DEVOPS

REALTIME PROJECTS

careerbytecode.substack.com
+32 - 471408908

500+ RealTime Projects platform

Project 1: CI/CD Pipeline for a .NET Core Application Using Azure DevOps

1. Project Scope

This project focuses on setting up a **Continuous Integration and Continuous Deployment (CI/CD) pipeline** for a .NET Core web application using Azure DevOps. The pipeline automates code integration, testing, and deployment to **Azure App Service**. The goal is to ensure **fast, reliable, and automated software delivery** with minimal manual intervention.

Key Features:

- Automated build and testing of the application
- Deployment to **Azure App Service**
- Versioning control and rollback mechanism
- Integration with Azure DevOps Repos and GitHub

---

2. Tools Used

- **Azure DevOps** – For managing repositories, pipelines, and releases
- **Azure Repos or GitHub** – Source code version control
- **Azure Pipelines** – CI/CD automation
- **Azure App Service** – Hosting the application
- **MSTest / NUnit** – For unit testing
- **SonarCloud** – Code quality and security analysis
- **Azure Key Vault** – Secure management of secrets
- **Application Insights** – Performance monitoring

---

3. Analysis Approach

Current Challenges Without CI/CD

- Manual deployments are time-consuming and error-prone
- Lack of automated testing can lead to undetected issues in production
- Security vulnerabilities due to hardcoded credentials

Proposed Solution

Implement a fully automated CI/CD pipeline to handle **build, testing, security scanning, and deployment** seamlessly.

Pipeline Flow:

1. Developer commits code to **Azure Repos/GitHub**.
2. Azure Pipelines **triggers build** and runs tests.
3. SonarCloud performs **static code analysis**.
4. Security scanning using **Azure Key Vault** to fetch secrets.
5. If all tests pass, the build artifact is stored in **Azure Artifacts**.
6. Azure Pipelines **deploys the application** to Azure App Service.
7. Application Insights provides **monitoring and logs**.

---

4. Step-by-Step Implementation

Step 1: Set Up Azure DevOps Repository

1. Go to Azure DevOps and create a new project.
2. Navigate to **Repos** → **Import Code** or clone a new repository.

Use the following Git commands to push an existing .NET Core app:

```
git init
git remote add origin <repo-url>
git add .
git commit -m "Initial commit"
git push origin main
```

3.

Step 2: Create a CI Pipeline in Azure DevOps

1. Go to **Pipelines** → **New Pipeline** → Choose **Azure Repos Git** or **GitHub**.

Select **Starter Pipeline** and replace it with this YAML:

```yaml
trigger:
  branches:
    include:
      - main

pool:
```

```yaml
  vmImage: 'ubuntu-latest'

steps:
- task: UseDotNet@2
  inputs:
    packageType: 'sdk'
    version: '6.x'

- script: dotnet build --configuration Release
  displayName: 'Build the project'

- script: dotnet test --logger trx
  displayName: 'Run Unit Tests'

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

2. Save and **run the pipeline** to build and test the application.

Step 3: Integrate SonarCloud for Code Quality

1. Go to SonarCloud and create an account.
2. Connect it to Azure DevOps and set up a new project.

Modify the pipeline to include:

```yaml
- task: SonarCloudPrepare@1
  inputs:
    SonarCloud: 'SonarCloud'
    organization: '<your-org>'
    projectKey: '<your-project-key>'
    scannerMode: 'MSBuild'
- script: dotnet build
- task: SonarCloudAnalyze@1
- task: SonarCloudPublish@1
```

Step 4: Set Up CD Pipeline to Azure App Service

1. Go to **Releases** → **New Release Pipeline**.
2. Choose **Azure App Service Deployment** template.
3. Select **Azure Subscription** and the App Service name.
4. Choose the **drop** artifact from CI pipeline.
5. Configure deployment stages like **Staging** and **Production**.
6. Enable **Approval Gates** for manual approvals before production deployment.
7. Click **Create Release** and deploy.

Step 5: Secure Secrets Using Azure Key Vault

1. In Azure Portal, go to **Key Vault** → **Create a new vault**.
2. Add a secret (e.g., `DatabaseConnectionString`).

In Azure Pipelines, add the **Azure Key Vault task**:

```
- task: AzureKeyVault@1
  inputs:
    azureSubscription: 'MyAzureSubscription'
    KeyVaultName: 'my-keyvault'
    SecretsFilter: '*'
```

Step 6: Monitor Using Application Insights

1. In Azure Portal, navigate to **Application Insights** and link it to the app.

Modify `appsettings.json`:

```
"ApplicationInsights": {
    "InstrumentationKey": "<your-instrumentation-key>"
}
```

2. Update the Azure DevOps release pipeline to enable **monitoring logs**.

5. Conclusion

This project successfully implemented an **end-to-end CI/CD pipeline** for a .NET Core web application using Azure DevOps. Key benefits:

✅ **Automated Build & Testing** – Ensuring high code quality
✅ **Faster Deployment** – Reducing manual intervention
✅ **Secure Configuration Management** – Using **Azure Key Vault**
✅ **Monitoring & Logging** – With **Application Insights**

Real-Time Example:

A software company developing a **finance dashboard** can use this pipeline to deploy new features faster while ensuring security and stability.

Project 2: Infrastructure as Code (IaC) with Terraform and Azure DevOps

1. Project Scope

This project focuses on **automating infrastructure provisioning** using **Terraform** with Azure DevOps. The goal is to manage and deploy infrastructure resources, such as **Azure Virtual Machines (VMs), Storage Accounts, Networking, and Databases**, using Infrastructure as Code (IaC) principles.

Key Features:

- Fully automated **resource provisioning** using Terraform
- Integration with **Azure DevOps Pipelines** for continuous deployment
- State management with **Azure Storage**
- **Role-based access control (RBAC)** for secure deployments
- Infrastructure **drift detection and remediation**

---

2. Tools Used

- **Azure DevOps** – CI/CD pipeline for infrastructure deployment
- **Terraform** – Infrastructure as Code tool
- **Azure CLI** – To manage Azure resources
- **Azure Key Vault** – Secure storage for sensitive credentials
- **Azure Storage Account** – To store Terraform state files
- **GitHub/Azure Repos** – Source code management for Terraform scripts

---

3. Analysis Approach

Challenges Without IaC

- **Manual provisioning** of infrastructure is time-consuming and error-prone
- **Inconsistent environments** between development, testing, and production
- **Difficult rollback process** if an infrastructure change breaks the system

Proposed Solution

By using **Terraform with Azure DevOps**, we can:
✅ Automate infrastructure deployment
✅ Maintain **version-controlled** infrastructure code
✅ Improve **reliability and repeatability**
✅ Easily scale and modify infrastructure as per business needs

Workflow:

1. **Developers** write Terraform scripts and push to **Azure Repos/GitHub**.
2. **Azure DevOps Pipelines** trigger Terraform execution.

3. Terraform **plans and applies** infrastructure changes.
4. Resources are deployed to **Azure**.
5. Terraform **state is stored securely** in an Azure Storage Account.

---

4. Step-by-Step Implementation

Step 1: Set Up Azure DevOps Repository

1. Go to **Azure DevOps** → Create a new project.
2. Navigate to **Repos** → Clone a new Git repository.

Push the Terraform code:

```
git init
git remote add origin <repo-url>
git add .
git commit -m "Initial Terraform commit"
git push origin main
```

Step 2: Create Terraform Configuration for Azure Infrastructure

Inside the repository, create a file `main.tf`:

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

resource "azurerm_resource_group" "rg" {
  name     = "my-resource-group"
  location = "East US"
}

resource "azurerm_storage_account" "storage" {
  name                     = "mystorageacct"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
```

```
  account_replication_type = "LRS"
}
```

Initialize Terraform:

```
terraform init
```

Check the execution plan:

```
terraform plan
```

Apply changes to create resources:

```
terraform apply -auto-approve
```

Step 3: Configure Remote State Storage in Azure

Create an **Azure Storage Account** to store Terraform state:

```
az storage account create --name mystoragestate --resource-group my-resource-group
--location eastus --sku Standard_LRS
```

Update `backend.tf`:

```
terraform {
  backend "azurerm" {
    resource_group_name   = "my-resource-group"
    storage_account_name  = "mystoragestate"
```

```
    container_name        = "tfstate"
    key                   = "terraform.tfstate"
  }
}
```

Run:

```
terraform init
```

Step 4: Create a CI/CD Pipeline in Azure DevOps

1. In **Azure DevOps**, go to **Pipelines** → **New Pipeline**.
2. Select **Azure Repos Git** or **GitHub**.

Choose **Starter Pipeline** and modify the YAML as follows:

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: TerraformInstaller@0
  inputs:
    terraformVersion: 'latest'

- script: terraform init
  displayName: 'Initialize Terraform'

- script: terraform plan -out=tfplan
  displayName: 'Terraform Plan'

- script: terraform apply -auto-approve tfplan
```

```
  displayName: 'Apply Terraform Changes'
```

3. Save and **run the pipeline**.

Step 5: Secure Credentials with Azure Key Vault

1. In Azure Portal, navigate to **Key Vault** and create a vault.
2. Store **Terraform service principal credentials** as secrets.

Add the **Azure Key Vault task** to the pipeline:

```
- task: AzureKeyVault@1
  inputs:
    azureSubscription: 'MyAzureSubscription'
    KeyVaultName: 'my-keyvault'
    SecretsFilter: '*'
```

Step 6: Deploy Changes Automatically

Push a **new Terraform resource** (e.g., Virtual Machine) in `main.tf`:

```
resource "azurerm_virtual_machine" "vm" {
  name                = "myVM"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  size               = "Standard_DS1_v2"
  admin_username     = "azureuser"

  os_profile {
    computer_name  = "myVM"
    admin_password = "SecureP@ssw0rd!"
  }

  network_interface_ids = [azurerm_network_interface.nic.id]
```

```
}
```

1. **Commit and push the changes** to trigger the pipeline and deploy the VM.

---

5. Conclusion

This project implemented **Infrastructure as Code (IaC) using Terraform and Azure DevOps**, enabling **automated, scalable, and version-controlled infrastructure provisioning.**

Key Benefits:

✅ **Automated Infrastructure Deployment** – No manual configuration
✅ **State Management** – Azure Storage ensures team collaboration
✅ **Security Best Practices** – Using **Azure Key Vault**
✅ **Seamless CI/CD Integration** – Infrastructure is provisioned on every commit

Real-Time Example:

A company migrating to **Azure Cloud** can use this approach to **automate provisioning of Virtual Machines, Storage, and Networking**, ensuring consistency across all environments.

Project 3: Automating Application Deployment with Azure DevOps CI/CD and Kubernetes (AKS)

1. Project Scope

This project focuses on **automating the deployment of a containerized application** on **Azure Kubernetes Service (AKS) using Azure DevOps CI/CD pipelines**.

Key Features:

✅ Build and package the application using **Docker**
✅ Push the container image to **Azure Container Registry (ACR)**
✅ Deploy the application to **AKS using Helm charts**
✅ Use **Azure DevOps Pipelines** for CI/CD automation
✅ Implement **rolling updates** and rollback strategies

---

2. Tools Used

- **Azure DevOps** - Continuous Integration & Continuous Deployment
- **Azure Kubernetes Service (AKS)** - Kubernetes cluster for container orchestration
- **Docker** - Containerization of applications
- **Helm** - Kubernetes package manager for deployment
- **Azure Container Registry (ACR)** - To store Docker images
- **Kubernetes Manifest (YAML)** - For AKS deployment
- **Azure CLI & Kubectl** - For managing the AKS cluster

---

3. Analysis Approach

Challenges Without CI/CD & Kubernetes

- **Manual deployments** lead to inconsistency and errors
- **Difficult rollback process** in case of failures
- **Scaling applications manually** is inefficient

Proposed Solution

By using **Azure DevOps CI/CD with AKS**, we can:
✅ **Automate application builds and deployments**
✅ **Ensure version-controlled releases**

✅ Enable rolling updates and rollback
✅ Improve application scalability and reliability

Workflow:

1. **Developers push code** to Azure Repos (or GitHub).
2. **Azure DevOps triggers a build pipeline**, creating a Docker image.
3. **The image is pushed** to Azure Container Registry (ACR).
4. **A release pipeline** deploys the application to AKS using Helm.

---

4. Step-by-Step Implementation

Step 1: Set Up Azure Kubernetes Service (AKS)

Create a **Resource Group** in Azure:

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

Create an **AKS Cluster**:

```
az aks create --resource-group MyResourceGroup --name MyAKSCluster --node-count 2
--enable-addons monitoring --generate-ssh-keys
```

Connect to the cluster:

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

Step 2: Set Up Azure Container Registry (ACR)

Create an **ACR instance**:

```
az acr create --resource-group MyResourceGroup --name MyACR --sku Basic
```

Attach ACR to AKS:

```
az aks update --resource-group MyResourceGroup --name MyAKSCluster --attach-acr
MyACR
```

Step 3: Build and Push a Docker Image to ACR

Clone a sample app:

```
git clone https://github.com/Azure-Samples/azure-voting-app-redis.git
cd azure-voting-app-redis
```

Build and tag the Docker image:

```
docker build -t myacr.azurecr.io/azure-vote:v1 .
```

Push the image to ACR:

```
docker login myacr.azurecr.io
docker push myacr.azurecr.io/azure-vote:v1
```

Step 4: Create a CI/CD Pipeline in Azure DevOps

1. Create a CI Pipeline for Building & Pushing the Docker Image

1. In **Azure DevOps**, go to **Pipelines** → **New Pipeline**
2. Select **GitHub or Azure Repos** as the source

Choose **Starter Pipeline** and modify the YAML:

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: Docker@2
  inputs:
    command: 'buildAndPush'
    repository: 'myacr.azurecr.io/azure-vote'
    dockerfile: 'Dockerfile'
    containerRegistry: 'MyACR'
    tags: 'v$(Build.BuildId)'
```

