Problem

Managing infrastructure across multiple environments, such as development, staging, and production, is challenging. Organizations often face issues like environment drift, manual errors, and difficulties in tracking changes across deployment pipelines. Traditional methods lack consistency, traceability, and automation, leading to inefficiencies and increased risks.

Solution

We address these challenges by using Terraform and GitHub Actions to implement a multienvironment infrastructure deployment pipeline. This GitOps-style workflow automatically plans and applies infrastructure changes to different environments based on Git branches. It ensures consistency, automation, and traceability throughout the pipeline.

Implementation

1. Terraform Configuration for Multiple Environments

We structure our Terraform configuration to support multiple environments by using Terraform workspaces and modular design.

```
# environments/main.tf

locals {
  environment = terraform.workspace
}

module "network" {
  source      = "../modules/network"
  environment = local.environment
}

module "compute" {
  source      = "../modules/compute"
  environment = local.environment
}
```

```
# Add additional modules as needed
```

- The `terraform.workspace` local variable dynamically sets the environment based on the selected workspace.
- We use modules (`network`, `compute`, etc.) to standardize configurations across environments.

---

2. GitHub Actions Workflow

We create a `.github/workflows/terraform.yml` file to define a CI/CD pipeline for infrastructure deployment.

```yaml
name: 'Terraform CI/CD'

on:
  push:
    branches:
      - main
      - staging
      - development
  pull_request:
    branches: [ main ]

jobs:
  terraform:
    name: 'Terraform'
    runs-on: ubuntu-latest
    env:
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}

    steps:
    - name: Checkout
      uses: actions/checkout@v2

    - name: Setup Terraform
      uses: hashicorp/setup-terraform@v1

    - name: Terraform Init
      run: terraform init
      working-directory: ./environments
    - name: Terraform Format
      run: terraform fmt -check
      working-directory: ./environments

    - name: Terraform Plan
      run: |
        terraform workspace select ${GITHUB_REF##*/} || terraform workspace
new ${GITHUB_REF##*/}
        terraform plan -no-color
      working-directory: ./environments

    - name: Terraform Apply
      if: github.ref == 'refs/heads/main' && github.event_name == 'push'
      run: terraform apply -auto-approve
      working-directory: ./environments
```

Key Features:

- **Branch-based Environments:**
  - Branches (`main`, `staging`, `development`) trigger the workflow.
  - Workspaces are created or selected dynamically based on branch names.
- **Automated Planning and Application:**
  - Pushes to the `main` branch automatically apply infrastructure changes.
  - Other branches require manual approval or code reviews before merging.

---

3. GitHub Repository Setup

- **Branch Protection Rules:**
  We enable pull request reviews for the `main` branch to ensure changes are reviewed before deployment.
- **GitHub Secrets:**
  AWS credentials (`AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`) are stored securely as GitHub Secrets.
- **Commit and Push:**
  Once the configuration is set up, we commit and push the Terraform files and workflow to the Git repository.

Discussion

Using Terraform and GitHub Actions for multienvironment infrastructure deployment offers several benefits:

1. **Environment Consistency**
   By sharing Terraform configurations across environments, we ensure consistent infrastructure and reduce environment-specific issues.
2. **Automated Workflow**
   The GitHub Actions pipeline automates planning and applying changes, reducing manual intervention and minimizing errors.
3. **Environment Isolation**
   Terraform workspaces isolate the state files for each environment, preventing unintended cross-environment changes.
4. **Change Traceability**
   All changes are tracked in Git, providing a clear audit trail for infrastructure modifications.
5. **Controlled Promotions**
   Using branches to represent environments allows changes to be tested in lower environments (e.g., development, staging) before being promoted to production.
6. **Pull Request Reviews**
   Code reviews ensure that all infrastructure changes are vetted before being applied to critical environments like production.

Best Practices

When implementing this solution, we follow these best practices:

- **Consistent Naming:**
  Use a clear naming convention for resources, incorporating the environment name as a prefix or suffix.
- **Parameterize Values:**
  Manage environment-specific values using Terraform variables for flexibility and reusability.
- **Access Controls:**
  Secure GitHub repository access and AWS IAM permissions to maintain security.
- **State Cleanup:**
  Regularly clean up old Terraform workspaces to avoid unnecessary state files.

- **Drift Detection:**
  Run `terraform plan` periodically on all environments to detect and address manual changes or drift.
- **Cost Optimization:**
  Optimize costs for nonproduction environments by using smaller instance types or fewer resources.
- **Compliance:**
  Ensure the infrastructure and pipeline meet organizational or industry compliance requirements.

Summary

By leveraging Terraform and GitHub Actions, we create a robust, automated, and version-controlled approach to managing infrastructure across multiple environments. This aligns with GitOps principles and modern DevOps practices, providing a scalable and efficient solution for infrastructure management. It ensures consistency, traceability, and reduced risk across the entire deployment pipeline.