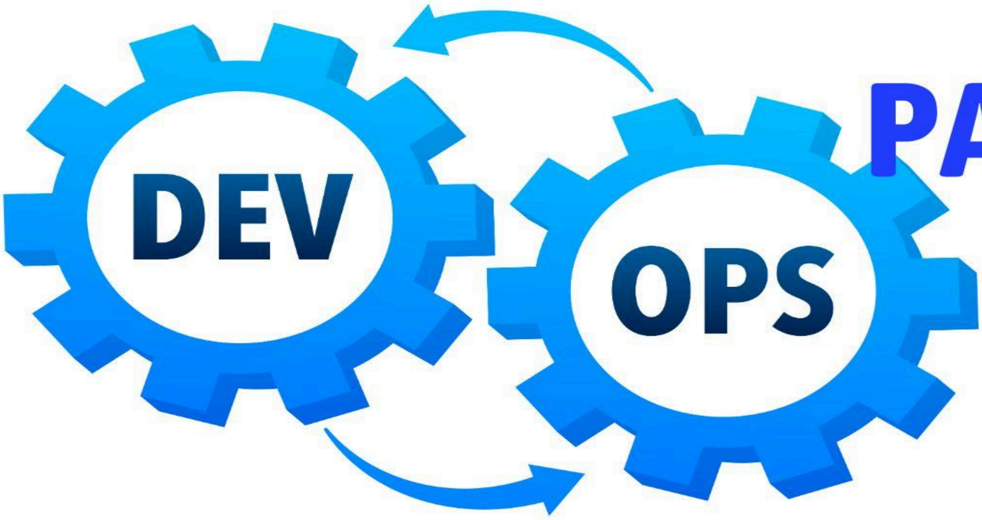
CAREER BYTE CODE
REALTIME PROJECTS PLATFORM

91 COUNTRIES
subscriber
241k Learners
+32 471 40 89 08
CAREERBYTECODE.SUBSTACK.COM

DEVOPS

REALTIME PROJECTS

PART 2



careerbytecode.substack.com
+32 - 471408908

500+ RealTime Projects platform

Project 1: Azure DevOps Pipeline for Automated Database Schema Deployment

---

1. Project Scope

This project focuses on automating the deployment of database schema updates using **Azure DevOps Pipelines**. When a developer updates the database schema (e.g., adding a new column, modifying a table), the pipeline will automatically validate and apply the changes to the database.

Key Features:

✅ Automate database schema changes with version control
✅ Ensure safe deployments with pre-deployment validation
✅ Implement rollback in case of failure
✅ Improve collaboration between Dev and DB teams
✅ Ensure consistency across environments (Dev, QA, Staging, Prod)

---

2. Tools Used

- **Azure DevOps Pipelines** – Automates the deployment process
- **Azure SQL Database** – Target database for schema changes
- **SQL Server Data Tools (SSDT)** – Schema validation
- **DACPAC (Data-tier Application Package)** – Schema deployment format
- **SQLCMD** – Executes SQL scripts
- **Azure Key Vault** – Securely stores database credentials
- **PowerShell** – Automates database operations

---

3. Analysis Approach

Challenges Without Automation

❌ Manual schema updates can introduce errors
❌ Developers may forget to apply updates to all environments
❌ Schema drift (inconsistencies across databases)
❌ Rollback is time-consuming

Proposed Solution

✅ Use Azure DevOps Pipelines to automate schema deployment
✅ Validate SQL scripts before applying them
✅ Implement rollback strategy for failed deployments
✅ Use DACPAC to track schema changes and maintain consistency

4. Step-by-Step Implementation

Step 1: Create an Azure DevOps Repository

1. Go to **Azure DevOps** → **Repos** → **New Repository**
2. Clone the repo and create a `Database` folder
3. Add a sample SQL project (`.sqlproj`)

Step 2: Develop and Store SQL Scripts

1. Inside the `Database` folder, create:
   - `Tables/Users.sql`
   - `Tables/Orders.sql`
   - `Procedures/UpdateUserDetails.sql`
2. Use SQL Server Data Tools (SSDT) to package `.sqlproj` into a **DACPAC** file

Step 3: Create a CI Pipeline for SQL Validation

1. Navigate to **Azure DevOps** → **Pipelines** → **New Pipeline**
2. Choose **Azure Repos Git**
3. Add the following YAML for schema validation:

```
Unset
trigger:

  branches:

    include:

      - main


pool:

  vmImage: 'windows-latest'


steps:

- task: VSBuild@1

  displayName: 'Build SQL Database Project'
```

```
  inputs:

    solution: '**/*.sqlproj'

    platform: 'Any CPU'

    configuration: 'Release'


- task: PublishBuildArtifacts@1

  displayName: 'Publish DACPAC File'

  inputs:

    pathToPublish: '$(Build.ArtifactStagingDirectory)'

    artifactName: 'DACPAC'
```

Step 4: Create a CD Pipeline for Deployment

1. Go to **Azure DevOps** → **Releases** → **New Release Pipeline**
2. Select **Azure SQL Database Deployment**
3. Configure the task with:
    - **Azure Subscription:** Link your Azure account
    - **Database Type:** Azure SQL Database
    - **Deploy Method:** DACPAC
4. Modify the pipeline YAML:

Unset
```
trigger:

  branches:

    include:

      - main



pool:
```

```
  vmImage: 'windows-latest'


steps:
- task: SqlAzureDacpacDeployment@1
  inputs:
    azureSubscription: 'MyAzureSubscription'
    serverName: 'mydatabase.database.windows.net'
    databaseName: 'MyDB'
    authenticationType: 'servicePrincipal'
    dacpacFile: '$(Build.ArtifactStagingDirectory)/DACPAC/*.dacpac'
```

Step 5: Implement Rollback Strategy

If deployment fails:

- Restore the previous DACPAC using:

```
Unset
sqlpackage.exe /Action:Publish
/SourceFile:"$(Build.ArtifactStagingDirectory)/PreviousDACPAC.dacpac"
```

---

5. Conclusion

🚀 **Key Benefits of Automated Database Deployment:** ✅ No manual intervention needed
✅ Schema remains consistent across environments
✅ Reduced risk of human error
✅ Easy rollback in case of issues

---

6. Real-Time Example

A **financial services company** with multiple databases can use this approach to ensure **schema consistency**, prevent manual errors, and **deploy changes securely** without downtime.

Project 2: Implementing Secure DevOps with Azure Policy and Compliance as Code

---

1. Project Scope

This project focuses on **automating security compliance** using **Azure Policy and Compliance as Code** within **Azure DevOps Pipelines**. The goal is to enforce security rules across cloud resources, ensuring that every deployment follows organizational policies.

Key Features:

✅ **Automate compliance checks** for Azure resources
✅ **Prevent non-compliant deployments**
✅ **Use Azure DevOps Pipelines** to apply policies automatically
✅ **Monitor compliance in real-time**
✅ **Ensure governance across multiple subscriptions**

---

2. Tools Used

- **Azure Policy** - Defines and enforces compliance rules
- **Azure DevOps Pipelines** - Automates policy deployment
- **Azure CLI / PowerShell** - Manages policy definitions
- **Azure Monitor & Compliance Center** - Tracks violations
- **Terraform / Bicep** - Deploys policies as code
- **Azure Key Vault** - Stores secure credentials

---

3. Analysis Approach

Challenges Without Compliance Automation

❌ Cloud misconfigurations lead to security risks
❌ Developers may deploy resources without following security guidelines
❌ Manual audits are time-consuming and error-prone
❌ Lack of **visibility** into non-compliant resources

Proposed Solution

✅ Use **Azure Policy as Code** to enforce security best practices
✅ Integrate **Azure DevOps Pipelines** to apply policies at every deployment
✅ Monitor violations using **Azure Security Center**
✅ Automate **remediation** for non-compliant resources