3. Save and run the pipeline. This builds and pushes the container image to **ACR**.

---

2. Create a Kubernetes Deployment YAML File

Create a `deployment.yaml` file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azure-vote
spec:
  replicas: 2
  selector:
```

```
    matchLabels:
      app: azure-vote
  template:
    metadata:
      labels:
        app: azure-vote
    spec:
      containers:
      - name: azure-vote
        image: myacr.azurecr.io/azure-vote:v1
        ports:
        - containerPort: 80
```

---

3. Create a Release Pipeline for Deployment to AKS

1. In **Azure DevOps**, go to **Releases** → **New Release Pipeline**.
2. Select **Azure Kubernetes Service Deployment**.
3. Set up stages:
   - **Build Stage**: Uses the CI pipeline to build and push the Docker image.
   - **Release Stage**: Deploys to **AKS** using the `kubectl` command.

Add a **task** to deploy the app:

```
- task: Kubernetes@1
  inputs:
    command: apply
    connectionType: 'Azure Resource Manager'
    azureSubscriptionEndpoint: 'MyAzureSubscription'
    resourceGroupName: 'MyResourceGroup'
    clusterName: 'MyAKSCluster'
    namespace: 'default'
    manifests: '$(System.DefaultWorkingDirectory)/deployment.yaml'
```

Step 5: Enable Rolling Updates & Rollback

Modify the **deployment.yaml** to use a new image tag:

```
image: myacr.azurecr.io/azure-vote:v2
```

Push the changes → The pipeline automatically **updates** the app on AKS.

If something breaks, rollback using:

```
kubectl rollout undo deployment azure-vote
```

---

5. Conclusion

This project automated the **end-to-end deployment of a containerized application** using **Azure DevOps and AKS**.

Key Benefits:

✅ **Seamless CI/CD integration** – No manual deployments
✅ **Automatic scaling & self-healing** – Handled by **Kubernetes**
✅ **Rolling updates & rollback support**
✅ **Secure and efficient container image management** using **ACR**

Real-Time Example:

A **DevOps team** managing a **microservices-based application** can use this setup to **automate deployments, improve reliability, and enable rapid scaling**.

Project 4: Infrastructure as Code (IaC) with Terraform and Azure DevOps

1. Project Scope

This project focuses on using **Terraform to provision and manage Azure infrastructure** through **Azure DevOps CI/CD pipelines.**

Key Features:

✅ Automate the creation of Azure infrastructure (VMs, Networks, Storage)
✅ Use Terraform for **Infrastructure as Code (IaC)**
✅ Store Terraform state files in **Azure Storage Account**
✅ Implement **Azure DevOps Pipelines** for **automated provisioning**
✅ Enable **approval-based deployment** for controlled infrastructure changes

---

2. Tools Used

- **Azure DevOps** – CI/CD for Infrastructure deployment
- **Terraform** – Infrastructure as Code tool
- **Azure CLI** – To manage Azure resources
- **Azure Storage Account** – For Terraform state management
- **Azure Key Vault** – To store sensitive credentials
- **Azure Virtual Network (VNet), Virtual Machines, Storage Accounts**

---

3. Analysis Approach

Challenges Without IaC & DevOps

- **Manual infrastructure provisioning** is error-prone and time-consuming
- **Difficult to track changes** made to cloud resources
- **No version control** for infrastructure changes
- **Security risks** due to hardcoded credentials

Proposed Solution

By integrating **Terraform with Azure DevOps**, we can:
✅ **Automate Azure resource provisioning**
✅ **Enable version control** for infrastructure
✅ **Improve security by managing credentials in Azure Key Vault**
✅ **Enforce approval-based deployment for governance**

Workflow:

1. **Terraform code** is stored in Azure Repos (or GitHub).
2. **Azure DevOps CI Pipeline** validates and formats Terraform code.
3. **Terraform applies infrastructure changes** in Azure.
4. **Terraform state file** is stored in an **Azure Storage Account**.

---

4. Step-by-Step Implementation

Step 1: Set Up Azure DevOps & Terraform Repository

1. **Create a new repository** in Azure DevOps (or GitHub)

Clone the repo locally:

```
git clone https://dev.azure.com/MyOrg/MyTerraformRepo.git
cd MyTerraformRepo
```

2. **Create Terraform Configuration Files:**
   - `main.tf` → Defines Azure infrastructure
   - `variables.tf` → Stores variables
   - `terraform.tfvars` → Defines actual values

Step 2: Configure Terraform Backend in Azure Storage

Create a Resource Group:

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

Create an Azure Storage Account:

```
az storage account create --name myterraformstate --resource-group MyResourceGroup
--location eastus --sku Standard_LRS
```

Create a Storage Container for Terraform state:

```
az storage container create --name terraform-state --account-name myterraformstate
```

Define Terraform backend in `main.tf`:

```
terraform {
  backend "azurerm" {
    resource_group_name   = "MyResourceGroup"
    storage_account_name  = "myterraformstate"
    container_name        = "terraform-state"
    key                   = "terraform.tfstate"
  }
}
```

Step 3: Define Infrastructure in Terraform

1. Define Azure Provider (`main.tf`)

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

2. Create an Azure Virtual Network

```
resource "azurerm_virtual_network" "my_vnet" {
  name                = "myVnet"
  location            = "East US"
  resource_group_name = "MyResourceGroup"
  address_space       = ["10.0.0.0/16"]
}
```

3. Create an Azure Virtual Machine

```
resource "azurerm_linux_virtual_machine" "my_vm" {
  name                = "MyVM"
  resource_group_name = "MyResourceGroup"
  location            = azurerm_virtual_network.my_vnet.location
  size                = "Standard_B1s"
  admin_username      = "adminuser"
  admin_password      = "P@ssw0rd123!"

  network_interface_ids = [azurerm_network_interface.my_nic.id]
}
```

Step 4: Create an Azure DevOps Pipeline for Terraform

1. Create a CI Pipeline to Validate Terraform Code

1. In **Azure DevOps**, go to **Pipelines** → **New Pipeline**
2. Select **GitHub or Azure Repos** as the source

Choose **Starter Pipeline** and modify the YAML:

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: TerraformInstaller@1
  inputs:
    terraformVersion: '1.2.0'

- script: |
    terraform init
    terraform validate
  displayName: 'Terraform Init & Validate'
```

3. Save and run the pipeline. This **validates Terraform code** before deployment.

Step 5: Create a CD Pipeline for Infrastructure Deployment

1. Add Terraform Plan & Apply Steps in Azure DevOps

Modify the **release pipeline YAML**:

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- script: |
    terraform init
    terraform plan -out=tfplan
  displayName: 'Terraform Plan'

- task: ManualValidation@0
  inputs:
    notifyUsers: 'user@example.com'
    instructions: 'Approve Terraform Apply'
  displayName: 'Manual Approval for Apply'

