CI/CD Pipeline Architecture for Azure DevOps

This design outlines a CI/CD pipeline in Azure DevOps for automated builds, unit testing, security checks, quality gates, and multi-environment deployments (Dev, QA, Prod).

Components:

1. **Source Code (Git):**
    - The codebase resides in a Git repository.
2. **Azure Repos:**
    - Azure Repos hosts the Git repository.
3. **Trigger (Push/PR):**
    - The pipeline is triggered by code pushes to specific branches (e.g., `main`, `release/*`) or pull requests.
4. **Build and Unit Tests:**
    - Compiles the source code and executes unit tests.
    - Generates build artifacts.
5. **Security & Quality Checks:**
    - Performs security scans (e.g., static analysis, vulnerability scanning).
    - Executes code quality analysis (e.g., SonarCloud).
6. **Quality Gate:**
    - Checks if the code meets predefined quality standards.
    - Fails the pipeline if the quality gate fails.
7. **Deploy to Dev:**
    - Deploys the build artifacts to the development environment.
8. **Deploy to QA:**
    - Deploys the build artifacts to the quality assurance environment.
9. **Manual Approval (Prod):**
    - Requires manual approval before deploying to the production environment.
10. **Deploy to Prod:**
    - Deploys the build artifacts to the production environment.
11. **Artifacts:**
    - Represents the build artifacts produced by the build stage.
12. **Artifact Storage:**
    - Azure Artifacts, or the build artifact staging directory, stores the generated artifacts.
13. **Pipeline Failed:**
    - Represents the state when the pipeline fails due to quality gate failures, approvals being rejected or other build/deployment issues.
14. **Environments:**
    - Represent the different deployment environments (Dev, QA, Prod).
15. **Security & Quality:**
    - Represents the security and quality check stages.
16. **Build & Test:**
    - Represents the build and unit testing stage.

Pipeline Stages:

1. **Build and Unit Tests:**
    - Compiles the code.
    - Runs unit tests.
    - Generates and publishes build artifacts.
2. **Security and Quality Checks:**
    - Performs static analysis (e.g., SonarCloud).
    - Runs vulnerability scans.
    - Checks quality gates.
3. **Deploy to Dev:**
    - Deploys the application to the development environment.

4. **Deploy to QA:**
    - Deploys the application to the quality assurance environment.
5. **Deploy to Prod:**
    - Requires manual approval before deployment.
    - Deploys the application to the production environment.

Implementation Details (Azure DevOps YAML):

- Use YAML pipelines for declarative configuration.
- Employ variable groups for managing environment-specific variables.
- Utilize templates for reusable pipeline components.
- Use environments within azure devops to track deployments.
- Implement manual approvals using the `ManualValidation` task.
- Integrate SonarCloud for code quality checks.
- Use Azure Web Apps or other appropriate deployment tasks for each environment.
- Use service connections to securely connect to Azure resources.
- Use dynamic yaml files to provide environment specific variables.

