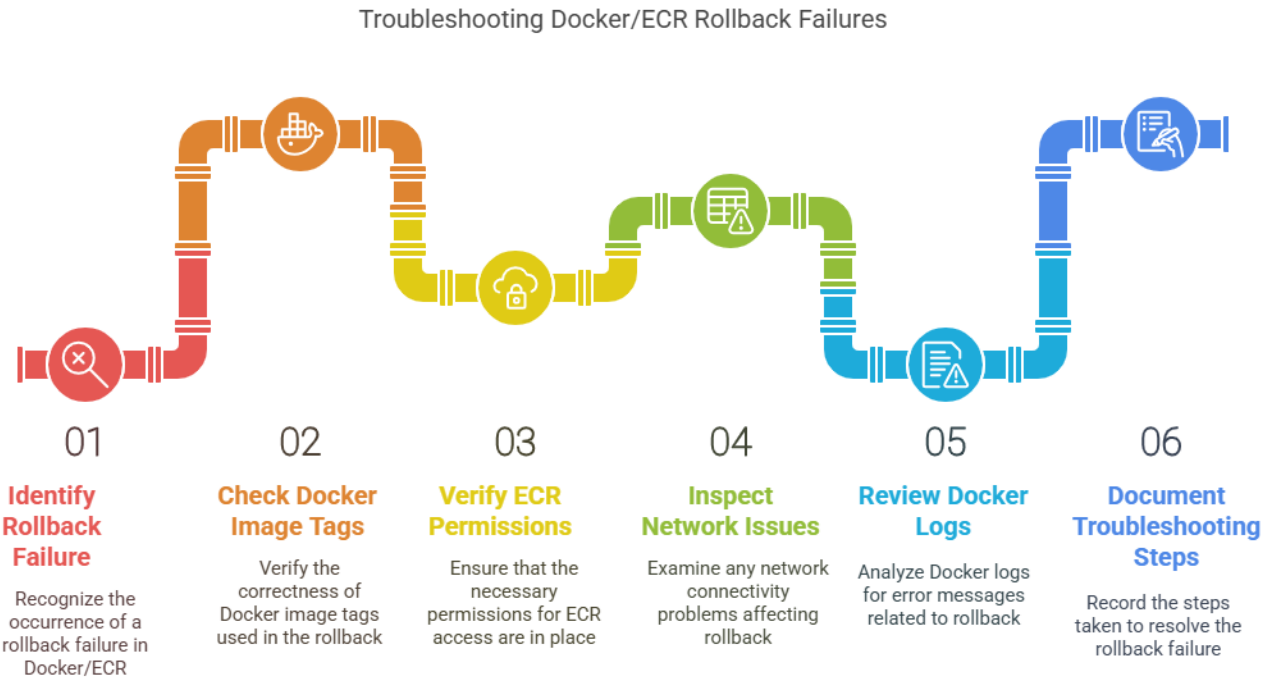
1. Rollback with Terraform:

- Step 1: View Terraform State to Check Previous Configuration:
  terraform state list
  terraform state show aws_ecs_service.my-app

- Step 2: Modify or Revert the ECS Configuration to a Stable Version: we can update the task definition or Docker image version in your Terraform code.

- Step 3: Apply Terraform Configuration:
  terraform apply

1.2.2 Troubleshooting Rollback Failures



A. Rollback Failures in Docker/ECR:

1. **Problem:** Unable to pull the Docker image.
    - Solution: Ensure the ECR repository exists and contains the correct image version. Use aws ecr describe-images to verify the image.
    - Command to Troubleshoot:
      aws ecr describe-images --repository-name my-app
2. **Problem:** ECS Service Fails to Start with New Image.
    - Solution: Check ECS service logs and the Task Definition for any errors in the Docker image configuration.
    - Commands to Troubleshoot:
      aws ecs describe-services --cluster my-cluster --services my-service
      aws ecs describe-tasks --cluster my-cluster --tasks <task_id>

B. Rollback Failures in Kubernetes:

1. **Problem:** Kubernetes Rollback Not Working.

- Solution: Check the deployment history and ensure that you are targeting the correct revision.
- Commands to Troubleshoot:
  kubectl rollout history deployment/my-app
  kubectl describe deployment my-app

2. **Problem:** Rollback Causes Pod Crash.

   - Solution: Check pod logs and events to understand why it crashes.
   - Commands to Troubleshoot:
     kubectl logs <pod-name>
     kubectl describe pod <pod-name>

C. Terraform Rollback Failures:

1. **Problem:** Terraform Fails to Apply Changes.
   - Solution: Ensure your local state is in sync with the remote state. Run terraform refresh to refresh the state.
   - Commands to Troubleshoot:
     terraform refresh
     terraform plan

2. **Problem:** Resources Are Not Reverting to Previous Configuration.
   - Solution: Manually revert the configurations or roll back to a previous state file.
   - Commands to Troubleshoot:
     terraform state pull

1.2.3 Best Practices for Rollback in GitLab CI/CD Pipelines

When implementing rollbacks in GitLab CI/CD pipelines, the following best practices can help ensure smooth, automated, and reliable deployments. Here's a detailed explanation of the five best essential practices:

a. Version Control: Tagging and Release Management
Why?
Version control is crucial for tracking and deploying stable application versions. Tags in GitLab allow you to revert to previous releases quickly.

How to Implement?

- Use Git tags to mark stable releases (v1.0, v1.1, etc.).
- Always deploy tagged versions instead of untagged commits.
- Store metadata (e.g., release notes, commit hash, changelog) in GitLab Releases.

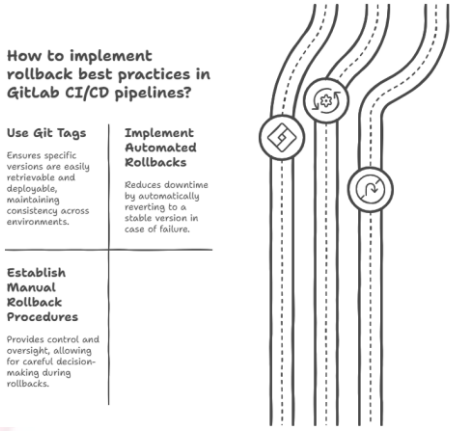
Git Commands for Version Control:

```
# Create a new tag for a stable release
git tag -a v1.1 -m "Release version 1.1 with bug fixes"
git push origin v1.1

# List available tags
git tag

# Rollback to a previous tag
git checkout v1.0
```

GitLab CI/CD Example:

Modify .gitlab-ci.yml to deploy only tagged releases:

```
stages:
 - build
 - deploy

variables:
 IMAGE_TAG: $CI_COMMIT_REF_NAME

deploy:
 stage: deploy
 only:
  - tags # Ensures only tagged releases are deployed
 script:
  - docker pull my-registry/my-app:$IMAGE_TAG
  - docker run -d -p 80:80 my-registry/my-app:$IMAGE_TAG
```



b. Automate Rollbacks on Failure

Why?

Manually rolling back deployments can lead to downtime. Automated rollbacks ensure rapid recovery when a failure is detected.

How to Implement?

- Use GitLab's after_script to detect failures.
- Implement health checks to trigger rollbacks.
- Monitor the deployment status and rollback automatically if necessary.