- script: |
    terraform apply -auto-approve tfplan
  displayName: 'Terraform Apply'
```

Step 6: Secure Terraform Secrets with Azure Key Vault

Create an Azure Key Vault:

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

Store Terraform Admin Password:

```
az keyvault secret set --vault-name MyKeyVault --name TerraformAdminPassword
--value "P@ssw0rd123!"
```

Modify Terraform Code to Retrieve Secrets:

```
data "azurerm_key_vault_secret" "admin_password" {
  name         = "TerraformAdminPassword"
  key_vault_id = azurerm_key_vault.my_keyvault.id
}
```

---

5. Conclusion

This project **automates infrastructure provisioning using Terraform and Azure DevOps**.

Key Benefits:

✅ Eliminates manual provisioning
✅ Ensures version control for infrastructure changes
✅ Improves security by storing secrets in Azure Key Vault
✅ Enables approval-based deployments

Real-Time Example:

A **cloud operations team** managing a **large-scale Azure environment** can use this setup to **automate deployments, enforce governance, and improve security**.

Project 5: CI/CD Pipeline for a .NET Core Application Using Azure DevOps

1. Project Scope

In this project, we will build a **Continuous Integration and Continuous Deployment (CI/CD) pipeline** in **Azure DevOps** for a **.NET Core web application**. The goal is to automate the **build, testing, and deployment** of the application to **Azure App Service**.

Key Features:

✅ Automate **code build and testing** using Azure DevOps pipelines
✅ Deploy the application to **Azure App Service**
✅ Implement **branch policies** to ensure code quality
✅ Enable **rollbacks** in case of failures
✅ Use **Secrets Management** with Azure Key Vault

---

2. Tools Used

- **Azure DevOps** - CI/CD automation
- **Azure App Service** - Hosting the .NET Core application
- **GitHub/Azure Repos** - Source code repository
- **Azure Key Vault** - Secrets management
- **Azure CLI** - To manage Azure resources
- **YAML Pipelines** - For defining the CI/CD workflow

---

3. Analysis Approach

Challenges Without CI/CD

- **Manual deployment takes time** and is error-prone
- **Inconsistent environments** between development, testing, and production
- **No automated testing**, leading to deployment failures
- **Security risks** due to hardcoded credentials

Proposed Solution

By integrating **Azure DevOps with .NET Core and Azure App Service**, we can:
✅ **Automate build and deployment** of the application
✅ **Enforce code quality checks** using branch policies

✅ **Secure application secrets** in Azure Key Vault
✅ **Enable rollback options** in case of failed deployments

Workflow:

1. **Developers push code** to Azure Repos (or GitHub).
2. **CI Pipeline builds and tests** the application automatically.
3. **CD Pipeline deploys** the application to Azure App Service.
4. **Azure Key Vault stores** sensitive configuration data.
5. **Rollback mechanism** enables safe deployment.

---

4. Step-by-Step Implementation

Step 1: Set Up Azure DevOps & Repository

1. **Create a new repository** in Azure DevOps (or GitHub).

Clone the repository and create a sample .NET Core web app:

```
git clone https://dev.azure.com/MyOrg/MyDotNetRepo.git
cd MyDotNetRepo
dotnet new webapp -o MyWebApp
```

Step 2: Create an Azure App Service

Create a Resource Group:

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

Create an Azure App Service Plan:

```
az appservice plan create --name MyAppServicePlan --resource-group MyResourceGroup
--sku B1 --is-linux
```

Create the Azure Web App:

```
az webapp create --resource-group MyResourceGroup --plan MyAppServicePlan --name
MyWebApp --runtime "DOTNET:6.0"
```

Step 3: Create CI Pipeline in Azure DevOps

1. **Go to Azure DevOps → Pipelines → New Pipeline**
2. **Select Azure Repos Git or GitHub** as the source
3. **Choose Starter Pipeline** and modify the YAML:

YAML Configuration for CI Pipeline

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: UseDotNet@2
  inputs:
    packageType: 'sdk'
    version: '6.x'
    installationPath: $(Agent.ToolsDirectory)/dotnet

- script: |
    dotnet restore
    dotnet build --configuration Release
  displayName: 'Build .NET Core Application'

- script: |
    dotnet test --configuration Release
  displayName: 'Run Unit Tests'

- task: PublishBuildArtifacts@1
  inputs:
```

```
    pathToPublish: '$(Build.ArtifactStagingDirectory)'
    artifactName: 'drop'
  displayName: 'Publish Build Artifacts'
```

4. **Save and run the pipeline.** This will build and test the .NET Core app automatically.

Step 4: Create CD Pipeline to Deploy to Azure App Service

1. Modify the Deployment YAML

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: DownloadBuildArtifacts@0
  inputs:
    buildType: 'current'
    artifactName: 'drop'
    targetPath: '$(Build.ArtifactStagingDirectory)'

- task: AzureWebApp@1
  inputs:
    azureSubscription: 'MyAzureServiceConnection'
    appName: 'MyWebApp'
    package: '$(Build.ArtifactStagingDirectory)/**/*.zip'
    deploymentMethod: 'zipDeploy'
  displayName: 'Deploy to Azure App Service'
```

Step 5: Secure Credentials Using Azure Key Vault

Create an Azure Key Vault:

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

Store Database Connection String:

```
az keyvault secret set --vault-name MyKeyVault --name "DBConnectionString" --value "Server=myserver.database.windows.net;Database=mydb;User Id=myuser;Password=mypassword;"
```

Modify Azure DevOps Pipeline to Fetch Secrets:

```
- task: AzureKeyVault@1
  inputs:
    azureSubscription: 'MyAzureServiceConnection'
    keyVaultName: 'MyKeyVault'
    secretsFilter: '*'
    runAsPreJob: true
```

Step 6: Implement Rollback Strategy

Enable Deployment Slots in Azure App Service

```
az webapp deployment slot create --resource-group MyResourceGroup --name MyWebApp --slot staging
```

Modify Deployment Pipeline to Use Staging Slot

```
- task: AzureWebApp@1
  inputs:
    azureSubscription: 'MyAzureServiceConnection'
    appName: 'MyWebApp'
    slotName: 'staging'
    package: '$(Build.ArtifactStagingDirectory)/**/*.zip'
    deploymentMethod: 'zipDeploy'
```

Perform a Swap to Production on Successful Deployment