---

4. Step-by-Step Implementation

Step 1: Create a Policy Definition in Azure

1. Open **Azure Portal** → **Policy**
2. Click **Definitions** → **Add Policy Definition**
3. Use the following JSON to prevent public IP creation:

```
Unset
{
  "mode": "All",
  "policyRule": {
    "if": {
      "field":
"Microsoft.Network/publicIPAddresses/publicIPAllocationMethod",
      "equals": "Static"
    },
    "then": {
      "effect": "Deny"
    }
  }
}
```

4. Click **Save** and **Assign Policy** to a Subscription.

Step 2: Automate Policy Deployment Using Bicep in Azure DevOps

1. Create a **Bicep file (`policy.bicep`)** for policy definition:

Unset

```
resource policyDef
'Microsoft.Authorization/policyDefinitions@2020-09-01' = {

  name: 'deny-public-ip'

  properties: {

    displayName: 'Deny Public IP'

    policyType: 'Custom'

    mode: 'All'

    policyRule: {

      if: {

        field:
'Microsoft.Network/publicIPAddresses/publicIPAllocationMethod'

        equals: 'Static'

      }

      then: {

        effect: 'Deny'

      }

    }

  }

}
```

Step 3: Create an Azure DevOps Pipeline for Policy Deployment

1. Navigate to **Azure DevOps** → **Pipelines** → **New Pipeline**
2. Select **Azure Repos Git**
3. Create a new **YAML pipeline**:

Unset
```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: AzureCLI@2
  displayName: 'Deploy Policy using Bicep'
  inputs:
    azureSubscription: 'MyAzureSubscription'
    scriptType: 'bash'
    scriptLocation: 'inlineScript'
    inlineScript: |
      az deployment sub create --location eastus --template-file
policy.bicep
```

Step 4: Validate Policy Enforcement

1. **Test non-compliance** by trying to create a public IP manually:

Unset

```
az network public-ip create --resource-group MyRG --name MyPublicIP
--allocation-method Static
```

2. If the policy is applied correctly, **Azure will deny the request**.

Step 5: Automate Compliance Reporting

1. Enable **Azure Monitor Logs** to track violations:

Unset

```
az policy state list --query "[?complianceState=='NonCompliant']"
```

2. Send alerts when non-compliance is detected using **Azure Monitor**.

---

5. Conclusion

🚀 **Key Benefits of Azure Policy Automation:**
✅ **Prevents security misconfigurations before deployment**
✅ **Ensures compliance with organizational policies**
✅ **Automates policy enforcement with DevOps workflows**
✅ **Provides real-time monitoring and alerts for violations**

---

6. Real-Time Example

A **healthcare company** handling sensitive data can use **Azure Policy as Code** to enforce **HIPAA compliance**, ensuring that no resources expose data to the public cloud.

Project 3: Implementing Multi-Stage CI/CD Pipeline with Azure DevOps for Microservices Deployment

---

1. Project Scope

This project focuses on setting up a **multi-stage CI/CD pipeline** in **Azure DevOps** to deploy a **microservices-based application** to **Azure Kubernetes Service (AKS)**. Each microservice will be built, tested, and deployed **independently** using a **multi-stage pipeline**.

Key Features:

✅ **Automate build, test, and deployment** of microservices
✅ **Use Docker & Kubernetes** for containerized deployment
✅ **Implement Canary & Rolling updates**
✅ **Manage configurations with Helm**
✅ **Secure secrets using Azure Key Vault**

---

2. Tools Used

- **Azure DevOps Pipelines** - Automate CI/CD
- **Azure Kubernetes Service (AKS)** - Host containerized microservices
- **Docker & Azure Container Registry (ACR)** - Store images
- **Helm** - Manage Kubernetes deployments
- **Azure Key Vault** - Store sensitive configurations
- **Prometheus & Grafana** - Monitor microservices

---

3. Analysis Approach

Challenges Without a Multi-Stage Pipeline

❌ Microservices deployments are inconsistent
❌ Manual testing increases deployment delays
❌ Lack of automated rollback increases downtime
❌ Security risks due to hardcoded secrets

Proposed Solution

✅ Implement a **multi-stage pipeline** with **separate build, test, and deploy stages**
✅ Automate deployment using **Helm & Kubernetes**
✅ Secure configurations using **Azure Key Vault**
✅ Enable **rolling updates** for zero downtime

---

4. Step-by-Step Implementation

Step 1: Create an AKS Cluster

1. **Provision an Azure Kubernetes Service (AKS) Cluster:**

```
Unset
az group create --name MyResourceGroup --location eastus

az aks create --resource-group MyResourceGroup --name MyAKSCluster
--node-count 3 --generate-ssh-keys
```

2. **Connect to AKS Cluster:**

```
Unset
az aks get-credentials --resource-group MyResourceGroup --name
MyAKSCluster
```

Step 2: Set Up Azure DevOps Repository & Docker Configuration

1. **Create a repository in Azure DevOps**
2. **Add microservices source code** (`/microservices/user-service/`, `/microservices/order-service/`)
3. **Create a Dockerfile for each microservice** (Example: `user-service/Dockerfile`):

```
Unset
FROM node:14

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

CMD ["node", "server.js"]
```

4. **Push code to Azure Repos:**

```
git add .

git commit -m "Initial commit"

git push origin main
```

Step 3: Implement CI Pipeline for Building & Pushing Docker Images

1. **Go to Azure DevOps → Pipelines → New Pipeline**
2. **Create a YAML pipeline (`ci-pipeline.yml`):**

```
trigger:

  branches:

    include:

      - main



pool:

  vmImage: 'ubuntu-latest'



stages:

- stage: Build

  jobs:

  - job: Build

    steps:

    - task: Docker@2

      inputs:
```

```
        command: 'buildAndPush'
        repository: 'myacr.azurecr.io/user-service'
        dockerfile: 'microservices/user-service/Dockerfile'
        containerRegistry: 'MyAzureContainerRegistry'
        tags: '$(Build.BuildId)'

    - task: Docker@2
      inputs:
        command: 'buildAndPush'
        repository: 'myacr.azurecr.io/order-service'
        dockerfile: 'microservices/order-service/Dockerfile'
        containerRegistry: 'MyAzureContainerRegistry'
        tags: '$(Build.BuildId)'
```

Step 4: Implement CD Pipeline for Microservices Deployment

1. **Create a Kubernetes Deployment YAML (`k8s-deployment.yaml`):**

```
Unset
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
spec:
```

```
  replicas: 2
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
      - name: user-service
        image: myacr.azurecr.io/user-service:latest
        ports:
        - containerPort: 3000
```

2. **Modify the CD pipeline (`cd-pipeline.yml`):**

Unset
```
trigger:
  branches:
    include:
      - main


pool:
```

```
  vmImage: 'ubuntu-latest'


stages:
- stage: Deploy
  jobs:
  - job: Deploy
    steps:
    - task: Kubernetes@1
      inputs:
        connectionType: 'Azure Resource Manager'
        azureSubscription: 'MyAzureSubscription'
        azureResourceGroup: 'MyResourceGroup'
        kubernetesCluster: 'MyAKSCluster'
        namespace: 'default'
        command: 'apply'
        useConfigurationFile: true
        configuration: 'k8s-deployment.yaml'
```

Step 5: Secure Secrets Using Azure Key Vault

1. **Create an Azure Key Vault:**

Unset
```
az keyvault create --name MyKeyVault --resource-group MyResourceGroup
--location eastus
```

2. **Store API keys and database credentials:**

```
Unset
az keyvault secret set --vault-name MyKeyVault --name
"DBConnectionString" --value "Server=myserver;Database=mydb;"
```

3. **Modify Kubernetes deployment to use secrets:**

```
Unset
env:

- name: DB_CONNECTION_STRING

  valueFrom:

    secretKeyRef:

      name: azure-keyvault

      key: DBConnectionString
```

Step 6: Implement Rolling Updates & Rollback

1. **Enable rolling updates in Kubernetes:**

```
Unset
strategy:

  type: RollingUpdate

  rollingUpdate:

    maxUnavailable: 1

    maxSurge: 1
```

2. **Rollback if needed:**

Unset
```
kubectl rollout undo deployment user-service
```

---

5. Conclusion

🚀 **Key Benefits of Multi-Stage CI/CD Pipeline for Microservices:**
✅ **Automates deployments** across multiple environments
✅ **Enables faster releases with minimal downtime**
✅ **Enhances security by managing secrets centrally**
✅ **Improves scalability with Kubernetes**

---

6. Real-Time Example

A **large e-commerce platform** deploying multiple microservices (cart, orders, payments) can use **Azure DevOps Pipelines** to ensure **continuous delivery** with **zero downtime**.