GitLab CI/CD Example**:**

This pipeline rolls back to the previous version if the deployment fails.

```
stages:
 - build
 - deploy

deploy:
 stage: deploy
 script:
  - kubectl apply -f deployment.yaml
  - kubectl rollout status deployment/my-app || kubectl rollout undo deployment/my-app
```

Automated Rollback Using Helm:

helm upgrade my-app ./helm-chart –atomic

The --atomic flag ensures rollback if deployment fails.

c. Monitoring & Alerting for Deployment Failures

Why?

Monitoring deployment success or failure is essential for rollback automation and minimizing downtime.

How to Implement?

- Integrate monitoring tools like **Prometheus, Grafana, AWS CloudWatch, Datadog**.
- Configure GitLab CI/CD to check application health using curl or kubectl.
- Send alerts to Slack, Microsoft Teams, or PagerDuty on failures.

GitLab CI/CD Example with Health Check:

```
deploy:
 stage: deploy
 script:
```

```
  - kubectl apply -f deployment.yaml
  - sleep 30  # Wait for pods to start
  - if ! curl -f http://my-app-service/health; then kubectl rollout undo deployment my-app; fi
```

Prometheus Alerting Rule:

```
groups:
 - name: deployment_failure_alert
   rules:
    - alert: DeploymentFailed
      expr: kube_deployment_status_replicas_unavailable > 0
      for: 5m
      labels:
        severity: critical
      annotations:
        summary: "Deployment Failure Detected"
        description: "Deployment {{ $labels.deployment }} has unavailable replicas."
```

This rule triggers an alert if a deployment has unavailable replicas for more than 5 minutes.

d. Post-Deployment Validation

Why?
Deploying code without verifying it in a live environment can cause undetected issues. Automated validation ensures new versions work correctly before finalizing deployment.

How to Implement?

- Run smoke tests after deployment to validate basic application functionality.
- Use end-to-end (E2E) tests with tools like **Selenium, Cypress, Postman**.
- Ensure rollback if validation fails.

GitLab CI/CD Example with Smoke Tests:

```
validate:
 stage: validate
 script:
   - echo "Running smoke tests..."
   - curl -f http://my-app-service/health || exit 1
```

Post-Deployment Testing with Cypress:

```
e2e_tests:
 stage: test
 script:
  - npx cypress run
```

If Cypress tests fail, the pipeline fails, triggering an automated rollback.

e. Pipeline Checks & Quality Gates

Why?

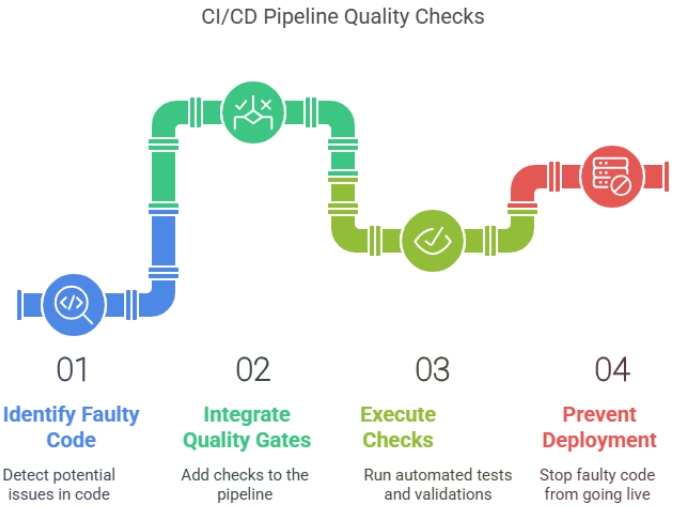
Skipping critical quality checks in CI/CD leads to unstable deployments. Quality gates ensure only stable versions reach production.

How to Implement?

- Use unit tests, integration tests, and security scans before deployment.
- Implement an approval step in GitLab CI/CD for production releases.
- Require manual approval before rollback (if needed).

GitLab CI/CD Example with Quality Gates:

```
stages:
 - test
 - security_scan
 - staging
 - production

unit_tests:
 stage: test
 script:
  - pytest tests/

security_scan:
 stage: security_scan
 script:
  - trivy image my-app:latest

staging:
 stage: staging
 script:
  - kubectl apply -f staging-deployment.yaml
 environment:
  name: staging
 when: manual  # Requires manual trigger
```



```
production:
 stage: production
 script:
   - kubectl apply -f production-deployment.yaml
 only:
   - tags  # Ensures only tagged releases are deployed
 when: manual  # Requires manual approval
```

- when: manual ensures a human review before deploying to staging or production.
- The pipeline enforces security scans and unit tests before deployment.

Conclusion

By following these best practices, you ensure that your **GitLab CI/CD pipeline is resilient, automated, and rollback-friendly**.

- ✅ **Version Control**: Always use Git tags to deploy stable releases.
- ✅ **Automate Rollbacks**: Automatically revert deployments upon failure.
- ✅ **Monitoring & Alerting**: Use Prometheus, Grafana, or CloudWatch to detect issues.
- ✅ **Post-Deployment Validation**: Run health checks, smoke tests, and end-to-end tests.
- ✅ **Quality Gates**: Require testing, security scans, and manual approvals before production.

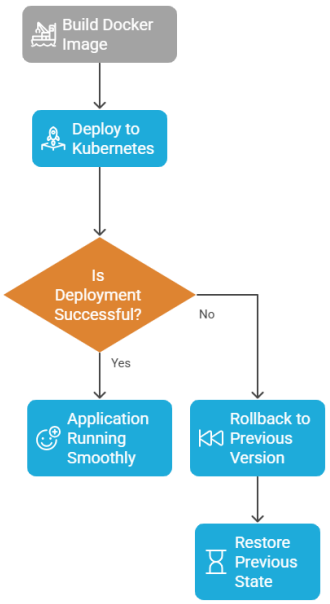
1.2.4 Since your deployment method involves **Docker, ECS, Kubernetes, and Terraform**, here's a tailored **.gitlab-ci.yml** template for **different scenarios:**

1. **Docker + Kubernetes (EKS) Deployment with Rollback**

```
stages:
 - build
 - test
 - security_scan
 - deploy_staging
 - deploy_production

variables:
 IMAGE_TAG: $CI_COMMIT_REF_NAME
 REGISTRY: my-registry/my-app
 KUBE_NAMESPACE: my-namespace
 DEPLOYMENT_NAME: my-app-deployment
```

```
  HEALTH_CHECK_URL: http://my-app-service/health

build:
  stage: build
  script:
    - docker build -t $REGISTRY:$IMAGE_TAG .
    - docker push $REGISTRY:$IMAGE_TAG

test:
  stage: test
  script:
    - pytest tests/

security_scan:
  stage: security_scan
  script:
    - trivy image $REGISTRY:$IMAGE_TAG

deploy_staging:
  stage: deploy_staging
  script:
    - kubectl config set-context --current --namespace=$KUBE_NAMESPACE
    - kubectl set image deployment/$DEPLOYMENT_NAME app=$REGISTRY:$IMAGE_TAG
    - sleep 30
    - if ! curl -f $HEALTH_CHECK_URL; then kubectl rollout undo deployment $DEPLOYMENT_NAME; fi
  environment:
    name: staging
  when: manual

deploy_production:
  stage: deploy_production
  script:
    - kubectl config set-context --current --namespace=$KUBE_NAMESPACE
    - kubectl set image deployment/$DEPLOYMENT_NAME app=$REGISTRY:$IMAGE_TAG
    - sleep 30
    - if ! curl -f $HEALTH_CHECK_URL; then kubectl rollout undo deployment $DEPLOYMENT_NAME; fi
  only:
    - tags
  when: manual
  environment:
    name: production
```

