High-Level Design: Azure DevOps + Terraform + CI/CD Pipelines

# Purpose

This document describes a repeatable, secure, and scalable way to manage cloud infrastructure using Terraform as the IaC tool and Azure DevOps for source control, CI (build), and CD (release/pipeline orchestration).

# Audience

* - Cloud Architects and Infrastructure Engineers
* - DevOps Engineers implementing CI/CD pipelines
* - Security Engineers reviewing IAM/policy controls
* - Platform Engineering teams building shared modules and patterns
* - Release Managers and Operations staff who will run and support the pipelines

# Introduction

Modern infrastructure teams use IaC to increase repeatability, reduce configuration drift, and enable auditability. Azure DevOps provides Git repos, pipelines (YAML), artifact storage, and release gates. Terraform provides provider-agnostic code, modules, and state management. Together these tools let teams develop, review, test and deploy infrastructure in an automated and secure manner.

# Architecture and Design

Logical flow:

* 1. Developer creates/updates Terraform code in a feature branch.

2. PR triggers pipeline: linting, formatting check, security scanning, and `terraform plan`.

3. After approval, changes are applied to non-prod and later promoted to prod with approvals.

4. Terraform state is stored in S3 with DynamoDB locking.

Pipeline Stages:

* - Lint & Format
* - Security Scan
* - Plan
* - Policy Evaluation
* - Apply (Dev → Prod)
* - Monitoring & Audit

State Mgmt: S3 + DynamoDB locking (with versioning).

Auth: OIDC or STS tokens preferred over long-lived credentials.

# Components and Services

Core AWS Services:

* - VPC, IAM, S3, DynamoDB, EC2, EKS/ECS, RDS, ECR, Lambda, API Gateway, CloudWatch, CloudTrail, SSM, Route53, KMS, SNS/SQS.

Infra tooling:

* - S3 for state
* - DynamoDB for locking
* - CloudWatch for logs

Optional cross-account:

* - AWS Organizations, SCPs, Control Tower

# Screenshots

Recommended screenshots:

* 1. Azure DevOps repo structure

2. Azure Pipelines run timeline

3. Pipeline YAML editor

4. AWS S3 bucket (Terraform state)

5. DynamoDB lock table

6. CloudWatch dashboard

7. Architecture diagram

# Observability (Backup and Restore)

* - State & code backups: S3 versioning, DynamoDB PITR, Git history
* - Resource-level backups: RDS snapshots, EBS snapshots, S3 Glacier, CRR
* - Monitoring: CloudWatch, CloudTrail, SNS alerts
* - DR Runbooks: restore Terraform state, RDS snapshots, recreate infra
* - DR Testing: periodic drills

# Common Services

Shared Services:

* - Shared networking (Transit Gateway)
* - Central logging & monitoring
* - Security baseline module
* - Terraform module registry

Best Practices:

* - Naming/tagging conventions
* - Least privilege roles
* - Immutable infra where possible
* - Drift detection with scheduled plans
* - Policy as code (OPA, tfsec)
* - Pipeline isolation per environment

# Appendix A – Example Pipeline

Example Azure Pipelines YAML:

trigger:

branches:

include: [ main, develop ]

stages:

* - stage: LintAndScan

jobs:

* - job: Lint

steps:

* - script: |

terraform fmt -check

tflint --init

tflint

displayName: 'Format & Lint'

* - job: SecurityScan

dependsOn: Lint

steps:

* - script: |

tfsec .

displayName: 'Security scan'

* - stage: Plan

dependsOn: LintAndScan

jobs:

* - job: TerraformPlan

steps:

* - script: |

terraform init -backend-config="bucket=$(TF\_STATE\_BUCKET)" -backend-config="region=$(AWS\_REGION)"

terraform plan -out=tfplan

displayName: 'Terraform Init & Plan'

* - publish: tfplan

artifact: tfplan

* - stage: Apply

condition: and(succeeded(), eq(variables['Build.SourceBranch'], 'refs/heads/main'))

jobs:

* - deployment: TerraformApply

environment: 'production'

strategy:

runOnce:

deploy:

steps:

* - download: current

artifact: tfplan

* - script: |

terraform apply tfplan

displayName: 'Apply plan'

# Appendix B – Roles & Permissions

* - CI Pipeline Role (assume role) with scoped AWS permissions (S3, DynamoDB, EC2/EKS as needed)
* - Read-only role for plan-only jobs
* - Admin role (restricted use for emergencies)

# Example Azure DevOps Pipeline (Full YAML)

trigger:  
 branches:  
 include:  
 - main  
 - develop  
  
stages:  
- stage: LintAndScan  
 displayName: "Lint & Security Scan"  
 jobs:  
 - job: Lint  
 displayName: "Terraform Lint"  
 pool:  
 vmImage: 'ubuntu-latest'  
 steps:  
 - script: |  
 terraform fmt -check  
 tflint --init  
 tflint  
 displayName: 'Check formatting & lint code'  
  
 - job: Security  
 displayName: "Security Scan"  
 dependsOn: Lint  
 pool:  
 vmImage: 'ubuntu-latest'  
 steps:  
 - script: |  
 tfsec .  
 checkov -d .  
 displayName: 'Run tfsec & Checkov'  
  
- stage: Plan  
 displayName: "Terraform Plan"  
 dependsOn: LintAndScan  
 jobs:  
 - job: Plan  
 pool:  
 vmImage: 'ubuntu-latest'  
 steps:  
 - script: |  
 terraform init -backend-config="bucket=$(TF\_STATE\_BUCKET)" -backend-config="region=$(AWS\_REGION)" -backend-config="dynamodb\_table=$(TF\_STATE\_LOCK\_TABLE)"  
 terraform plan -out=tfplan  
 displayName: 'Terraform Init & Plan'  
 - publish: tfplan  
 artifact: tfplan  
  
- stage: Apply  
 displayName: "Terraform Apply"  
 dependsOn: Plan  
 condition: and(succeeded(), eq(variables['Build.SourceBranch'], 'refs/heads/main'))  
 jobs:  
 - deployment: Apply  
 environment: 'production'  
 strategy:  
 runOnce:  
 deploy:  
 steps:  
 - download: current  
 artifact: tfplan  
 - script: |  
 terraform apply -auto-approve tfplan  
 displayName: 'Terraform Apply'