Project 4: Implementing Azure DevOps CI/CD Pipeline for Infrastructure as Code (IaC) using Bicep

---

1. Project Scope

This project focuses on **Infrastructure as Code (IaC) with Bicep** to automate the provisioning and management of **Azure resources** using **Azure DevOps Pipelines**. Instead of using ARM templates, Bicep provides a **simpler syntax** while still leveraging Azure's native deployment capabilities.

Key Features:

✅ **Automate Infrastructure Deployment using Bicep**
✅ **Manage Infrastructure as Code (IaC) in Azure DevOps**
✅ **Implement Role-Based Access Control (RBAC) policies**
✅ **Ensure consistency across environments (Dev, QA, Prod)**
✅ **Enable rollback in case of failed deployments**

---

2. Tools Used

- **Azure DevOps Pipelines** – Automate deployments
- **Bicep** – Declarative IaC language for Azure
- **Azure CLI** – Manages Azure resources via scripts
- **Azure Key Vault** – Securely stores credentials
- **Azure Storage Account** – Stores deployment logs
- **PowerShell & YAML** – Automation scripting

---

3. Analysis Approach

Challenges Without Infrastructure as Code (IaC)

❌ Manual deployments are **error-prone** and time-consuming
❌ Difficult to **track** changes in infrastructure
❌ No **version control**, making rollback difficult
❌ Security risks due to **hardcoded credentials**

Proposed Solution

✅ Define **Azure infrastructure** as code using Bicep
✅ Automate **resource provisioning** via Azure DevOps
✅ Use **RBAC policies** to ensure secure deployments
✅ Store **secrets in Azure Key Vault**

---

4. Step-by-Step Implementation

Step 1: Create an Azure DevOps Repository for Bicep Code

1. Go to **Azure DevOps** → **Repos** → **New Repository**
2. Clone the repo and create a new directory:

```
Unset
git clone https://dev.azure.com/MyOrg/BicepDeployment.git

cd BicepDeployment

mkdir bicep-files
```

3. Inside the `bicep-files` folder, create a **Bicep template (`main.bicep`)** to deploy an Azure Storage Account:

```
Unset
resource storageAccount 'Microsoft.Storage/storageAccounts@2021-09-01'
= {

  name: 'myuniquestorageaccount'

  location: 'East US'

  sku: {

    name: 'Standard_LRS'

  }

  kind: 'StorageV2'

}
```

4. Commit and push the code to Azure Repos:

```
Unset
git add .

git commit -m "Added Bicep template"
```

```
git push origin main
```

Step 2: Configure an Azure DevOps CI Pipeline for Bicep Validation

1. Go to **Azure DevOps** → **Pipelines** → **New Pipeline**
2. Select **Azure Repos Git** as the source
3. Create a new **YAML pipeline** (**`bicep-ci.yml`**):

Unset
```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: AzureCLI@2
  displayName: 'Validate Bicep Template'
  inputs:
    azureSubscription: 'MyAzureSubscription'
    scriptType: 'bash'
    scriptLocation: 'inlineScript'
    inlineScript: |
```

```
az bicep build --file bicep-files/main.bicep
```

4. **Run the pipeline** to validate the Bicep syntax.

Step 3: Create an Azure DevOps CD Pipeline for Deployment

1. Navigate to **Azure DevOps** → **Pipelines** → **New Release Pipeline**
2. Select **Azure Resource Group Deployment**
3. Modify the pipeline YAML (`bicep-cd.yml`) for deployment:

Unset
```
trigger:

  branches:

    include:

      - main


pool:

  vmImage: 'ubuntu-latest'


steps:

- task: AzureCLI@2

  displayName: 'Deploy Bicep Template'

  inputs:

    azureSubscription: 'MyAzureSubscription'

    scriptType: 'bash'

    scriptLocation: 'inlineScript'
```

```
    inlineScript: |
      az deployment group create --resource-group MyResourceGroup
--template-file bicep-files/main.bicep
```

4. **Run the pipeline** to deploy resources automatically.

Step 4: Secure Infrastructure Using Azure Key Vault

1. **Create an Azure Key Vault:**

```
Unset
az keyvault create --name MyKeyVault --resource-group MyResourceGroup
--location eastus
```

2. **Store sensitive credentials in Key Vault:**

```
Unset
az keyvault secret set --vault-name MyKeyVault --name
"StorageAccountKey" --value "my-secure-key"
```

3. **Modify Bicep to retrieve secrets from Key Vault:**

```
Unset
param storageKey string =
resourceId('Microsoft.KeyVault/vaults/secrets', 'MyKeyVault',
'StorageAccountKey')
```

Step 5: Implement Role-Based Access Control (RBAC) Policies

1. **Assign "Contributor" role to DevOps pipeline:**

```
Unset
az role assignment create --assignee "<pipeline-service-principal>"
--role Contributor --scope
"/subscriptions/<subscription-id>/resourceGroups/MyResourceGroup"
```

2. **Modify Bicep to enforce RBAC roles:**

```
Unset
resource roleAssignment
'Microsoft.Authorization/roleAssignments@2020-10-01-preview' = {

  name: guid('MyResourceGroup', 'Contributor')

  properties: {

    roleDefinitionId:
'/providers/Microsoft.Authorization/roleDefinitions/<role-id>'

    principalId: '<pipeline-service-principal>'

  }

}
```

Step 6: Implement Rollback in Case of Failed Deployment

1. **Use Azure Deployment History to track changes:**

```
Unset
az deployment group show --resource-group MyResourceGroup --name latest
```

2. **Rollback to previous version if deployment fails:**

```
Unset
az deployment group create --resource-group MyResourceGroup --mode
Complete --template-file previous-template.bicep
```

---

5. Conclusion

🚀 **Key Benefits of Bicep for IaC in Azure DevOps:**
✅ **Simplifies Infrastructure as Code** with an easy syntax
✅ **Ensures consistent deployments** across multiple environments
✅ **Secures credentials** using **Azure Key Vault**
✅ **Provides rollback capabilities** for failed deployments

---

6. Real-Time Example

A **financial services company** managing multiple Azure environments (Dev, Staging, Prod) can use **Bicep with Azure DevOps** to **automate and standardize** infrastructure deployments while ensuring **security and compliance**.

Project 5: Implementing Azure DevOps Pipeline for Automated Security Scanning using SonarQube and WhiteSource

---

1. Project Scope

This project focuses on integrating **automated security scanning** into an **Azure DevOps CI/CD pipeline** using **SonarQube** for code quality analysis and **WhiteSource (Mend)** for open-source dependency security checks.