```
az webapp deployment slot swap --resource-group MyResourceGroup --name MyWebApp
--slot staging
```

---

5. Conclusion

This project **automates the CI/CD process for a .NET Core application** using Azure DevOps.

Key Benefits:

✅ **Automated Build & Testing** ensures code quality
✅ **Seamless Deployment** to Azure App Service
✅ **Secrets Management** with Azure Key Vault
✅ **Rollback Support** using deployment slots

Real-Time Example:

A **software development team** working on a **.NET Core web application** can use this pipeline to **automate deployments, reduce manual effort, and improve software quality.**

Project 6: Implementing Infrastructure as Code (IaC) Using Terraform and Azure DevOps

1. Project Scope

In this project, we will automate the provisioning of **Azure cloud infrastructure** using **Terraform** within an Azure DevOps pipeline. The goal is to create a **fully automated Infrastructure as Code (IaC) solution** that deploys a **virtual network, subnets, a virtual machine, and storage accounts** on Azure.

Key Features:

✅ Automate infrastructure provisioning using Terraform
✅ Store Terraform state files securely in Azure Storage
✅ Implement **CI/CD pipeline** for Terraform deployment
✅ Use Azure DevOps **Service Connections** for authentication
✅ Enforce **Infrastructure as Code (IaC)** best practices

---

2. Tools Used

- **Azure DevOps** – CI/CD automation
- **Terraform** – Infrastructure as Code (IaC)
- **Azure Storage Account** – Terraform state management
- **Azure Virtual Network (VNet)** – Network provisioning
- **Azure Virtual Machine (VM)** – Compute resource
- **Azure CLI** – Command-line automation

---

3. Analysis Approach

Challenges Without Infrastructure as Code

- **Manual infrastructure setup** leads to inconsistencies
- **Difficult to replicate environments** (dev, staging, prod)
- **Security risks** due to improper state management
- **No version control** for infrastructure changes

Proposed Solution

By using **Terraform with Azure DevOps**, we can:

✅ **Automate Infrastructure Deployment** via pipelines
✅ **Manage Terraform state securely** in Azure Storage
✅ **Enable version control** and rollback for infrastructure changes
✅ **Ensure consistency** across multiple environments

Workflow:

1. **Terraform scripts define infrastructure** as code
2. **Azure DevOps pipeline applies Terraform** to Azure
3. **Terraform state is stored** in an Azure Storage Account
4. **Changes are validated** before deployment
5. **Infrastructure is deployed/updated** automatically

---

4. Step-by-Step Implementation

Step 1: Set Up Azure DevOps & Repository

1. **Create a new repository** in Azure DevOps (or GitHub).

Clone the repository and create a Terraform configuration folder:

```
git clone https://dev.azure.com/MyOrg/TerraformAzureRepo.git
cd TerraformAzureRepo
mkdir terraform
```

Step 2: Install Terraform & Configure Backend Storage

Install Terraform on your local machine (if not installed):

```
sudo apt-get update && sudo apt-get install -y terraform
```

Create an Azure Storage Account for Terraform state management:

```
az group create --name TerraformStateRG --location eastus
az storage account create --name tfstateaccount --resource-group TerraformStateRG
--sku Standard_LRS
az storage container create --name terraform-state --account-name tfstateaccount
```

Update Terraform backend configuration (`terraform/backend.tf`):

```
terraform {
  backend "azurerm" {
    resource_group_name  = "TerraformStateRG"
    storage_account_name = "tfstateaccount"
    container_name       = "terraform-state"
    key                  = "terraform.tfstate"
  }
}
```

Step 3: Write Terraform Configuration for Infrastructure

Create a new Terraform configuration file (`terraform/main.tf`) to define infrastructure:

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

resource "azurerm_resource_group" "rg" {
  name     = "MyTerraformRG"
  location = "East US"
}

resource "azurerm_virtual_network" "vnet" {
  name                = "MyVNet"
  location            = azurerm_resource_group.rg.location
```

```
  resource_group_name = azurerm_resource_group.rg.name
  address_space       = ["10.0.0.0/16"]
}

resource "azurerm_subnet" "subnet1" {
  name                 = "MySubnet1"
  resource_group_name  = azurerm_resource_group.rg.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = ["10.0.1.0/24"]
}

resource "azurerm_virtual_machine" "vm" {
  name                  = "MyTerraformVM"
  location              = azurerm_resource_group.rg.location
  resource_group_name   = azurerm_resource_group.rg.name
  network_interface_ids = [azurerm_network_interface.nic.id]
  vm_size               = "Standard_B2s"

  storage_os_disk {
    name              = "myosdisk"
    caching           = "ReadWrite"
    create_option     = "FromImage"
  }

  os_profile {
    computer_name  = "MyVM"
    admin_username = "azureuser"
    admin_password = "MySecurePassword123!"
  }

  os_profile_linux_config {
    disable_password_authentication = false
  }

  source_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "18.04-LTS"
    version   = "latest"
  }
}
```

Step 4: Create Terraform CI/CD Pipeline in Azure DevOps

1. **Go to Azure DevOps → Pipelines → New Pipeline**
2. **Select Azure Repos Git or GitHub** as the source
3. **Choose Starter Pipeline** and modify the YAML:

YAML Configuration for Terraform Pipeline

```yaml
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: TerraformInstaller@0
  inputs:
    terraformVersion: '1.0.0'

- script: |
    terraform init
  displayName: 'Initialize Terraform'

- script: |
    terraform plan -out=tfplan
  displayName: 'Plan Terraform Changes'

- script: |
    terraform apply -auto-approve tfplan
  displayName: 'Apply Terraform Configuration'
```

Step 5: Secure Credentials Using Azure DevOps Service Connection

1. **Go to Azure DevOps → Project Settings → Service connections**
2. **Create a new service connection** for Azure Resource Manager
3. **Grant permissions** to the pipeline to use this connection

Modify the pipeline YAML to use the service connection:

```
- task: TerraformCLI@0
  inputs:
    command: 'init'
    backendType: 'azurerm'
    backendAzureRmSubscriptionId: '$(AZURE_SUBSCRIPTION_ID)'
```

Step 6: Validate & Deploy Infrastructure

Commit changes to the repository:

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

1. **Run the Azure DevOps Pipeline**
2. **Verify Infrastructure in Azure Portal**

---

5. Conclusion

This project **automates infrastructure provisioning using Terraform** with Azure DevOps.

Key Benefits:

✅ Automated Infrastructure Deployment
✅ Secure Terraform State Management
✅ Version-Controlled Infrastructure
✅ Consistent & Repeatable Environments

Real-Time Example:

A **DevOps team** responsible for **managing Azure infrastructure** can use this approach to **provision, update, and scale** cloud resources efficiently.

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** for an **Azure Web App** using **Azure DevOps Pipelines**. The goal is to reduce downtime and minimize risk when deploying new application versions by maintaining two identical environments (Blue and Green).