Docker and Kubernetes Deployment with Rollback



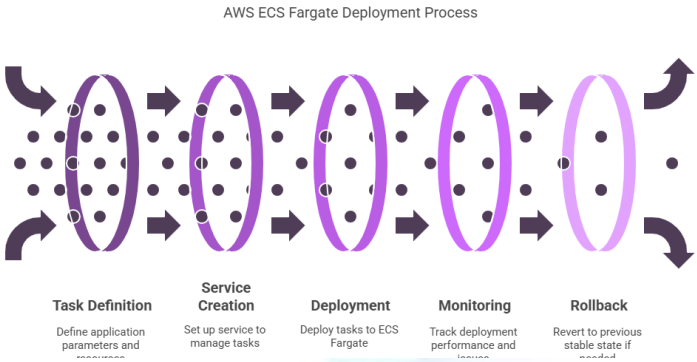
✅ **Features**:

- Pushes a Docker image to the registry.
- Runs **unit tests and security scans** before deployment.
- Deploys to **staging manually** (requires approval).

- Deploys to **production only when a tag is pushed** (e.g., v1.1).
- **Automatic rollback if health check fails** (curl -f validation).

2. AWS ECS Fargate Deployment with Rollback

```
stages:
 - build
 - test
 - deploy_staging
 - deploy_production

variables:
 AWS_REGION: us-east-1
 ECR_REPO: my-ecr-repo
 CLUSTER: my-ecs-cluster
 SERVICE: my-ecs-service
 IMAGE_TAG: $CI_COMMIT_REF_NAME
```




```
before_script:
 - export AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID
 - export AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY

build:
 stage: build
 script:
  - docker build -t $ECR_REPO:$IMAGE_TAG .
  - docker tag $ECR_REPO:$IMAGE_TAG
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/$ECR_REPO:$IMAGE_TAG
  - docker push $AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/$ECR_REPO:$IMAGE_TAG

test:
 stage: test
 script:
  - pytest tests/

deploy_staging:
 stage: deploy_staging
 script:
  - aws ecs update-service --cluster $CLUSTER --service $SERVICE --force-new-deployment
 environment:
  name: staging
 when: manual

deploy_production:
 stage: deploy_production
```

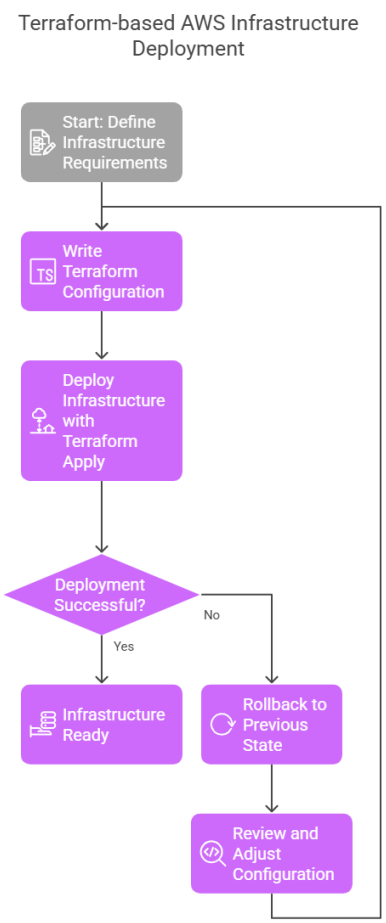
```
  script:
    - aws ecs update-service --cluster $CLUSTER --service $SERVICE --force-new-deployment
  only:
    - tags
  when: manual
  environment:
    name: production
```

✅ **Features**:

- Builds and pushes Docker images to **AWS ECR**.
- Uses aws ecs update-service --force-new-deployment for **zero-downtime deployment**.
- Staging requires **manual approval** before deployment.
- Production deploys **only on tagged releases**.

3. Terraform-based AWS Infrastructure Deployment

```
stages:
  - terraform_init
  - terraform_plan
  - terraform_apply

variables:
  TF_WORKSPACE: production
  AWS_REGION: us-east-1

terraform_init:
  stage: terraform_init
  script:
    - terraform init

terraform_plan:
  stage: terraform_plan
  script:
    - terraform plan -out=tfplan
  when: manual

terraform_apply:
  stage: terraform_apply
  script:
    - terraform apply tfplan
  only:
    - tags
```



```
when: manual
```

✅ **Features**:

- **Terraform init, plan, and apply are separated**.
- Requires **manual approval before Terraform apply**.
- Production deployments **only happen when a tag is pushed**.

Which One Fits Your Stack Best?

- **If using Kubernetes/EKS:** Go with the first template.
- **If deploying via AWS ECS:** The second template is ideal.
- **If provisioning infrastructure with Terraform:** Use the third one

1.2.5 GitLab CI/CD pipeline for deploying Terraform and AWS CDK infrastructure securely.

1. **Terraform GitLab CI/CD Pipeline**

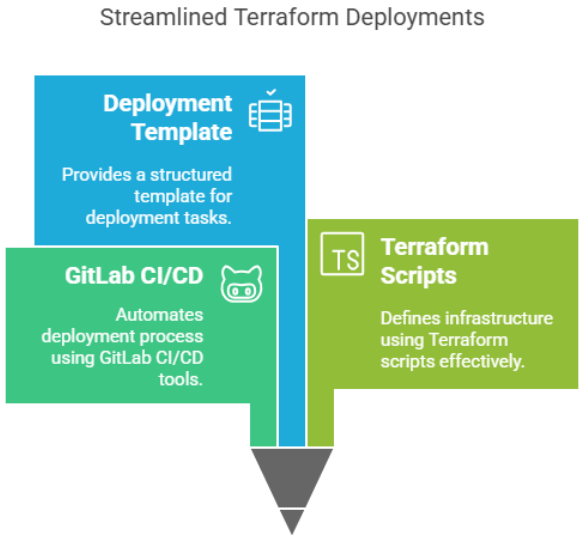
This pipeline:

✅ **Validates** Terraform configuration.
✅ **Plans** changes before applying.
✅ **Locks** Terraform state using AWS S3 + DynamoDB.
✅ **Automatically applies** changes in production.
✅ **Supports manual approval before deployment.**