Key Features:

✅ **Identify vulnerabilities early** in the development lifecycle
✅ **Enforce code quality** using **SonarQube**
✅ **Scan for security vulnerabilities** in open-source dependencies using **WhiteSource**
✅ **Prevent insecure code from being deployed**
✅ **Generate reports for auditing and compliance**

---

2. Tools Used

- **Azure DevOps Pipelines** – Automates the security scanning process
- **SonarQube** – Performs static code analysis
- **WhiteSource (Mend)** – Scans for open-source security vulnerabilities
- **Azure Key Vault** – Stores sensitive API keys
- **Azure Artifacts** – Securely stores scanned and approved builds
- **OWASP Dependency-Check** – Identifies known vulnerabilities in dependencies

---

3. Analysis Approach

Challenges Without Security Scanning

❌ **Vulnerabilities** may exist in deployed applications
❌ No **automated enforcement** of security best practices
❌ **Insecure dependencies** could introduce risks
❌ Lack of **visibility** into application security

Proposed Solution

✅ **Automate security scanning** for every code commit
✅ **Block builds if security issues** are found
✅ **Monitor security posture** using SonarQube dashboards
✅ **Ensure compliance with security policies**

---

4. Step-by-Step Implementation

Step 1: Set Up SonarQube in Azure DevOps

1. **Install SonarQube Extension:**
    - Go to **Azure DevOps** → **Extensions Marketplace**
    - Search for **SonarQube** and install it
2. **Create a SonarQube Server in Azure:**
    - Deploy SonarQube using **Azure Container Instances:**

```
az container create --resource-group SecurityRG --name SonarQubeServer \

--image sonarqube --cpu 2 --memory 4 \

--ports 9000 --restart-policy Always
```

3. **Generate a SonarQube Token:**
    - Login to `http://<SonarQubeServer-IP>:9000`
    - Navigate to **Administration** → **Security** → **Generate Token**
4. **Store SonarQube Token in Azure Key Vault:**

```
az keyvault secret set --vault-name SecurityVault --name SonarQubeToken --value "<generated-token>"
```

Step 2: Add SonarQube Scanning to Azure DevOps Pipeline

1. **Go to Azure DevOps** → **Pipelines** → **New Pipeline**
2. **Modify the CI/CD pipeline (`sonar-pipeline.yml`):**

```
trigger:

  branches:
```

```
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

variables:
  sonarToken: $(SONARQUBE_TOKEN)

steps:
- task: SonarQubePrepare@4
  inputs:
    SonarQube: 'MySonarQubeService'
    scannerMode: 'CLI'
    configMode: 'manual'
    cliProjectKey: 'my-security-project'
    cliProjectName: 'SecureApp'
    cliSources: 'src'
    cliArguments: '-Dsonar.token=$(sonarToken)'

- task: SonarQubeAnalyze@4
  displayName: 'Run SonarQube Analysis'
```

```
- task: SonarQubePublish@4

  displayName: 'Publish SonarQube Results'

  inputs:

    pollingTimeoutSec: '300'
```

Step 3: Add Open-Source Vulnerability Scanning with WhiteSource

1. **Install the WhiteSource Bolt Extension in Azure DevOps**
2. **Modify pipeline to include WhiteSource security scan (`whitesource-pipeline.yml`):**

Unset
```
trigger:

  branches:

    include:

      - main


pool:

  vmImage: 'ubuntu-latest'


steps:

- task: WhiteSource@21

  displayName: 'Run WhiteSource Scan'

  inputs:

    cwd: '$(Build.SourcesDirectory)'

    productName: 'SecureApp'
```

```
    configFile: 'whitesource.config.json'
```

3. **Configure WhiteSource Policy** to block builds for high-severity vulnerabilities.

Step 4: Implement Dependency Scanning with OWASP Dependency-Check

1. **Add OWASP Dependency-Check to the pipeline:**

```
Unset
- task: Bash@3
  displayName: 'Run OWASP Dependency Check'
  inputs:
    targetType: 'inline'
    script: |
      curl -sSL https://get.owasp.org/dependency-check-cli/ -o dependency-check.sh
      chmod +x dependency-check.sh
      ./dependency-check.sh --project SecureApp --scan $(Build.SourcesDirectory)
```

Step 5: Block Deployment if Security Issues Are Found

1. **Add a Quality Gate Check in Azure DevOps:**
   - Go to **Azure DevOps** → **Pipelines** → **Releases** → **Pre-deployment Conditions**
   - Enable **Gates** → **Query SonarQube Quality Gate**
   - Set condition to **Fail deployment if security score < 80%**

Step 6: Automate Security Reports & Alerts

1. **Send alerts if vulnerabilities are found:**

Unset

```
az monitor metrics alert create --name "HighSeverityAlert"
--resource-group SecurityRG \

--condition "severity > 3" --action-group "SecOpsTeam"
```

---

5. Conclusion

🚀 **Key Benefits of Security Scanning in Azure DevOps:**
✅ **Automates vulnerability detection** in code and dependencies
✅ **Prevents insecure code from being deployed**
✅ **Enforces security policies using WhiteSource & OWASP checks**
✅ **Improves compliance for organizations following security standards**

---

6. Real-Time Example

A **banking institution** building an online payment system can integrate **SonarQube and WhiteSource in Azure DevOps** to **detect vulnerabilities early** and **ensure compliance with PCI-DSS standards**.

Project 6: Implementing Azure DevOps CI/CD Pipeline for Serverless Applications Using Azure Functions

---

1. Project Scope

This project focuses on **automating the deployment of serverless applications** using **Azure DevOps Pipelines** and **Azure Functions.** Serverless computing enables **event-driven, scalable applications** without managing infrastructure.

Key Features:

✅ **Automate CI/CD for Azure Functions**
✅ **Use Azure DevOps Pipelines for Continuous Deployment**
✅ **Manage Function App configurations using Azure App Configuration**
✅ **Enable monitoring and logging with Azure Application Insights**
✅ **Secure environment variables using Azure Key Vault**

---

2. Tools Used

- **Azure DevOps Pipelines** - Automates deployment
- **Azure Functions** - Serverless compute platform
- **Azure Storage Account** - Event trigger storage
- **Azure Key Vault** - Stores API keys and secrets
- **Azure Application Insights** - Logs and monitors function executions
- **PowerShell & YAML** - Automates infrastructure provisioning

---

3. Analysis Approach

Challenges Without CI/CD for Azure Functions

❌ **Manual deployments** introduce inconsistencies
❌ **Difficult to track code changes** for functions
❌ **No version control** for function configurations
❌ **Security risks** due to hardcoded secrets

Proposed Solution

✅ Automate **Azure Functions deployment** using **Azure DevOps Pipelines**
✅ **Centralize configuration management** using **Azure App Configuration**
✅ Secure API keys and database credentials using **Azure Key Vault**
✅ Enable **monitoring & logging** with **Azure Application Insights**

---

4. Step-by-Step Implementation

Step 1: Create an Azure Function App

1. **Provision an Azure Function App using CLI:**

Unset
```
az group create --name FunctionAppRG --location eastus

az storage account create --name functionstorage --resource-group
FunctionAppRG --location eastus --sku Standard_LRS

az functionapp create --name MyFunctionApp --storage-account
functionstorage --resource-group FunctionAppRG
--consumption-plan-location eastus --runtime python
```

Step 2: Store Configuration in Azure Key Vault

1. **Create an Azure Key Vault:**

Unset
```
az keyvault create --name FunctionAppKeyVault --resource-group
FunctionAppRG --location eastus
```

2. **Store a connection string in the Key Vault:**

Unset
```
az keyvault secret set --vault-name FunctionAppKeyVault --name
"DBConnectionString" --value
"Server=myserver.database.windows.net;Database=mydb;"
```

Step 3: Develop and Push an Azure Function to Azure Repos

1. **Clone the Azure Repos Git Repository:**

Unset
```
git clone https://dev.azure.com/MyOrg/FunctionAppRepo.git

cd FunctionAppRepo
```

2. **Create a sample Python function (`HttpTrigger/__init__.py`):**

Python
```python
import logging

import azure.functions as func



def main(req: func.HttpRequest) -> func.HttpResponse:

    logging.info("Azure Function triggered successfully.")

    return func.HttpResponse("Hello, Azure Serverless World!",
status_code=200)
```

3. **Commit and push the code:**

Unset
```
git add .

git commit -m "Added Azure Function"

git push origin main
```

Step 4: Configure CI Pipeline for Function App Deployment

1. **Navigate to Azure DevOps → Pipelines → New Pipeline**
2. **Create the following YAML file (`azure-functions-ci.yml`):**