Key Features:

✅ Automate Blue-Green deployment using Azure DevOps
✅ Deploy and test new versions before switching traffic
✅ Use **Azure App Service Slots** for zero-downtime deployment
✅ Implement rollback strategies in case of failure

---

2. Tools Used

- **Azure DevOps Pipelines** – Automate CI/CD
- **Azure App Service** – Web application hosting
- **App Service Deployment Slots** – Blue-Green switch
- **Azure Monitor** – Track application performance
- **Azure Traffic Manager** – Route traffic between slots
- **GitHub or Azure Repos** – Store application source code

---

3. Analysis Approach

Challenges With Traditional Deployments

- **Application downtime** during deployment
- **High risk of breaking production** with new updates
- **Rollback process is slow and manual**
- **Testing in production impacts users**

Proposed Solution: Blue-Green Deployment

With Blue-Green deployment, we:
✅ Maintain **two identical environments** (Blue & Green)
✅ Deploy and test the new version **in the Green slot**
✅ **Switch traffic** to the Green slot only after successful validation
✅ Keep Blue slot as a backup for **instant rollback** if needed

Workflow:

1. **Blue Slot (Current Production) runs the existing app**
2. **Green Slot (Staging) is used for new deployments**
3. **Traffic is routed to Green slot after testing**
4. **If issues arise, rollback instantly by swapping slots**

---

4. Step-by-Step Implementation

Step 1: Create an Azure Web App with Deployment Slots

1. **Go to Azure Portal → App Services → Create a Web App**
   - Name: `myapp-bluegreen`
   - Runtime: `Node.js / .NET / Python / Java` (as needed)
   - Plan: `Standard or Premium` (supports deployment slots)
2. **Create a deployment slot:**
   - In the Web App, go to **Deployment Slots**
   - Click **+ Add Slot**
   - Name the slot `green`
   - Clone settings from the Blue slot

Step 2: Configure CI/CD in Azure DevOps

1. **Go to Azure DevOps → Pipelines → Create New Pipeline**
2. **Select Repository** (Azure Repos or GitHub)
3. **Choose Starter Pipeline and Update the YAML**

CI/CD YAML for Blue-Green Deployment

```
trigger:
  branches:
    include:
      - main

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

```
steps:
- task: UseDotNet@2
  inputs:
    packageType: 'sdk'
    version: '6.x'

- task: DotNetCoreCLI@2
  inputs:
    command: 'restore'
    projects: '**/*.csproj'

- task: DotNetCoreCLI@2
  inputs:
    command: 'build'
    projects: '**/*.csproj'

- task: DotNetCoreCLI@2
  inputs:
    command: 'publish'
    publishWebProjects: true
    arguments: '--configuration Release --output
$(Build.ArtifactStagingDirectory)'
    zipAfterPublish: true

- task: AzureWebApp@1
  inputs:
    azureSubscription: 'MyAzureServiceConnection'
    appType: 'webApp'
    appName: 'myapp-bluegreen'
    deployToSlotOrASE: true
    resourceGroupName: 'MyResourceGroup'
    slotName: 'green'
    package: '$(Build.ArtifactStagingDirectory)/*.zip'
  displayName: 'Deploy to Green Slot'

- script: echo "Swapping Blue and Green slots"
  displayName: 'Swap Slots'
- task: AzureCLI@2
  inputs:
    azureSubscription: 'MyAzureServiceConnection'
    scriptType: 'bash'
    scriptLocation: 'inlineScript'
```

```
    inlineScript: |
      az webapp deployment slot swap --resource-group MyResourceGroup --name
myapp-bluegreen --slot green
  displayName: 'Swap Slots Blue <-> Green'
```

Step 3: Secure Azure DevOps Pipeline With Service Connection

1. **Go to Azure DevOps → Project Settings → Service connections**
2. **Create a new Azure Resource Manager service connection**
3. **Authorize the pipeline to use this connection**

Modify the YAML to reference the service connection securely:

```
azureSubscription: '$(AZURE_SERVICE_CONNECTION)'
```

Step 4: Validate Deployment & Swap Slots

Commit and push changes to GitHub or Azure Repos:

```
git add .
git commit -m "Added Blue-Green deployment pipeline"
git push origin main
```

1. **Azure DevOps Pipeline triggers the deployment** to the **Green slot**
2. **Manually verify the Green slot** before switching traffic
3. **Run slot swap command** to make Green the new production

---

Step 5: Implement Rollback Strategy

If something goes wrong after deployment:

Instant rollback by swapping slots again:

```
az webapp deployment slot swap --resource-group MyResourceGroup --name
```

```
myapp-bluegreen --slot green
```

Monitor logs and Azure Application Insights for debugging

---

5. Conclusion

Key Benefits of Blue-Green Deployment:

✅ **Zero-Downtime Deployment** – Users experience no service interruptions
✅ **Instant Rollback** – Revert to the previous version in seconds
✅ **Safe Testing in Staging** – Test before going live
✅ **Increased Deployment Frequency** – Deploy confidently

Real-World Use Case:

A **banking application** that requires **high availability** can use this **Blue-Green deployment strategy** to push updates **without affecting live users**.

Project 8: Implementing Canary Deployment for an Azure Kubernetes Service (AKS) Using Azure DevOps

1. Project Scope

This project focuses on **Canary Deployment** in **Azure Kubernetes Service (AKS)** using **Azure DevOps Pipelines.** The **Canary strategy** allows gradual release of new application versions to a small subset of users before rolling it out to everyone.

Key Features:

✅ Deploy a **new version to a small percentage** of traffic
✅ Monitor performance before increasing rollout
✅ Use **Azure DevOps Pipelines for automated CI/CD**
✅ **Rollback if issues occur** before full rollout

---

2. Tools Used

- **Azure Kubernetes Service (AKS)** – Hosts the containerized application
- **Azure DevOps Pipelines** – Automates CI/CD
- **Helm** – Manages Kubernetes deployments
- **Kustomize** – Manages environment-specific configurations
- **Prometheus + Grafana** – Monitors deployment performance
- **Nginx Ingress Controller** – Manages Canary traffic routing

---

3. Analysis Approach

Challenges With Traditional Deployments

- Deploying to **all users at once increases risk**
- Difficult to **test new features safely in production**
- **Rollback is complex** if issues occur after full deployment

Proposed Solution: Canary Deployment

With **Canary Deployment**, we:
✅ Deploy **new version to a small % of users** first
✅ Monitor performance with **Prometheus and Grafana**
✅ Gradually **increase rollout percentage** if no issues occur
✅ **Rollback immediately** if failures are detected

Workflow:

1. **Deploy the current stable version (V1) to 100% of users**
2. **Deploy new version (V2) to 10% of users** (Canary)
3. **Monitor logs, metrics, and user feedback**
4. **Gradually increase to 25%, 50%, then 100%**
5. **Rollback to V1 if failures occur**

---

4. Step-by-Step Implementation

Step 1: Set Up Azure Kubernetes Service (AKS)

Create an AKS Cluster:

```
az aks create --resource-group MyResourceGroup --name myAKSCluster --node-count 2
--enable-addons monitoring --generate-ssh-keys
```

Connect to AKS Cluster:

```
az aks get-credentials --resource-group MyResourceGroup --name myAKSCluster
```

Install Nginx Ingress Controller (for traffic routing):

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/prov
ider/cloud/deploy.yaml
```

Step 2: Configure CI/CD Pipeline in Azure DevOps

1. Go to Azure DevOps → Pipelines → Create New Pipeline
2. Select Repository (Azure Repos or GitHub)
3. Choose Starter Pipeline and Update YAML