GitLab CI/CD Configuration (.gitlab-ci.yml)

```
stages:
 - validate
 - plan
 - apply

variables:
 TF_VERSION: "1.5.0"
 AWS_REGION: "us-east-1"
 S3_BUCKET: "my-terraform-state-bucket"
 DYNAMODB_TABLE: "terraform-locks"

before_script:
```

```
 - apt-get update && apt-get install -y unzip
 - curl -fsSL https://apt.releases.hashicorp.com/gpg | apt-key add -
 - echo "deb [arch=amd64] https://apt.releases.hashicorp.com $(lsb_release -cs) main" | tee
/etc/apt/sources.list.d/hashicorp.list
 - apt-get update && apt-get install -y terraform=$TF_VERSION

# Validate Terraform Configuration
terraform-validate:
 stage: validate
 script:
  - terraform init
  - terraform validate
 only:
  - merge_requests
  - main

# Terraform Plan (Preview Changes)
terraform-plan:
 stage: plan
 script:
  - terraform init
  - terraform plan -out=tfplan
 artifacts:
  paths:
   - tfplan
 only:
  - merge_requests

# Terraform Apply (Manual Approval Required)
terraform-apply:
 stage: apply
 script:
  - terraform init
  - terraform apply -auto-approve tfplan
 environment:
  name: production
 only:
  - main
 when: manual  # Requires manual approval in GitLab UI
```



Terraform Backend Configuration (backend.tf)

```
terraform {
 backend "s3" {
  bucket       = "my-terraform-state-bucket"
  key          = "infra/terraform.tfstate"
```

```
  region         = "us-east-1"
  dynamodb_table = "terraform-locks"
  encrypt        = true
 }
}
```

Pipeline Workflow

8. **Push changes to a feature branch** (feature-vpc).
9. **Create a merge request** → GitLab runs terraform validate & terraform plan.
10. **Review the Terraform Plan** in GitLab.
11. **Merge into main branch** → terraform apply is triggered (manual approval required).
12. **Terraform updates infrastructure** in AWS.

2. AWS CDK GitLab CI/CD Pipeline

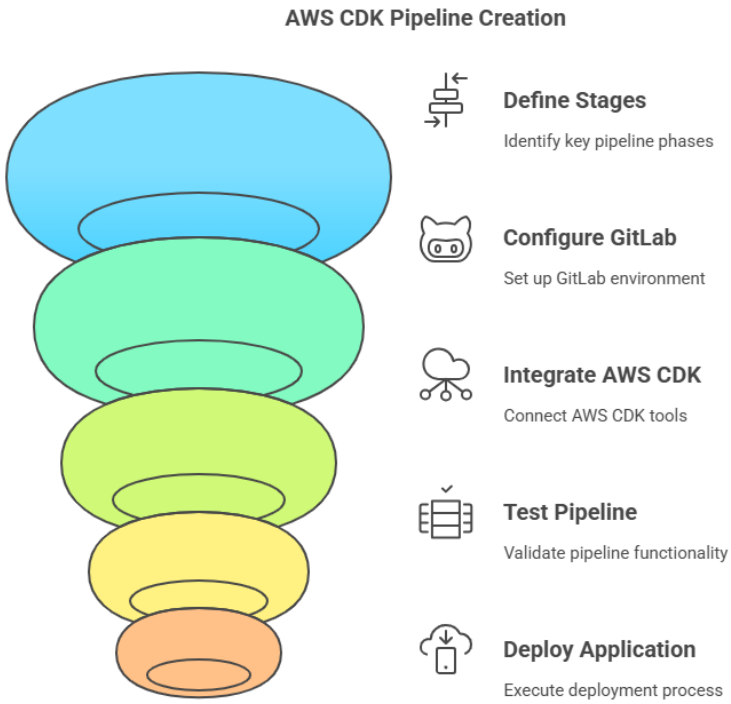
This pipeline:

- ✅ **Installs AWS CDK**
- ✅ **Synthesizes CloudFormation templates**
- ✅ **Deploys CDK stacks** to AWS
- ✅ **Supports rollback in case of failures**

GitLab CI/CD Configuration (.gitlab-ci.yml)

```
stages:
 - build
 - synth
 - deploy

variables:
 AWS_REGION: "us-east-1"
 CDK_VERSION: "2.120.0"
```

```
before_script:
 - apt-get update && apt-get install -y nodejs npm
 - npm install -g aws-cdk@$CDK_VERSION

# Install dependencies
install-dependencies:
 stage: build
 script:
  - npm install
 only:
  - merge_requests
  - main

# CDK Synth (Generates CloudFormation Templates)
cdk-synth:
 stage: synth
 script:
  - cdk synth
 only:
  - merge_requests

# CDK Deploy (Manual Approval Required)
cdk-deploy:
 stage: deploy
 script:
  - cdk deploy --require-approval never
 environment:
  name: production
 only:
```



```
    - main
  when: manual  # Requires manual approval in GitLab UI
```

CDK Stack Example (cdk-stack.ts)

```
import * as cdk from "aws-cdk-lib";
import * as lambda from "aws-cdk-lib/aws-lambda";

export class MyStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    new lambda.Function(this, "MyLambdaFunction", {
      runtime: lambda.Runtime.NODEJS_18_X,
      handler: "index.handler",
      code: lambda.Code.fromAsset("lambda"),
    });
  }
}
```

Pipeline Workflow

1. **Push changes to a feature branch** (feature-lambda).
2. **Create a merge request** → GitLab runs npm install & cdk synth.
3. **Merge into main** → cdk deploy is triggered (manual approval required).
4. **CDK updates AWS infrastructure**.

Bonus: Rollback & Drift Detection

Terraform Rollback in GitLab

If terraform apply fails:

terraform state list | grep aws_vpc

terraform state rm aws_vpc.main

terraform import aws_vpc.main vpc-12345678

CDK Rollback in GitLab

If cdk deploy fails:

aws cloudformation rollback-stack --stack-name MyStack

1.2.6 Conclusion:

With this **GitLab CI/CD pipeline**, you can **automate infrastructure deployment** while ensuring **validation, approvals, and rollback mechanisms**.

1. **AWS CodePipeline for Terraform**

This pipeline:

- ✅ Uses **AWS CodeCommit** as the repo.
- ✅ Validates, plans, and applies Terraform code.
- ✅ Stores state in **S3 + DynamoDB locking**.
- ✅ Supports **manual approval before deployment**

Terraform CodePipeline Architecture

1. **AWS CodeCommit** – Stores Terraform code.
2. **AWS CodeBuild** – Runs Terraform commands.
3. **Manual Approval** – Requires confirmation before terraform apply.
4. **AWS CodeBuild** – Applies Terraform changes.
5. **State Storage** – Uses **S3 + DynamoDB** for state locking.

Terraform Pipeline (buildspec.yml)

```
version: 0.2

env:
 variables:
  TF_VERSION: "1.5.0"
  AWS_REGION: "us-east-1"
  S3_BUCKET: "my-terraform-state-bucket"
  DYNAMODB_TABLE: "terraform-locks"

phases:
 install:
  runtime-versions:
   python: latest
 pre_build:
  commands:
   - echo "Installing Terraform..."
   - curl -O https://releases.hashicorp.com/terraform/${TF_VERSION}/terraform_${TF_VERSION}_linux_amd64.zip
   - unzip terraform_${TF_VERSION}_linux_amd64.zip
   - mv terraform /usr/local/bin/
   - terraform --version
 build:
  commands:
   - echo "Initializing Terraform..."
   - terraform init
   - echo "Running Terraform Plan..."
   - terraform plan -out=tfplan
   - terraform show -json tfplan > tfplan.json
 post_build:
```

```
  commands:
    - echo "Terraform Plan Completed."
    - echo "Upload tfplan.json to S3 for approval."
```

Terraform Apply Pipeline (apply-buildspec.yml)

```
version: 0.2

phases:
  install:
    runtime-versions:
      python: latest
  build:
    commands:
      - echo "Applying Terraform Changes..."
      - terraform apply -auto-approve tfplan
  post_build:
    commands:
      - echo "Terraform Apply Completed."
```

Pipeline Workflow

1. **Developer pushes Terraform code** → CodePipeline triggers.
2. **Terraform Plan runs** in CodeBuild.
3. **Manual Approval step** in AWS Console.
4. **Terraform Apply runs** to update infrastructure.