Unset
```
trigger:
```

```yaml
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: UsePythonVersion@0
  inputs:
    versionSpec: '3.x'
    addToPath: true

- script: |
    python -m venv env
    source env/bin/activate
    pip install -r requirements.txt
  displayName: 'Install Dependencies'

- task: ArchiveFiles@2
  inputs:
    rootFolderOrFile: '$(Build.SourcesDirectory)'
    includeRootFolder: false
```

```
    archiveType: 'zip'

    archiveFile: '$(Build.ArtifactStagingDirectory)/function.zip'


- task: PublishBuildArtifacts@1

  inputs:

    pathToPublish: '$(Build.ArtifactStagingDirectory)/function.zip'

    artifactName: 'drop'
```

Step 5: Configure CD Pipeline for Function Deployment

1. Go to Azure DevOps → Releases → New Release Pipeline
2. Select "Azure Function App Deployment"
3. Modify the CD pipeline (`azure-functions-cd.yml`):

```
Unset
trigger:

  branches:

    include:

      - main


pool:

  vmImage: 'ubuntu-latest'


steps:

- task: DownloadBuildArtifacts@0
```

```
  inputs:
    artifactName: 'drop'
    downloadPath: '$(System.DefaultWorkingDirectory)'

- task: AzureFunctionApp@1
  inputs:
    azureSubscription: 'MyAzureSubscription'
    appType: 'functionApp'
    appName: 'MyFunctionApp'
    package: '$(System.DefaultWorkingDirectory)/drop/function.zip'
```

Step 6: Secure API Keys and Configurations

1. **Modify function to fetch secrets from Azure Key Vault (`function.json`):**

Unset
```
{
  "bindings": [
    {
      "type": "httpTrigger",
      "direction": "in",
      "authLevel": "function",
      "methods": ["get"]
    }
```

```
  ],
  "scriptFile": "__init__.py",
  "env": {
    "DB_CONNECTION_STRING":
"@Microsoft.KeyVault(SecretUri=https://FunctionAppKeyVault.vault.azure.
net/secrets/DBConnectionString/)"
  }
}
```

Step 7: Enable Monitoring with Azure Application Insights

1. **Enable Application Insights for Function App:**

Unset
```
az functionapp update --name MyFunctionApp --resource-group
FunctionAppRG --set appInsightsEnabled=true
```

2. **Modify pipeline to enable logging:**

Unset
```
- task: AzureCLI@2
  inputs:
    azureSubscription: 'MyAzureSubscription'
    scriptType: 'bash'
    scriptLocation: 'inlineScript'
    inlineScript: |
      az monitor app-insights component create --app
MyFunctionAppInsights --location eastus --resource-group FunctionAppRG
```

Step 8: Validate and Deploy

1. **Run the Azure DevOps pipeline to deploy the function**
2. **Test the function using cURL:**

Unset
```
curl
https://myfunctionapp.azurewebsites.net/api/HttpTrigger?code=<function-key>
```

3. **Verify logs in Application Insights**

---

5. Conclusion

🚀 **Key Benefits of CI/CD for Serverless Apps in Azure DevOps:**
✅ **Automates Azure Function deployment** for rapid releases
✅ **Secures function configurations** using **Azure Key Vault**
✅ **Improves monitoring** with **Application Insights**
✅ **Reduces costs** by utilizing **serverless architecture**

---

6. Real-Time Example

A **logistics company** can use **Azure Functions with DevOps Pipelines** to automate order processing **without provisioning servers**, ensuring **cost efficiency and scalability**.

Project 7: Implementing Blue-Green Deployment for an Azure Web App Using Azure DevOps

---

1. Project Scope

This project focuses on implementing a **Blue-Green Deployment strategy** using **Azure DevOps Pipelines** and **Azure App Service Deployment Slots**. The **Blue-Green** model reduces downtime and risk by maintaining **two identical environments**, where one serves **live traffic (Blue)** while the other acts as **staging (Green)** for testing.

Key Features:

✅ **Zero-downtime deployments** for web applications
✅ **Instant rollback to the previous version** if issues occur
✅ **Use Azure DevOps Pipelines to automate deployments**
✅ **Route production traffic safely using Azure Traffic Manager**
✅ **Secure app configurations using Azure Key Vault**

---

2. Tools Used

- **Azure DevOps Pipelines** – Automates deployment
- **Azure App Service** – Hosts the web application
- **Azure Traffic Manager** – Routes traffic between Blue and Green environments
- **Azure App Service Deployment Slots** – Enables swapping environments
- **Azure Key Vault** – Stores application secrets
- **PowerShell & YAML** – Automates infrastructure provisioning

---

3. Analysis Approach

Challenges Without Blue-Green Deployment

❌ **Deployment downtime** when updating applications
❌ **Difficult rollback** process if a deployment fails
❌ **Inconsistent testing environments**
❌ **Customer impact during production updates**

Proposed Solution

✅ **Deploy new versions in the Green slot** without affecting users
✅ **Run tests and validations on the Green slot** before switching traffic
✅ **Instant rollback by switching back to the Blue slot** if issues arise
✅ **Use Azure Traffic Manager** to manage **gradual rollout strategies**

4. Step-by-Step Implementation

Step 1: Create an Azure Web App with Deployment Slots

1. **Provision an Azure Web App:**

Unset
```
az group create --name BlueGreenRG --location eastus

az appservice plan create --name BlueGreenPlan --resource-group
BlueGreenRG --sku S1 --is-linux

az webapp create --resource-group BlueGreenRG --plan BlueGreenPlan
--name blue-webapp --runtime "DOTNETCORE:6.0"
```

2. **Create a Deployment Slot (Green):**

Unset
```
az webapp deployment slot create --name blue-webapp --resource-group
BlueGreenRG --slot green
```

Step 2: Store Configuration in Azure Key Vault

1. **Create an Azure Key Vault:**

Unset
```
az keyvault create --name BlueGreenKeyVault --resource-group
BlueGreenRG --location eastus
```

2. **Store an API key in Key Vault:**

Unset
```
az keyvault secret set --vault-name BlueGreenKeyVault --name "APIKey"
--value "SecureAPIKey123"
```

Step 3: Set Up Azure DevOps Repository and CI Pipeline

1. **Clone the Azure Repos Git Repository:**

```
Unset
git clone https://dev.azure.com/MyOrg/BlueGreenDeployment.git

cd BlueGreenDeployment
```

2. **Create a simple .NET Web Application (`Program.cs`):**

```
Unset
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () => "Hello from Blue-Green Deployment!");

app.Run();
```

3. **Modify the CI pipeline (`ci-pipeline.yml`):**

```
Unset
trigger:

  branches:

    include:

      - main



pool:

  vmImage: 'ubuntu-latest'



steps:

- task: UseDotNet@2
```

```
  inputs:
    packageType: 'sdk'
    version: '6.x'

- script: dotnet restore
  displayName: 'Restore dependencies'

- script: dotnet build --configuration Release
  displayName: 'Build Web App'

- task: ArchiveFiles@2
  inputs:
    rootFolderOrFile: '$(Build.SourcesDirectory)'
    includeRootFolder: false
    archiveType: 'zip'
    archiveFile: '$(Build.ArtifactStagingDirectory)/webapp.zip'

- task: PublishBuildArtifacts@1
  inputs:
    pathToPublish: '$(Build.ArtifactStagingDirectory)/webapp.zip'
    artifactName: 'drop'
```