CI/CD YAML for Canary Deployment

```yaml
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: Kubernetes@1
  displayName: 'Deploy Canary Version'
  inputs:
    connectionType: 'Azure Resource Manager'
    azureSubscription: 'MyAzureServiceConnection'
    azureResourceGroup: 'MyResourceGroup'
    kubernetesCluster: 'myAKSCluster'
    namespace: 'default'
    command: 'apply'
    useConfigurationFile: true
    configuration: |
      apiVersion: apps/v1
      kind: Deployment
      metadata:
        name: myapp
        labels:
          app: myapp
      spec:
        replicas: 3
        selector:
          matchLabels:
            app: myapp
        template:
          metadata:
            labels:
              app: myapp
```

```yaml
            version: canary
        spec:
          containers:
          - name: myapp
            image: myacr.azurecr.io/myapp:v2
            ports:
            - containerPort: 80
            resources:
              limits:
                cpu: "500m"
                memory: "256Mi"
              requests:
                cpu: "250m"
                memory: "128Mi"

- script: echo "Applying Traffic Routing Rules"
  displayName: 'Route 10% Traffic to Canary'
- task: Kubernetes@1
  inputs:
    connectionType: 'Azure Resource Manager'
    azureSubscription: 'MyAzureServiceConnection'
    kubernetesCluster: 'myAKSCluster'
    namespace: 'default'
    command: 'apply'
    useConfigurationFile: true
    configuration: |
      apiVersion: networking.k8s.io/v1
      kind: Ingress
      metadata:
        name: myapp-ingress
      spec:
        rules:
        - host: myapp.example.com
          http:
            paths:
            - path: /
              pathType: Prefix
              backend:
                service:
                  name: myapp
                  port:
                    number: 80
        canary:
```

```
        weight: 10
```

Step 3: Monitor Canary Performance

Deploy Prometheus for Monitoring

```
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
helm repo update
helm install prometheus prometheus-community/kube-prometheus-stack
```

Deploy Grafana for Dashboards

```
helm install grafana grafana/grafana
```

1. **Check Performance Metrics**
    - Open **Grafana** and connect it to **Prometheus**
    - Monitor CPU, memory, and response times
    - If issues occur, **rollback Canary Deployment**

Step 4: Gradual Rollout of Canary

Increase traffic to 25% if V2 is stable

```
kubectl patch ingress myapp-ingress --type='json' -p='[{"op": "replace", "path":
"/spec/canary/weight", "value": 25}]'
```

1. **Monitor Logs and Errors**
2. **Increase to 50%, then 100% if successful**

Step 5: Rollback If Issues Occur

Rollback to V1 immediately if users report issues:

```
kubectl rollout undo deployment myapp
```

1. **Monitor System Logs**
2. **Fix Issues & Restart Canary Process**

---

5. Conclusion

Key Benefits of Canary Deployment:

✅ **Reduces Risk** – Only a small % of users are affected if issues arise
✅ **Better User Experience** – Catch issues before full rollout
✅ **Quick Rollback** – Instantly revert to the previous version
✅ **Data-Driven Deployment** – Use real user feedback to validate releases

Real-World Use Case:

A **SaaS company** rolling out a new **payment feature** can use **Canary Deployment** to release it **gradually** and detect **bugs early** before full rollout.

Project 9: Implementing Infrastructure as Code (IaC) with Terraform in Azure DevOps

1. Project Scope

This project focuses on **Infrastructure as Code (IaC) using Terraform** in **Azure DevOps**. Terraform allows provisioning and managing Azure infrastructure in a **declarative and automated manner**, eliminating manual setup.

Key Features:

✅ Automate **Azure infrastructure provisioning**
✅ Use **Terraform as IaC** within Azure DevOps Pipelines
✅ **Apply changes consistently** across environments (Dev, QA, Prod)
✅ **Maintain version control** for infrastructure

---

2. Tools Used

- **Azure DevOps Pipelines** – Automates Terraform execution
- **Terraform** – Defines and deploys infrastructure
- **Azure Resource Manager (ARM)** – Manages Azure resources
- **Azure Key Vault** – Stores sensitive Terraform credentials
- **Azure Storage Account** – Stores Terraform state files

---

3. Analysis Approach

Challenges Without IaC:

❌ Manual deployment **is time-consuming and error-prone**
❌ Difficult to **maintain infrastructure consistency**
❌ No **version control** over infrastructure changes

Proposed Solution: Using Terraform With Azure DevOps

✅ Define **infrastructure as code** using Terraform
✅ Automate deployments via **Azure DevOps Pipelines**
✅ Use **Terraform state management** for tracking infrastructure
✅ Store **secrets securely in Azure Key Vault**

Workflow:

1. **Write Terraform scripts** to define Azure infrastructure
2. **Store Terraform files in an Azure DevOps repository**
3. **Use Azure DevOps Pipelines to execute Terraform commands**
4. **Manage infrastructure across multiple environments (Dev, QA, Prod)**
5. **Apply, update, and destroy infrastructure as needed**

---

4. Step-by-Step Implementation

Step 1: Install Terraform Locally

Download and install **Terraform**:

```
wget
https://releases.hashicorp.com/terraform/1.2.0/terraform_1.2.0_linux_amd64.zip
unzip terraform_1.2.0_linux_amd64.zip
sudo mv terraform /usr/local/bin/
terraform --version
```

Step 2: Create an Azure Storage Account for Terraform State Files

Terraform needs a **backend storage** to store state files.

Create a Resource Group:

```
az group create --name terraform-rg --location eastus
```

Create a Storage Account:

```
az storage account create --name terraformstorageacct --resource-group
```

```
terraform-rg --location eastus --sku Standard_LRS
```

Create a Storage Container:

```
az storage container create --name tfstate --account-name terraformstorageacct
```

Step 3: Define Terraform Configuration Files

Create the following Terraform files in an Azure DevOps Git repository:

1. **main.tf (Defines Azure Resources)**

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

resource "azurerm_resource_group" "rg" {
  name     = "my-devops-rg"
  location = "East US"
}

resource "azurerm_virtual_network" "vnet" {
  name                = "my-vnet"
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
  address_space       = ["10.0.0.0/16"]
}

resource "azurerm_subnet" "subnet" {
  name                 = "my-subnet"
  resource_group_name  = azurerm_resource_group.rg.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = ["10.0.1.0/24"]
}
```

2. variables.tf (Defines Variables)

```
variable "location" {
  default = "East US"
}

variable "resource_group_name" {
  default = "my-devops-rg"
}
```

3. backend.tf (Configures Terraform State)

```
terraform {
  backend "azurerm" {
    resource_group_name   = "terraform-rg"
    storage_account_name  = "terraformstorageacct"
    container_name        = "tfstate"
    key                   = "terraform.tfstate"
  }
}
```

Step 4: Create an Azure DevOps Pipeline for Terraform

1. Go to Azure DevOps → Pipelines → Create New Pipeline
2. Select Git Repository
3. Add Terraform Tasks to Pipeline YAML

Terraform CI/CD Pipeline in Azure DevOps (azure-pipelines.yml)