2. AWS Code Pipeline for AWS CDK

This pipeline:

✅ Uses **AWS Code Commit** for CDK source.
✅ Synthesizes CloudFormation templates.
✅ Deploys CDK stacks to AWS.
✅ Supports **rollback in case of failure**.

CDK Pipeline Architecture

1. **AWS Code Commit** – Stores CDK code.
2. **AWS Code Build** – Runs cdk synth.
3. **Manual Approval** – Confirms before cdk deploy.
4. **AWS Code Build** – Deploys CDK stacks.

CDK Buildspec (cdk-buildspec.yml)

```
version: 0.2

phases:
  install:
    commands:
      - echo "Installing AWS CDK..."
      - npm install -g aws-cdk
  pre_build:
    commands:
      - npm install
  build:
    commands:
      - echo "Synthesizing CloudFormation Templates..."
      - cdk synth
```

```
 post_build:
  commands:
   - echo "CDK Synth Completed."
```

CDK Deploy Buildspec (cdk-deploy-buildspec.yml)

```
version: 0.2

phases:
 install:
  commands:
   - npm install -g aws-cdk
 build:
  commands:
   - echo "Deploying CDK Stack..."
   - cdk deploy --require-approval never
 post_build:
  commands:
   - echo "CDK Deployment Completed."
```

Pipeline Workflow

1. **Developer pushes CDK code** → Code Pipeline triggers.
2. **CDK Synth runs** in Code Build.
3. **Manual Approval step** in AWS Console.
4. **CDK Deploy runs** to update infrastructure.

3. GitHub Actions for Terraform

This **GitHub Actions pipeline**:

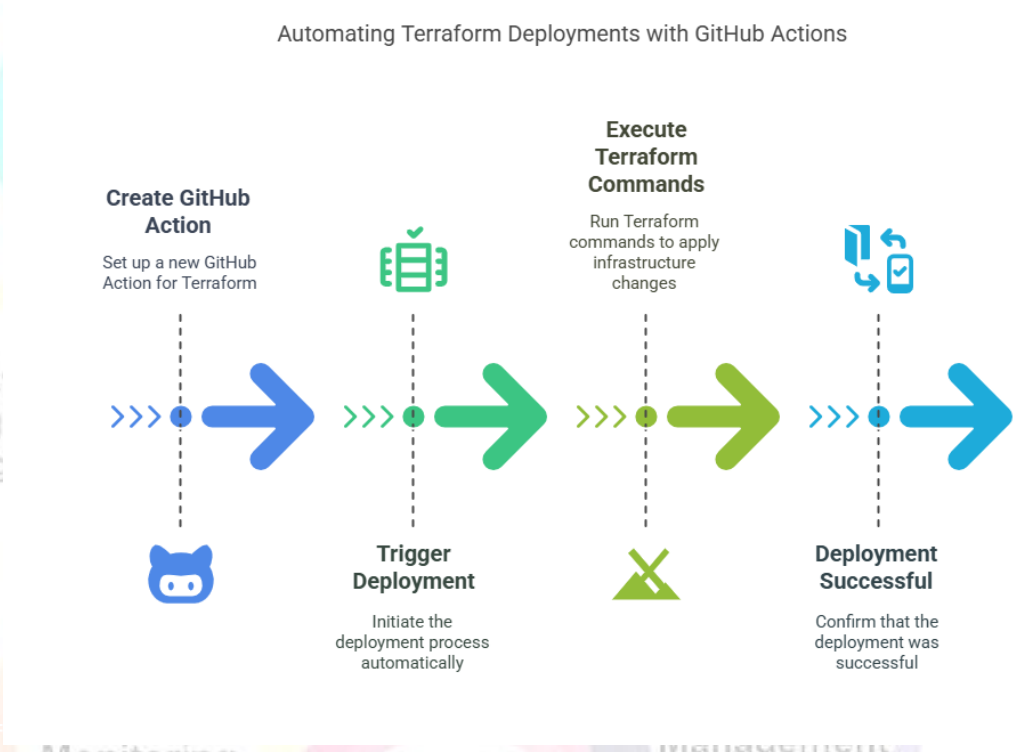
✅ Runs Terraform validate, plan, and apply.

✅ Uses **AWS OIDC authentication** (no credentials stored).

✅ Supports **manual approval before apply**.

GitHub Actions Workflow (terraform.yml)

```
name: Terraform Deployment

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

permissions:
  id-token: write
  contents: read

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

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2
        with:
          terraform_version: "1.5.0"
```



```
    - name: Terraform Init
      run: terraform init

    - name: Terraform Validate
      run: terraform validate

    - name: Terraform Plan
      run: terraform plan -out=tfplan

    - name: Require Manual Approval
      uses: hmarr/auto-approve-action@v3
      with:
        github-token: ${{ secrets.GITHUB_TOKEN }}

    - name: Terraform Apply
      run: terraform apply -auto-approve tfplan
```

4. GitHub Actions for AWS CDK

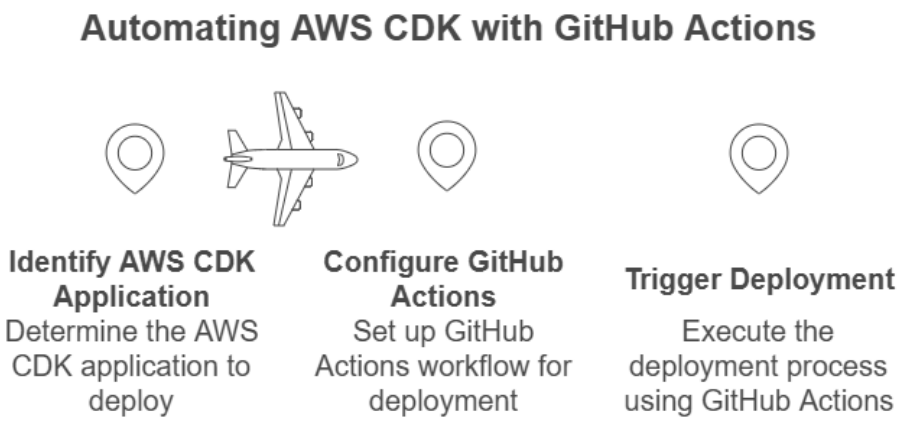
This **GitHub Actions pipeline**:

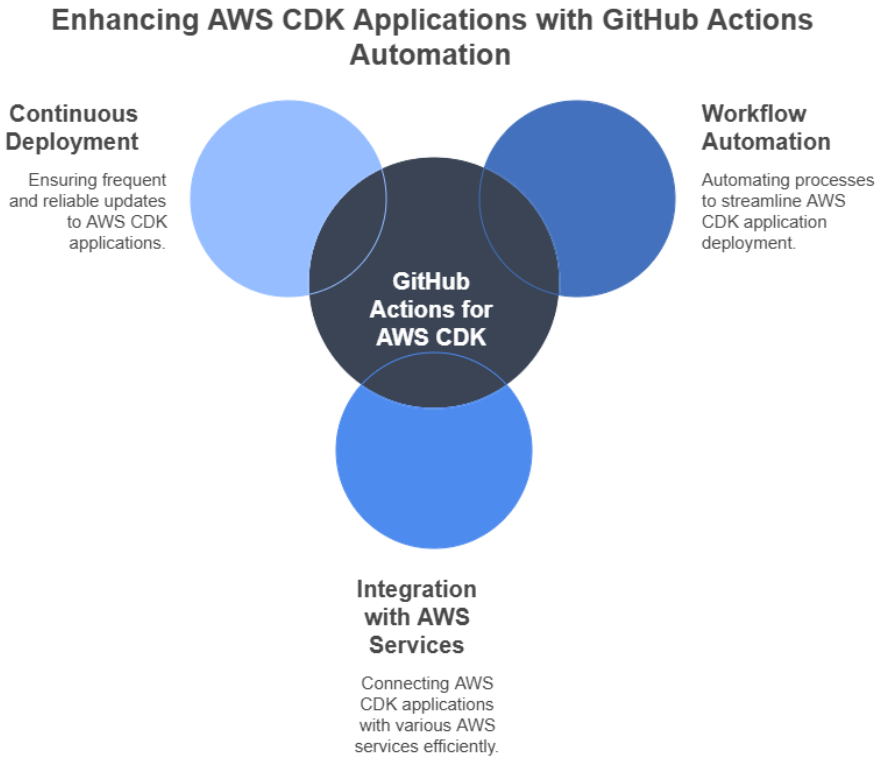
- ✅ Synthesizes CloudFormation templates.
- ✅ Deploys CDK stacks automatically.
- ✅ Uses **AWS OIDC authentication**.

GitHub Actions Workflow (cdk.yml)

```
name: AWS CDK Deployment

on:
  push:
```

```
  branches:
    - main

permissions:
  id-token: write
  contents: read

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

    - name: Install Node.js
      uses: actions/setup-node@v3
      with:
        node-version: "18"

    - name: Install AWS CDK
      run: npm install -g aws-cdk

    - name: Install Dependencies
      run: npm install

    - name: CDK Synth
      run: cdk synth

    - name: CDK Deploy
      run: cdk deploy --require-approval never
```





Which One to Use?

Feature	AWS CodePipeline	GitHub Actions
Managed by AWS	✅	❌
Fully integrated with AWS IAM	✅	⚠️ (Uses OIDC)
Manual Approval	✅	✅ (With workaround)
Cost	💰 AWS Service Cost	Free for small teams
Speed	🚀 Faster for AWS	⚡ Better for multi-cloud

1. Setting Up S3 for Terraform State Storage

S3 will store your Terraform state files, and DynamoDB will be used for **state locking** to prevent concurrent modifications.

Step 1: Create an S3 Bucket

1. **Go to AWS S3 Console** and create a new bucket for storing Terraform state. For example, my-terraform-state-bucket.
    - Set the **bucket name** to something unique.
    - Enable **versioning** to keep historical versions of your state file.
    - Enable **encryption** (recommended for security) using AWS **S3 Server-Side Encryption (SSE-S3)**.

Step 2: Create a DynamoDB Table for State Locking

1. **Go to AWS DynamoDB Console** and create a new table with:
    - **Table name**: terraform-locks
    - **Partition key**: LockID (string).
    - **Provisioned** capacity mode is sufficient for most cases.
2. This DynamoDB table will help lock the state file to prevent concurrent Terraform runs from affecting each other.

2. Configuring Terraform to Use S3 and DynamoDB

You now need to modify your Terraform configuration to use the **S3 backend** for state storage and **DynamoDB** for state locking.

Example Terraform Configuration (backend.tf)

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state-bucket"     # Name of the S3 bucket
    key            = "terraform.tfstate"             # Path within the S3 bucket (state file name)
    region         = "us-east-1"                     # AWS region where the S3 bucket and DynamoDB table exist
    encrypt        = true                            # Enable server-side encryption for the state file
    dynamodb_table = "terraform-locks"               # DynamoDB table for state locking
    acl            = "bucket-owner-full-control"     # ACL for the state bucket
  }
}
```

- **bucket**: The name of the S3 bucket you created for storing the state.
- **key**: Path to store the state file. Typically, this is terraform.tfstate or you can set it up with subfolders (e.g., project_name/terraform.tfstate).
- **region**: The AWS region where your S3 bucket and DynamoDB table reside.
- **dynamodb_table**: Name of the DynamoDB table used for state locking.
- **encrypt**: Ensures your state file is encrypted at rest in S3.

3. Initializing Terraform with the Backend

- Once your configuration is ready, run the following command in your terminal:

```
terraform init
```

Terraform will initialize the backend configuration and move your state to S3.

- **Initial Setup**: During the terraform init process, if this is your first time setting up the backend, Terraform will ask you to confirm that it can migrate your local state to the new backend.
- **State Migration**: If you have an existing state file, Terraform will prompt to migrate your local state to the S3 bucket.

4. Ensuring State Locking Works with DynamoDB

When Terraform is working with the state, it will **create a lock** in DynamoDB to ensure that only one Terraform process can modify the state at any given time.

Verify DynamoDB Locking

- When you run terraform apply or any other command that modifies the state, Terraform will attempt to acquire a lock in DynamoDB.
- If another Terraform process is already running, the lock will be in place, and your process will wait until it can acquire the lock.

5. Running Terraform with Remote Backend

- Once the backend is set up, any Terraform command (e.g., terraform plan, terraform apply, terraform destroy) will interact with the remote backend. For example:

terraform plan

terraform apply

6. Advanced Considerations

State File Versioning in S3

- **Versioning** is essential for keeping multiple versions of your state files in case you need to roll back changes.
- It can be enabled while creating the bucket or updated later through the S3 console.

State File Encryption

- **Encryption** ensures the confidentiality of your state file, especially since it might contain sensitive information (e.g., passwords, API keys).
- You can enable **SSE-S3** (default encryption) or **SSE-KMS** (more control) to encrypt the state files in S3.

IAM Permissions for S3 & DynamoDB

Make sure the **IAM role** running Terraform (whether it's via **AWS CLI**, **CodePipeline**, or **GitHub Actions**) has the necessary permissions:

1. **S3 Permissions**: To read/write the state file.
2. **DynamoDB Permissions**: To lock and unlock the state file.

Here's an example of a basic IAM policy:

{