Step 4: Set Up CD Pipeline for Blue-Green Deployment

1. **Modify the CD pipeline (`cd-pipeline.yml`):**

Unset
```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

stages:
- stage: DeployToGreen
  jobs:
  - job: Deploy
    steps:
    - task: DownloadBuildArtifacts@0
      inputs:
        artifactName: 'drop'
        downloadPath: '$(System.DefaultWorkingDirectory)'

    - task: AzureWebApp@1
      inputs:
        azureSubscription: 'MyAzureSubscription'
        appType: 'webApp'
```

```
        appName: 'blue-webapp'
        slotName: 'green'
        package: '$(System.DefaultWorkingDirectory)/drop/webapp.zip'

- stage: TestGreenSlot
  jobs:
  - job: Validate
    steps:
    - script: curl -f https://blue-webapp-green.azurewebsites.net ||
exit 1
      displayName: 'Health Check for Green Slot'

- stage: SwapSlots
  dependsOn: TestGreenSlot
  jobs:
  - job: Swap
    steps:
    - task: AzureCLI@2
      inputs:
        azureSubscription: 'MyAzureSubscription'
        scriptType: 'bash'
        scriptLocation: 'inlineScript'
        inlineScript: |
```

```
         az webapp deployment slot swap --name blue-webapp
--resource-group BlueGreenRG --slot green
```

Step 5: Implement Rollback Strategy

1. **Rollback to previous version in case of failure:**

```
Unset
az webapp deployment slot swap --name blue-webapp --resource-group
BlueGreenRG --slot green
```

2. **Configure automatic rollback in Azure DevOps:**
   - Navigate to **Pipelines** → **Releases** → **Pre-deployment Conditions**
   - Enable **Rollback on failure**

Step 6: Gradual Traffic Switching Using Azure Traffic Manager

1. **Create an Azure Traffic Manager Profile:**

```
Unset
az network traffic-manager profile create --name BlueGreenTM
--resource-group BlueGreenRG --routing-method Weighted --dns-name
bluegreentm
```

2. **Add endpoints for Blue and Green Slots:**

```
Unset
az network traffic-manager endpoint create --resource-group BlueGreenRG
--profile-name BlueGreenTM --name blue-endpoint --type azureEndpoints
--target-resource-id
/subscriptions/<sub_id>/resourceGroups/BlueGreenRG/providers/Microsoft.
Web/sites/blue-webapp --weight 100
```

```
az network traffic-manager endpoint create --resource-group BlueGreenRG
--profile-name BlueGreenTM --name green-endpoint --type azureEndpoints
--target-resource-id
/subscriptions/<sub_id>/resourceGroups/BlueGreenRG/providers/Microsoft.
Web/sites/blue-webapp/slots/green --weight 0
```

3. **Gradually switch traffic to Green:**

```
Unset
az network traffic-manager endpoint update --resource-group BlueGreenRG
--profile-name BlueGreenTM --name green-endpoint --weight 50
```

---

5. Conclusion

🚀 **Key Benefits of Blue-Green Deployment in Azure DevOps:**
✅ **Ensures zero-downtime deployment** with deployment slots
✅ **Enables instant rollback** if the new version has issues
✅ **Minimizes risk** by testing before going live
✅ **Allows gradual rollout** using Traffic Manager

---

6. Real-Time Example

An **e-commerce company** launching new website features can use **Blue-Green Deployment** to **test updates safely** and **switch traffic seamlessly without downtime**.

Project 8: Implementing Azure DevOps CI/CD Pipeline for Containerized Applications with Azure Kubernetes Service (AKS) and Helm

---

1. Project Scope

This project focuses on **automating the deployment of containerized applications** using **Azure DevOps Pipelines**, **Azure Kubernetes Service (AKS)**, and **Helm**. The goal is to ensure **scalability, high availability, and automated updates** for containerized workloads.

Key Features:

✅ **Automate CI/CD for Kubernetes deployments**
✅ **Use Helm for managing Kubernetes manifests**
✅ **Secure Kubernetes secrets using Azure Key Vault**
✅ **Enable rolling updates for zero downtime**
✅ **Monitor workloads using Prometheus & Grafana**

---

2. Tools Used

- **Azure DevOps Pipelines** – Automates build and deployment
- **Azure Kubernetes Service (AKS)** – Hosts containerized workloads
- **Docker & Azure Container Registry (ACR)** – Stores container images
- **Helm** – Manages Kubernetes deployments
- **Azure Key Vault** – Secures Kubernetes secrets
- **Prometheus & Grafana** – Monitors AKS workloads

---

3. Analysis Approach

Challenges Without Automated Kubernetes Deployments

❌ **Manual deployments** lead to inconsistencies
❌ **Lack of version control** for Kubernetes configurations
❌ **Security risks** due to exposed secrets in YAML files
❌ **Difficult rollback strategy** if an issue arises

Proposed Solution

✅ **Use Azure DevOps CI/CD Pipelines** to automate deployments
✅ **Leverage Helm for versioned Kubernetes deployments**
✅ **Secure Kubernetes secrets using Azure Key Vault**
✅ **Enable auto-scaling for high availability**

4. Step-by-Step Implementation

Step 1: Create an AKS Cluster and Configure Kubernetes Context

1. **Provision an Azure Kubernetes Service (AKS) Cluster:**

```
Unset
az group create --name AKSRG --location eastus

az aks create --resource-group AKSRG --name MyAKSCluster --node-count 3
--generate-ssh-keys
```

2. **Connect to the AKS Cluster:**

```
Unset
az aks get-credentials --resource-group AKSRG --name MyAKSCluster

kubectl get nodes
```

Step 2: Store Kubernetes Secrets in Azure Key Vault

1. **Create an Azure Key Vault:**

```
Unset
az keyvault create --name AKSKeyVault --resource-group AKSRG --location
eastus
```

2. **Store a database connection string:**

```
Unset
az keyvault secret set --vault-name AKSKeyVault --name
"DBConnectionString" --value "Server=mydbserver;Database=mydb;"
```

Step 3: Develop and Push a Sample Containerized Application

1. **Clone the Azure DevOps Repository:**

```
git clone https://dev.azure.com/MyOrg/KubernetesApp.git

cd KubernetesApp
```

2. **Create a Dockerfile (`Dockerfile`):**

```
FROM node:14

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

CMD ["node", "server.js"]

EXPOSE 3000
```

3. **Commit and push the code:**

```
git add .

git commit -m "Added Dockerfile"

git push origin main
```

Step 4: Create an Azure DevOps CI Pipeline for Building and Pushing Docker Images

1. **Modify the CI pipeline (`ci-pipeline.yml`):**

Unset

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: Docker@2
  inputs:
    command: 'buildAndPush'
    repository: 'myacr.azurecr.io/kubernetesapp'
    dockerfile: 'Dockerfile'
    containerRegistry: 'MyAzureContainerRegistry'
    tags: '$(Build.BuildId)'

- task: PublishBuildArtifacts@1
  inputs:
    pathToPublish: '$(Build.ArtifactStagingDirectory)'
    artifactName: 'drop'
```

Step 5: Configure Helm for Kubernetes Deployment

1. **Create a Helm Chart (`helm/kubernetesapp/Chart.yaml`):**

```
Unset
apiVersion: v2

name: kubernetesapp

description: A Helm chart for deploying a Node.js application

version: 1.0.0

appVersion: "1.0"
```

2. **Define Kubernetes Deployment and Service (`helm/kubernetesapp/templates/deployment.yaml`):**

```
Unset
apiVersion: apps/v1

kind: Deployment

metadata:

  name: kubernetesapp

spec:

  replicas: 2

  selector:

    matchLabels:

      app: kubernetesapp

  template:

    metadata:

      labels:

        app: kubernetesapp

    spec:
```

```
      containers:
      - name: kubernetesapp
        image: myacr.azurecr.io/kubernetesapp:latest
        ports:
        - containerPort: 3000
```

3. **Define a Service for Load Balancing (`helm/kubernetesapp/templates/service.yaml`):**

```
Unset
apiVersion: v1
kind: Service
metadata:
  name: kubernetesapp
spec:
  type: LoadBalancer
  selector:
    app: kubernetesapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

