Lab 17: CI/CD with Terraform using GitHub Actions

Author: Dr. Sandeep Kumar Sharma
Level: Advanced
Platform: Ubuntu Linux + Microsoft Azure + GitHub
Prerequisite: Lab 1 to Lab 16

Learning Objective

Participants will learn:

- What CI/CD means in Terraform
- Why CI/CD is needed for Infrastructure as Code
- How Terraform fits into DevOps pipelines
- How GitHub Actions works
- How to automate Terraform using GitHub Actions
- How to build a Terraform CI/CD pipeline

Learning Outcome

After completing this lab, participants will:

- Build automated Terraform pipelines
- Use GitHub Actions for IaC
- Implement DevOps practices
- Automate infrastructure deployment
- Apply Terraform in real production workflow

Concept Explanation

What is CI/CD in Terraform?

CI/CD in Terraform means:

- CI (Continuous Integration):
- Validate Terraform code
- Format code

- Plan infrastructure

- CD (Continuous Deployment):

- Apply infrastructure changes automatically

Terraform + CI/CD = Automated Infrastructure

Why CI/CD is Needed for Terraform

- No manual deployments
- No human error
- Standard process
- Controlled changes
- Audit trail
- Approval flow
- Enterprise compliance

Architecture

```
Developer → GitHub Repo → GitHub Actions → Terraform → Azure
```

Hands-On Lab

Step 1: Create GitHub Repository

Repository name:

```
terraform-azure-cicd
```

Step 2: Clone Repository

```
git clone https://github.com/<your-username>/terraform-azure-cicd.git
cd terraform-azure-cicd
```

Step 3: Add Terraform Code

Create `main.tf`:

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "rg" {
  name     = "rg-cicd-demo"
  location = "East US"
}
```

GitHub Actions Pipeline

Step 4: Create Workflow Folder

```
mkdir -p .github/workflows
```

Step 5: Create Workflow File

```
touch .github/workflows/terraform.yml
nano .github/workflows/terraform.yml
```

Step 6: GitHub Actions YAML

```
name: Terraform CI/CD Pipeline

on:
  push:
    branches:
      - main
  pull_request:

jobs:
  terraform:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Code
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2

      - name: Terraform Init
        run: terraform init

      - name: Terraform Format
        run: terraform fmt -check

      - name: Terraform Validate
        run: terraform validate

      - name: Terraform Plan
        run: terraform plan

      - name: Terraform Apply
        if: github.ref == 'refs/heads/main'
        run: terraform apply -auto-approve
```

Azure Authentication

Step 7: Create Azure Service Principal

```
az ad sp create-for-rbac --name terraform-cicd-sp --role Contributor --scopes /
subscriptions/<SUBSCRIPTION_ID>
```

Output gives:

- clientId
- clientSecret
- tenantId
- subscriptionId

Step 8: Add GitHub Secrets

In GitHub repo → Settings → Secrets → Actions

Add:

```
ARM_CLIENT_ID
ARM_CLIENT_SECRET
ARM_TENANT_ID
ARM_SUBSCRIPTION_ID
```

Authentication Config in Pipeline

Add env to workflow:

```
env:
  ARM_CLIENT_ID: ${{ secrets.ARM_CLIENT_ID }}
  ARM_CLIENT_SECRET: ${{ secrets.ARM_CLIENT_SECRET }}
  ARM_TENANT_ID: ${{ secrets.ARM_TENANT_ID }}
  ARM_SUBSCRIPTION_ID: ${{ secrets.ARM_SUBSCRIPTION_ID }}
```

Full Pipeline Flow

```
Code Push → GitHub Actions Trigger → Terraform Init → Plan → Apply → Azure Infra
Created
```

Verification

- Go to GitHub Actions tab
- See pipeline running
- Check Azure Portal
- Resource Group `rg-cicd-demo` created

Enterprise Model

```
Git → CI/CD → Terraform → Azure
```

Cleanup

Delete resource using commit:

```
# remove resource block
```

Push code → pipeline runs → Terraform destroy logic (manual or pipeline-based)

Professional Practice

In real companies:

- Plan in PR
- Apply after approval
- Manual gates
- Policy checks
- Security scanning
- Environment pipelines