```
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "s3:GetObject",
    "s3:PutObject",
    "s3:ListBucket"
   ],
   "Resource": [
    "arn:aws:s3:::my-terraform-state-bucket",
    "arn:aws:s3:::my-terraform-state-bucket/*"
   ]
  },
  {
   "Effect": "Allow",
   "Action": "dynamodb:PutItem",
   "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/terraform-locks"
  }
 ]
}
```

7. Final Notes

- **S3 + DynamoDB** backend is ideal for **team environments** where multiple users or CI/CD pipelines are working on the same infrastructure.
- For **single user** or **small projects**, local state can work, but using S3 for shared state and DynamoDB for locking is **highly recommended** for collaboration and stability.

1. Automating the Terraform Backend Setup (S3 + DynamoDB)

- First, we'll walk through automating the setup using **AWS CloudFormation** (to automate S3 and DynamoDB creation), and then automate **Terraform** deployment through **AWS CodePipeline** or **GitHub Actions**.

Step 1: Create S3 Bucket and DynamoDB Table Using CloudFormation

- You can automate the creation of **S3** and **DynamoDB** resources using **AWS CloudFormation**. Here's the CloudFormation template:

CloudFormation Template (infrastructure-backend.yaml)

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
 TerraformStateBucket:
  Type: "AWS::S3::Bucket"
  Properties:
   BucketName: "my-terraform-state-bucket"
   VersioningConfiguration:
    Status: "Enabled"
   Tags:
    - Key: "Name"
      Value: "Terraform State Bucket"
```

TerraformStateLockTable:

```
  Type: "AWS::DynamoDB::Table"
  Properties:
   TableName: "terraform-locks"
   AttributeDefinitions:
```

```
    - AttributeName: "LockID"
      AttributeType: "S"
  KeySchema:
    - AttributeName: "LockID"
      KeyType: "HASH"
  ProvisionedThroughput:
    ReadCapacityUnits: 5
    WriteCapacityUnits: 5
  Tags:
    - Key: "Name"
      Value: "Terraform State Lock Table"
```

This template does the following:

- **Creates an S3 bucket** (my-terraform-state-bucket) with versioning enabled for Terraform state files.
- **Creates a DynamoDB table** (terraform-locks) to handle state locking during Terraform operations.

Deploy CloudFormation Stack

You can deploy the CloudFormation stack using the AWS CLI:

aws cloudformation create-stack --stack-name terraform-backend-stack --template-body file://infrastructure-backend.yaml

Step 2: Automate Terraform Deployment in CI/CD

Now that the backend resources are created, we will automate the deployment of Terraform code through **AWS CodePipeline** or **GitHub Actions**.

Option 1: Using AWS CodePipeline

In **AWS CodePipeline**, you can automate your Terraform deployment process, including initializing the backend, planning, applying, and destroying resources.

1. Create CodePipeline with Terraform

You will need:

- **Source Stage**: Use **AWS CodeCommit** or **GitHub** for storing your Terraform code.
- **Build Stage**: Use **AWS CodeBuild** to run Terraform commands like terraform init, terraform plan, and terraform apply.

Example CodeBuild Buildspec File (buildspec.yml)

```
version: 0.2

phases:
  install:
    runtime-versions:
      terraform: 1.x
    commands:
      - echo "Installing Terraform"
  pre_build:
    commands:
      - echo "Initializing Terraform"
      - terraform init -backend-config="bucket=my-terraform-state-bucket" -backend-config="key=terraform.tfstate" -backend-config="region=us-east-1" -backend-config="dynamodb_table=terraform-locks"
  build:
    commands:
      - echo "Running Terraform Plan"
      - terraform plan -out=tfplan
  post_build:
    commands:
      - echo "Applying Terraform Plan"
      - terraform apply -auto-approve tfplan

artifacts:
```

```
  files:

    - terraform.tfstate
```

Steps in CodePipeline:

4. **Source Stage**: Pull Terraform code from **AWS CodeCommit** or **GitHub**.
5. **Build Stage**: Run Terraform commands through **AWS CodeBuild**.
6. **Deploy Stage**: Apply the Terraform plan via the build step.

Option 2: Using GitHub Actions for CI/CD

In **GitHub Actions**, you can automate the process by using workflows.

GitHub Actions Workflow Example (.github/workflows/terraform.yml)

```
name: Terraform CI/CD Pipeline

on:

  push:

    branches:

      - main

jobs:

  terraform:

    runs-on: ubuntu-latest

    steps:

      - name: Checkout code

        uses: actions/checkout@v2
```

```
    - name: Set up Terraform

      uses: hashicorp/setup-terraform@v1


    - name: Terraform Init

      run: |

        terraform init -backend-config="bucket=my-terraform-state-bucket" -backend-config="key=terraform.tfstate" -backend-config="region=us-east-1" -backend-config="dynamodb_table=terraform-locks"


    - name: Terraform Plan

      run: terraform plan -out=tfplan


    - name: Terraform Apply

      run: terraform apply -auto-approve tfplan

      if: github.event_name == 'push' && github.ref == 'refs/heads/main'
```

How GitHub Actions Works:

1. **Push to main**: Triggered when pushing code to the main branch.
2. **Terraform Setup**: Sets up Terraform using **HashiCorp's GitHub Action**.
3. **Terraform Initialization**: Initializes the backend, specifying S3 and DynamoDB.
4. **Terraform Plan**: Runs terraform plan to see the changes.
5. **Terraform Apply**: Applies the changes if on main.

3: Automate IAM Role and Policy for Terraform in CI/CD

For **CodePipeline**, **CodeBuild**, or **GitHub Actions** to interact with AWS resources securely, ensure they have the appropriate IAM roles and permissions.

IAM Policy for Terraform in CI/CD

Attach this policy to the service role in **AWS CodeBuild** or the IAM user used in **GitHub Actions**.