Step 6: Create an Azure DevOps CD Pipeline for AKS Deployment Using Helm

1. **Modify the CD pipeline (`cd-pipeline.yml`):**

Unset
```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

stages:
- stage: DeployToAKS
  jobs:
  - job: Deploy
    steps:
    - task: HelmDeploy@0
      inputs:
        connectionType: 'Azure Resource Manager'
        azureSubscription: 'MyAzureSubscription'
        azureResourceGroup: 'AKSRG'
        kubernetesCluster: 'MyAKSCluster'
        namespace: 'default'
        command: 'upgrade'
        chartType: 'FilePath'
        chartPath: 'helm/kubernetesapp'
```

```
      releaseName: 'kubernetesapp'
```

Step 7: Enable Auto-Scaling for AKS Deployment

1. **Enable Horizontal Pod Autoscaler (HPA):**

Unset
```
kubectl autoscale deployment kubernetesapp --cpu-percent=50 --min=2
--max=5
```

2. **Monitor autoscaling events:**

Unset
```
kubectl get hpa
```

Step 8: Enable Monitoring Using Prometheus and Grafana

1. **Deploy Prometheus in AKS:**

Unset
```
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts

helm install prometheus prometheus-community/kube-prometheus-stack
```

2. **Deploy Grafana for Visualization:**

Unset
```
helm install grafana stable/grafana

kubectl port-forward svc/grafana 3000:80
```

3. **Access Grafana dashboard at** `http://localhost:3000/`

5. Conclusion

🚀 **Key Benefits of CI/CD for AKS with Helm:**
✅ **Automates Kubernetes deployments with Azure DevOps**
✅ **Ensures high availability using auto-scaling**
✅ **Secures sensitive configurations using Azure Key Vault**
✅ **Enables monitoring with Prometheus and Grafana**

6. Real-Time Example

A **fintech company** deploying microservices-based banking applications can use **Azure DevOps, AKS, and Helm** to **ensure high availability, automated scaling, and security** for its workloads.

Project 9: Implementing Azure DevOps Pipeline for Infrastructure as Code (IaC) Using Terraform on Azure

---

1. Project Scope

This project focuses on **using Terraform** to define, provision, and manage **Azure infrastructure** through **Azure DevOps Pipelines.** Terraform enables Infrastructure as Code (IaC), making it easy to automate cloud resource provisioning while ensuring consistency and scalability.

Key Features:

✅ **Automate Infrastructure Deployment using Terraform**
✅ **Manage Infrastructure as Code (IaC) in Azure DevOps**
✅ **Use Remote State for Team Collaboration**
✅ **Ensure Consistent Deployments Across Environments (Dev, QA, Prod)**
✅ **Implement Role-Based Access Control (RBAC) Policies**

---

2. Tools Used

- **Azure DevOps Pipelines** – Automates Terraform execution
- **Terraform** – Manages Azure Infrastructure as Code
- **Azure Storage Account** – Stores Terraform state files
- **Azure Key Vault** – Secures Terraform secrets
- **PowerShell & YAML** – Used for automation scripting

---

3. Analysis Approach

Challenges Without Infrastructure as Code (IaC)

❌ **Manual provisioning** is error-prone and time-consuming
❌ **Difficult to track changes** and maintain consistency
❌ **Security risks** due to hardcoded secrets
❌ **No version control** for infrastructure configurations

Proposed Solution

✅ Use **Terraform for Infrastructure as Code (IaC)**
✅ Automate **Azure resource provisioning** with **Azure DevOps Pipelines**
✅ Store **Terraform state remotely** using **Azure Storage Account**
✅ Secure credentials using **Azure Key Vault**

---

4. Step-by-Step Implementation

Step 1: Set Up an Azure Storage Account for Terraform State

1. **Create a Resource Group:**

Unset
```
az group create --name TerraformRG --location eastus
```

2. **Create an Azure Storage Account:**

Unset
```
az storage account create --name terraformstate1234 --resource-group
TerraformRG --location eastus --sku Standard_LRS
```

3. **Create a Storage Container for Terraform State:**

Unset
```
az storage container create --name tfstate --account-name
terraformstate1234
```

Step 2: Store Terraform Secrets in Azure Key Vault

1. **Create an Azure Key Vault:**

Unset
```
az keyvault create --name TerraformKeyVault --resource-group
TerraformRG --location eastus
```

2. **Store Service Principal Credentials in Key Vault:**

Unset
```
az keyvault secret set --vault-name TerraformKeyVault --name "ClientID"
--value "your-client-id"
```

```
az keyvault secret set --vault-name TerraformKeyVault --name
"ClientSecret" --value "your-client-secret"
```

Step 3: Create a Terraform Configuration File (`main.tf`)

1. **Clone the Azure DevOps Repository:**

```
git clone https://dev.azure.com/MyOrg/TerraformDeployment.git

cd TerraformDeployment
```

2. **Define Azure Infrastructure in `main.tf`:**

```
terraform {

  backend "azurerm" {

    resource_group_name  = "TerraformRG"

    storage_account_name = "terraformstate1234"

    container_name       = "tfstate"

    key                  = "terraform.tfstate"

  }

}


provider "azurerm" {

  features {}

}
```

```
resource "azurerm_resource_group" "example" {
  name     = "MyTerraformRG"
  location = "East US"
}
```

3. **Commit and Push Code:**

```
Unset
git add .
git commit -m "Added Terraform configuration"
git push origin main
```

Step 4: Configure Azure DevOps CI Pipeline for Terraform Plan

1. **Modify the CI Pipeline (`terraform-ci.yml`):**

```
Unset
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'
```

```yaml
steps:
- task: TerraformInstaller@1
  inputs:
    terraformVersion: 'latest'

- task: TerraformTaskV2@2
  displayName: 'Initialize Terraform'
  inputs:
    provider: 'azurerm'
    command: 'init'
    backendServiceArm: 'MyAzureSubscription'
    backendAzureRmResourceGroupName: 'TerraformRG'
    backendAzureRmStorageAccountName: 'terraformstate1234'
    backendAzureRmContainerName: 'tfstate'
    backendAzureRmKey: 'terraform.tfstate'

- task: TerraformTaskV2@2
  displayName: 'Run Terraform Plan'
  inputs:
    provider: 'azurerm'
    command: 'plan'
    environmentServiceNameAzureRM: 'MyAzureSubscription'
```

Step 5: Configure Azure DevOps CD Pipeline for Terraform Apply

1. **Modify the CD Pipeline (`terraform-cd.yml`):**

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

stages:
- stage: Deploy
  jobs:
  - job: ApplyTerraform
    steps:
    - task: TerraformTaskV2@2
      displayName: 'Apply Terraform Configuration'
      inputs:
        provider: 'azurerm'
        command: 'apply'
        environmentServiceNameAzureRM: 'MyAzureSubscription'
        commandOptions: '-auto-approve'
```

Step 6: Implement Role-Based Access Control (RBAC) Policies

1. **Assign Contributor Role to Terraform Service Principal:**

```
Unset
az role assignment create --assignee "<service-principal-id>" --role
Contributor --scope
"/subscriptions/<subscription-id>/resourceGroups/TerraformRG"
```

Step 7: Enable Automated Rollback for Terraform Deployments

1. **Track Terraform Deployment History:**

```
Unset
terraform show
```

2. **Rollback to Previous State if Deployment Fails:**