Azure DevOps YAML (azure-pipelines.yml):

```
# azure-pipelines.yml

trigger:
  branches:
    include:
      - main
      - release/*
  paths:
    include:
      - 'terraform/*'
      - 'tests/*'
      - 'azure-pipelines.yml'
      - 'deployments/*'

variables:
  - group: TerraformVariables # Store common variables
  - name: BuildConfiguration
    value: 'Release'

stages:
```

```yaml
- stage: BuildAndTest
  displayName: Build and Unit Tests
  jobs:
    - job: Build
      displayName: Terraform Build and Test
      pool:
        vmImage: 'ubuntu-latest'
      steps:
        # (Terraform build and test steps remain the same as previous example)
        - task: TerraformInstaller@0
          displayName: 'Install Terraform'
          inputs:
            terraformVersion: '$(terraformVersion)'
        - task: TerraformCLI@0
          displayName: 'Terraform Format'
          inputs:
            command: 'fmt'
            workingDirectory: 'terraform'
            arguments: '-check'
        - task: TerraformCLI@0
          displayName: 'Terraform Validate'
          inputs:
            command: 'validate'
            workingDirectory: 'terraform'
        - script: |
            go test -v ./tests/...
          displayName: 'Run Terratest'
        - task: TerraformCLI@0
          displayName: 'Terraform Plan'
          inputs:
            command: 'plan'
            workingDirectory: 'terraform'
            environmentServiceName: '$(devServiceConnection)'
            commandOptions: '-out=plan.out'
        - task: PublishPipelineArtifact@1
          displayName: 'Publish Terraform Plan'
          inputs:
            targetPath: 'terraform/plan.out'
            artifact: 'terraformPlan'

- stage: SecurityAndQuality
  displayName: Security and Quality Checks
  dependsOn: BuildAndTest
  jobs:
    - job: SecurityScan
      displayName: Terraform Security Scan
      pool:
        vmImage: 'ubuntu-latest'
      steps:
        # (Security scan steps remain the same as previous example)
        - task: TerraformInstaller@0
          displayName: 'Install Terraform'
          inputs:
            terraformVersion: '$(terraformVersion)'
```

```
        - script: |
            tfsec ./terraform
          displayName: 'Run tfsec'
        - script: |
            checkov -d ./terraform
          displayName: 'Run Checkov'

  - stage: DeployDev
    displayName: Deploy to Dev
    dependsOn: SecurityAndQuality
    variables:
      - template: deployments/dev.yml # Dynamic YAML for Dev
    jobs:
      - template: deployments/deploy-template.yml
        parameters:
          environmentName: 'Dev'
          serviceConnection: '$(devServiceConnection)'

  - stage: DeployQA
    displayName: Deploy to QA
    dependsOn: DeployDev
    variables:
      - template: deployments/qa.yml # Dynamic YAML for QA
    jobs:
      - template: deployments/deploy-template.yml
        parameters:
          environmentName: 'QA'
          serviceConnection: '$(qaServiceConnection)'

  - stage: DeployProd
    displayName: Deploy to Prod
    dependsOn: DeployQA
    variables:
      - template: deployments/prod.yml # Dynamic YAML for Prod
    jobs:
      - template: deployments/deploy-template.yml
        parameters:
          environmentName: 'Prod'
          serviceConnection: '$(prodServiceConnection)'
          approvalRequired: true

Deployment Template (deployments/deploy-template.yml):
# deployments/deploy-template.yml
parameters:
  environmentName: ''
  serviceConnection: ''
  approvalRequired: false
jobs:
  - job: Deploy
    displayName: 'Terraform Apply ${{ parameters.environmentName }}'
    pool:
      vmImage: 'ubuntu-latest'
    environment: ${{ parameters.environmentName }}
    ${{ if and(parameters.approvalRequired, eq(parameters.environmentName, 'Prod')) }}:
```

```
    strategy:
      runOnce:
        preDeploy:
          steps:
            - task: ManualValidation@0
              timeoutInMinutes: 1440
              inputs:
                notifyUsers: 'your-email@example.com'
                instructions: 'Please validate the Terraform deployment to Prod.'
  steps:
    - task: TerraformInstaller@0
      displayName: 'Install Terraform'
      inputs:
        terraformVersion: '$(terraformVersion)'
    - task: DownloadPipelineArtifact@2
      displayName: 'Download Terraform Plan'
      inputs:
        artifact: 'terraformPlan'
        path: '$(System.ArtifactsDirectory)/terraform'
    - task: TerraformCLI@0
      displayName: 'Terraform Apply ${{ parameters.environmentName }}'
      inputs:
        command: 'apply'
        workingDirectory: 'terraform'
        environmentServiceName: '${{ parameters.serviceConnection }}'
        commandOptions: 'plan.out'
```

Dynamic YAML Files (Example deployments/dev.yml):

```
# deployments/dev.yml

variables:
  # Environment-specific variables
  resourceGroupName: 'dev-rg'
  storageAccountName: 'devstorage'
  # ... other variables
```

Key points:

- **Dynamic YAML:** Environment-specific variables are stored in separate YAML files, making the pipeline more modular and maintainable.
- **Deployment Template:** A reusable deployment template reduces code duplication and ensures consistency across environments.

This dynamic YAML approach enhances the Terraform CI/CD pipeline, providing a flexible and scalable solution for managing infrastructure deployments. Remember to adjust the variables and configurations to match your specific project requirements.

Key Considerations:

- **Variable Groups:** Use Azure DevOps variable groups to store environment-specific variables and secrets.
- **Service Connections:** Configure service connections for each environment to authenticate with Azure.

- **Terraform State:** Store Terraform state in Azure Storage or Terraform Cloud for collaboration and consistency.
- **Terratest:** Write comprehensive unit tests using Terratest to validate Terraform configurations.
- **Security Scanning:** Integrate security scanning tools like `tfsec` and `Checkov` to identify potential vulnerabilities.
- **Quality Gates:** Implement custom quality checks to enforce coding standards and best practices.
- **Manual Approval:** Require manual approval for production deployments to minimize risk.
- **Terraform Cloud:** Consider using Terraform Cloud for remote state management, collaboration, and policy enforcement.
- **Environments:** Utilize Azure DevOps environments feature for better tracking of deployments.

This setup provides a robust and automated Terraform CI/CD pipeline, enabling efficient and secure infrastructure deployments.

Terraform code to set up the necessary Azure resources for a Dev environment CI/CD pipeline in Azure DevOps. It includes:

1. **Resource Group:** To organize resources.
2. **Azure Storage Account:** To store Terraform state.
3. **Azure Key Vault:** To store secrets.
4. **Azure DevOps Service Connection:** To grant Azure DevOps pipeline access to Azure resources.