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "s3:GetObject",
    "s3:PutObject",
    "s3:ListBucket",
    "s3:PutBucketVersioning",
    "s3:PutBucketEncryption"
   ],
   "Resource": [
    "arn:aws:s3:::my-terraform-state-bucket",
    "arn:aws:s3:::my-terraform-state-bucket/*"
   ]
  },
  {
   "Effect": "Allow",
   "Action": "dynamodb:PutItem",
   "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/terraform-locks"
  },
  {
   "Effect": "Allow",
   "Action": "cloudformation:DescribeStacks",
   "Resource": "*"
  }
 ]
}
```

Step 4: Verify the Automation Flow

After setting up the **CI/CD pipeline**:

1. **Push your code** (Terraform configuration) to **GitHub** or **CodeCommit**.
2. The **GitHub Action** or **AWS CodePipeline** will trigger the pipeline and automatically run the Terraform steps.
3. **Check S3** for the state file and **DynamoDB** for state locking.

Conclusion:

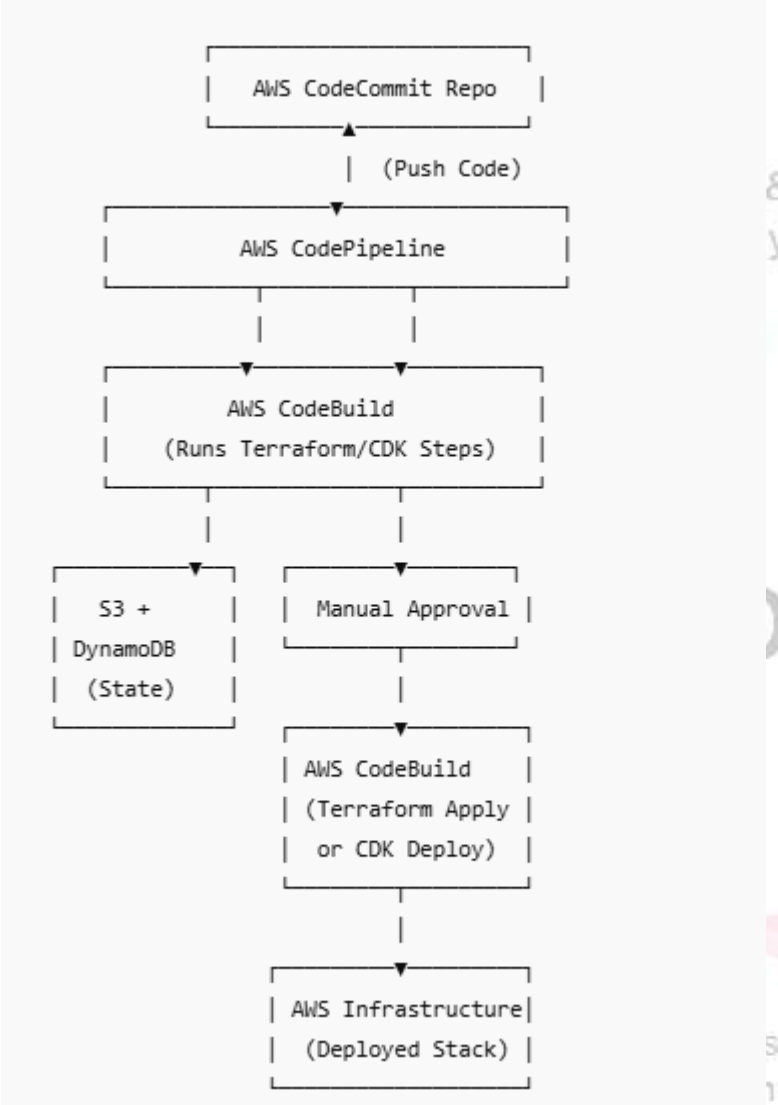
By using **CloudFormation**, **AWS CodePipeline**, or **GitHub Actions**, you can fully automate the **S3 + DynamoDB backend setup** for Terraform.

This approach ensures:

- **State is stored remotely** (in **S3**).

- **State locking** is enabled via **DynamoDB**.
- Terraform actions (like apply) are automated in a CI/CD pipeline, reducing manual intervention.

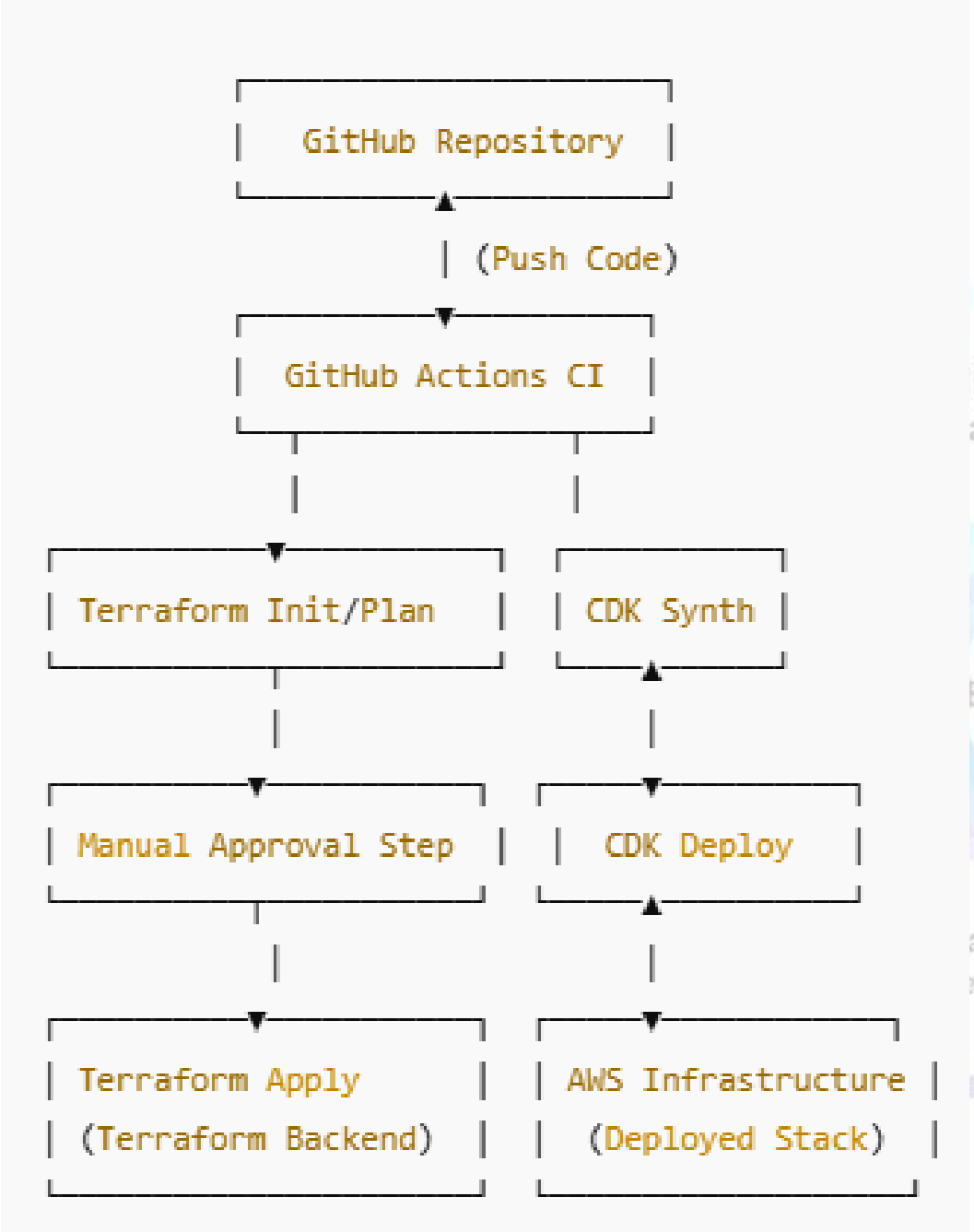
🚀 AWS CodePipeline Architecture (Terraform & CDK)



Key Features:

✅ Uses **AWS CodeCommit** for source control.
✅ Runs Terraform/CDK via **AWS CodeBuild**.
✅ Uses **S3 + DynamoDB** for Terraform state locking.
✅ Supports **manual approval before apply/deploy**.

⚡ GitHub Actions Workflow (Terraform & CDK)



Key Features:

✅ Uses **GitHub** for source control.

✅ Runs Terraform/CDK in **GitHub Actions**.

✅ Uses **AWS OIDC** (no stored credentials).

✅ Supports **manual approval before apply/deploy**.

Comparison

Feature	AWS CodePipeline	GitHub Actions
Fully AWS-managed	✅ Yes	❌ No
Manual Approval	✅ Yes	✅ Yes (With Workaround)
Terraform State	✅ S3 + DynamoDB	✅ S3 + DynamoDB
Security	✅ AWS IAM	⚠️ AWS OIDC
Multi-cloud Support	❌ No	✅ Yes
Speed	🚀 Faster in AWS	⚡ More Flexible

THANK YOU

LINKEDIN: WWW.LINKEDIN.COM/IN/MACHAN-VISHAL

MAIL VISHALMACHAN15@GMAIL.COM

CALL (217)5889385

BY VISHAL MACHAN