```
Unset
terraform apply -refresh=false terraform.tfstate.backup
```

---

5. Conclusion

🚀 **Key Benefits of Terraform for Azure Infrastructure:**

✅ Automates Infrastructure Deployment using Terraform & Azure DevOps
✅ Ensures Consistency with Remote State Storage
✅ Enhances Security by Storing Secrets in Azure Key Vault
✅ Enables Rollback to Previous Deployments

---

6. Real-Time Example

A **large enterprise** managing multiple cloud environments (Dev, QA, Prod) can use **Terraform in Azure DevOps Pipelines** to automate **infrastructure provisioning** while ensuring **consistency and security** across teams.

Project 10: Implementing Azure DevOps CI/CD Pipeline for Azure API Management (APIM) with Automated API Deployment

---

1. Project Scope

This project focuses on **automating API deployment and management** using **Azure DevOps Pipelines** and **Azure API Management (APIM)**. The goal is to ensure seamless API versioning, security enforcement, and automated updates across different environments.

Key Features:

✅ **Automate API deployment and versioning using Azure DevOps**
✅ **Enforce security policies (Rate Limiting, JWT, OAuth, API Keys)**
✅ **Use Azure API Management (APIM) for API gateway capabilities**
✅ **Enable logging and monitoring for API traffic**
✅ **Integrate automated API testing in CI/CD pipeline**

---

2. Tools Used

- **Azure DevOps Pipelines** – Automates API deployment
- **Azure API Management (APIM)** – API gateway for managing APIs
- **Swagger / OpenAPI** – Defines API specifications
- **Azure Key Vault** – Stores API secrets and keys
- **Azure Monitor & App Insights** – API logging and monitoring
- **PowerShell & YAML** – Automation scripting

---

3. Analysis Approach

Challenges Without API Management & Automation

❌ **Difficult to manage multiple API versions**
❌ **No centralized security enforcement (Rate Limits, Authentication, IP Restrictions)**
❌ **Manual API deployments cause inconsistencies**
❌ **Lack of monitoring and logging for API performance**

Proposed Solution

✅ Automate **API deployment, versioning, and security enforcement** using **Azure DevOps & APIM**
✅ **Centralize API authentication** using **OAuth, JWT, and API Keys**
✅ **Ensure API high availability and load balancing**
✅ **Enable monitoring & logging with Azure Monitor and App Insights**

---

4. Step-by-Step Implementation

Step 1: Provision an Azure API Management Instance

1. **Create a Resource Group:**

```
Unset
az group create --name APIMRG --location eastus
```

2. **Create an API Management Instance:**

```
Unset
az apim create --name MyAPIM --resource-group APIMRG --publisher-name
"MyCompany" --publisher-email "admin@mycompany.com" --sku-name
Consumption
```

Step 2: Define API Specifications Using OpenAPI (Swagger)

1. **Create an OpenAPI Specification File (`swagger.json`):**

```
Unset
{
  "swagger": "2.0",
  "info": {
    "title": "My API",
    "description": "API for managing users",
    "version": "1.0.0"
  },
  "paths": {
    "/users": {
      "get": {
```

```
          "summary": "Get all users",
          "responses": {
            "200": {
              "description": "OK"
            }
          }
        }
      }
    }
  }
```

2. **Commit and Push the API Spec to Azure Repos:**

```
Unset
git add .
git commit -m "Added OpenAPI specification"
git push origin main
```

Step 3: Configure Azure DevOps CI Pipeline for API Validation

1. **Modify the CI pipeline (`api-ci.yml`):**

```
Unset
trigger:
  branches:
    include:
```

```
    - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: OpenApi@1
  inputs:
    openApiFile: 'swagger.json'
    validationRules: 'All'

- task: PublishBuildArtifacts@1
  inputs:
    pathToPublish: 'swagger.json'
    artifactName: 'drop'
```

Step 4: Configure Azure DevOps CD Pipeline for API Deployment to APIM

1. **Modify the CD pipeline (`api-cd.yml`):**

Unset
```
trigger:
  branches:
    include:
```

```
      - main


pool:
  vmImage: 'ubuntu-latest'


stages:
- stage: DeployToAPIM
  jobs:
  - job: Deploy
    steps:
    - task: DownloadBuildArtifacts@0
      inputs:
        artifactName: 'drop'
        downloadPath: '$(System.DefaultWorkingDirectory)'


    - task: AzureCLI@2
      inputs:
        azureSubscription: 'MyAzureSubscription'
        scriptType: 'bash'
        scriptLocation: 'inlineScript'
        inlineScript: |
          az apim api import --resource-group APIMRG --service-name
MyAPIM --path /users --api-id MyAPI --specification-format OpenAPI
```

```
--specification-path
$(System.DefaultWorkingDirectory)/drop/swagger.json
```

Step 5: Implement API Security Policies

1. **Apply JWT Authentication Policy in APIM (`jwt-policy.xml`):**

```
Unset
<inbound>

  <validate-jwt header-name="Authorization"
failed-validation-httpcode="401">

    <openid-config
url="https://login.microsoftonline.com/{tenant-id}/v2.0/.well-known/ope
nid-configuration" />

    <audiences>

      <audience>api://my-api</audience>

    </audiences>

  </validate-jwt>

</inbound>
```

2. **Apply Rate Limiting Policy in APIM (`rate-limit-policy.xml`):**

```
Unset
<inbound>

  <rate-limit calls="10" renewal-period="60" />

</inbound>
```

3. **Apply these policies via Azure CLI:**

Unset
```
az apim api policy create --resource-group APIMRG --service-name MyAPIM
--api-id MyAPI --xml-policy-file jwt-policy.xml
```

Step 6: Enable API Logging & Monitoring with Azure Monitor

1. **Enable App Insights for API Management:**

Unset
```
az monitor app-insights component create --app MyAPIMInsights
--location eastus --resource-group APIMRG
```

2. **Enable Request Logging in APIM:**

Unset
```
<inbound>

  <base />

  <log-to-eventhub logger-id="apim-logger">

    <message>Request received</message>

  </log-to-eventhub>

</inbound>
```

3. **Deploy the Logging Policy in APIM:**

Unset
```
az apim api policy create --resource-group APIMRG --service-name MyAPIM
--api-id MyAPI --xml-policy-file logging-policy.xml
```

Step 7: Automate API Testing Using Postman & Newman

1. **Run API Tests in Azure DevOps CI/CD Pipeline:**

Unset
```
- task: NodeTool@0

  inputs:

    versionSpec: '16.x'


- script: |

    npm install -g newman

    newman run api-tests.postman_collection.json --reporters cli,junit

  displayName: 'Run API Tests with Newman'
```

---

5. Conclusion

🚀 Key Benefits of Azure API Management with DevOps:
✅ Automates API deployments and updates
✅ Enforces security policies (OAuth, JWT, API Keys, Rate Limiting)
✅ Enables centralized logging and monitoring for APIs
✅ Integrates automated API testing into DevOps workflows

---

6. Real-Time Example

A **banking company** can use **Azure API Management (APIM) and Azure DevOps Pipelines** to **securely expose APIs**, enforce **rate limits**, and enable **seamless API versioning** while ensuring **compliance with security policies**.

🎉 Congratulations! All 10 Azure DevOps Real-Time Projects Are Completed! 🎉

→ TRAININGS

WE ARE DIFFERENT

At CareerByteCode, we redefine training by focusing on real-world, hands-on experience. Unlike traditional learning methods, we provide step-by-step implementation guides, 500+ real-time use cases, and industry-relevant projects across cutting-edge technologies like AWS, Azure, GCP, DevOps, AI, FullStack Development and more.

Our approach goes beyond theoretical knowledge—we offer expert mentorship, helping learners understand how to study effectively, close career gaps, and gain the practical skills that employers value.

16+
Years of operations

91+
Countries worldwide

241 K Happy clients

Our Usecases Platform
https://careerbytecode.substack.com

Our WebShop
https://careerbytecode.shop

CAREER BYTE CODE
REALTIME PROJECTS PLATFORM

91 COUNTRIES | subscriber 241k Learners

+32 471 40 89 08

CAREERBYTECODE.SUBSTACK.COM

CareerByteCode
Learning Made simple

CareerByteCode
All in One Platform

STAY IN TOUCH WITH US!

Website

Our WebShop https://careerbytecode.shop
Our Usecases Platform https://careerbytecode.substack.com

E-mail
careerbytec@gmail.com

HQ address
Belgium, Europe

Social Media
@careerbytecode

Phone
+32 471 40 8908

For any RealTime Handson Projects

And for more tips like this

Follow

Like & ReShare

@careerbytecode