```
trigger:
  branches:
```

```yaml
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: TerraformInstaller@0
  displayName: 'Install Terraform'
  inputs:
    terraformVersion: '1.2.0'

- script: |
    terraform init
  displayName: 'Initialize Terraform'

- script: |
    terraform validate
  displayName: 'Validate Terraform Configuration'

- script: |
    terraform plan -out=tfplan
  displayName: 'Terraform Plan'

- script: |
    terraform apply -auto-approve tfplan
  displayName: 'Apply Terraform Changes'
```

Step 5: Secure Terraform Secrets Using Azure Key Vault

Create an Azure Key Vault

```
az keyvault create --name my-keyvault --resource-group terraform-rg --location eastus
```

Store Terraform Service Principal Credentials

```
az keyvault secret set --vault-name my-keyvault --name "terraform-client-id"
--value "<your-client-id>"
az keyvault secret set --vault-name my-keyvault --name "terraform-client-secret"
--value "<your-client-secret>"
```

Update Terraform Pipeline to Fetch Secrets

```
- task: AzureKeyVault@1
  inputs:
    azureSubscription: 'MyAzureSubscription'
    keyVaultName: 'my-keyvault'
    secretsFilter: '*'
```

Step 6: Deploy Terraform Infrastructure

1. Commit Terraform files to Azure DevOps Git repository
2. Run Azure DevOps Pipeline
3. Verify Resources in Azure Portal

Step 7: Destroy Infrastructure (Cleanup)

If you want to **destroy all resources**, run:

```
terraform destroy -auto-approve
```

---

5. Conclusion

Key Benefits of Terraform in Azure DevOps:

✅ **Automates Infrastructure** – No manual intervention
✅ **Ensures Consistency** – Infrastructure is identical across environments
✅ **Enables Rollbacks** – Easily revert infrastructure changes
✅ **Improves Security** – Secrets are stored in Azure Key Vault

Real-World Use Case:

A **financial services company** managing **multi-cloud infrastructure** can use Terraform for **repeatable, version-controlled** infrastructure deployments across **Azure, AWS, and Google Cloud**.

Project 10: Implementing Blue-Green Deployment Strategy in Azure DevOps

1. Project Scope

This project focuses on **Blue-Green Deployment** using **Azure DevOps Pipelines and Azure App Services**. The goal is to ensure **zero downtime deployment** by switching traffic between two identical environments (Blue & Green) in **a controlled manner**.

Key Features:

✅ **Deploy new versions without downtime**
✅ **Instant rollback if issues occur**
✅ **Ensure smooth traffic switching using Azure Traffic Manager**
✅ **Improve reliability in production deployments**

---

2. Tools Used

- **Azure DevOps Pipelines** – Automates application deployment
- **Azure App Services** – Hosts web applications
- **Azure Traffic Manager** – Routes traffic between Blue & Green
- **Azure Repos** – Stores application source code
- **Azure CLI & PowerShell** – Manages deployments

---

3. Analysis Approach

Traditional Deployment Issues:

❌ **Downtime** when updating production
❌ **Rollback complexity** if issues arise
❌ **Customer impact** during application restarts

Proposed Solution: Blue-Green Deployment

✅ Maintain **two identical environments (Blue & Green)**
✅ Deploy updates to **Green** while **Blue serves traffic**
✅ **Switch traffic** to Green once verified
✅ If issues occur, **switch back to Blue instantly**

Workflow:

1. **Deploy application to Blue (Production) and Green (Staging)**
2. **Update application in Green** and run tests
3. **Switch traffic** to Green if tests pass
4. **If failure occurs**, roll back to Blue

---

4. Step-by-Step Implementation

Step 1: Create Two Azure App Service Instances

Create a Resource Group:

```
az group create --name bluegreen-rg --location eastus
```

Create Two App Services for Blue & Green:

```
az webapp create --resource-group bluegreen-rg --plan myAppServicePlan --name
blue-app --runtime "DOTNETCORE:6.0"
```

```
az webapp create --resource-group bluegreen-rg --plan myAppServicePlan --name
green-app --runtime "DOTNETCORE:6.0"
```

Step 2: Set Up Azure Traffic Manager

Create Traffic Manager Profile:

```
az network traffic-manager profile create --resource-group bluegreen-rg --name
bluegreen-tm --routing-method Weighted --dns-name bluegreen-tm
```

Add Endpoints (Blue & Green) to Traffic Manager:

```
az network traffic-manager endpoint create --resource-group bluegreen-rg
--profile-name bluegreen-tm --name blue-endpoint --type azureEndpoints
--target-resource-id
/subscriptions/<sub_id>/resourceGroups/bluegreen-rg/providers/Microsoft.Web/sites/
blue-app --endpoint-status enabled --weight 100
```

```
az network traffic-manager endpoint create --resource-group bluegreen-rg
--profile-name bluegreen-tm --name green-endpoint --type azureEndpoints
--target-resource-id
/subscriptions/<sub_id>/resourceGroups/bluegreen-rg/providers/Microsoft.Web/sites/
green-app --endpoint-status enabled --weight 0
```

Step 3: Configure Azure DevOps Pipeline for Blue-Green Deployment

1. Go to Azure DevOps → Pipelines → Create New Pipeline
2. Select Git Repository
3. Add Blue-Green Deployment Stages

Pipeline YAML (azure-pipelines.yml)

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

variables:
```

```yaml
  blueAppServiceName: 'blue-app'
  greenAppServiceName: 'green-app'
  azureSubscription: 'AzureServiceConnection'

stages:
- stage: DeployToGreen
  jobs:
  - job: Deploy
    steps:
    - task: AzureWebApp@1
      displayName: 'Deploy to Green App'
      inputs:
        azureSubscription: $(azureSubscription)
        appName: $(greenAppServiceName)
        package: $(Build.ArtifactStagingDirectory)/app.zip

- stage: SwitchTraffic
  dependsOn: DeployToGreen
  jobs:
  - job: TrafficManagerUpdate
    steps:
    - script: |
        az network traffic-manager endpoint update --resource-group bluegreen-rg
--profile-name bluegreen-tm --name green-endpoint --weight 100
        az network traffic-manager endpoint update --resource-group bluegreen-rg
--profile-name bluegreen-tm --name blue-endpoint --weight 0
      displayName: 'Switch Traffic to Green'

- stage: Rollback
  condition: failed()
  dependsOn: SwitchTraffic
  jobs:
  - job: RevertTraffic
    steps:
    - script: |
        az network traffic-manager endpoint update --resource-group bluegreen-rg
--profile-name bluegreen-tm --name blue-endpoint --weight 100
        az network traffic-manager endpoint update --resource-group bluegreen-rg
--profile-name bluegreen-tm --name green-endpoint --weight 0
      displayName: 'Rollback to Blue'
```

Step 4: Test and Verify the Deployment

1. Trigger the Azure DevOps pipeline
2. Verify Green environment updates correctly
3. Ensure traffic switches to Green smoothly
4. If rollback triggers, confirm traffic returns to Blue

5. Conclusion

Key Benefits of Blue-Green Deployment:

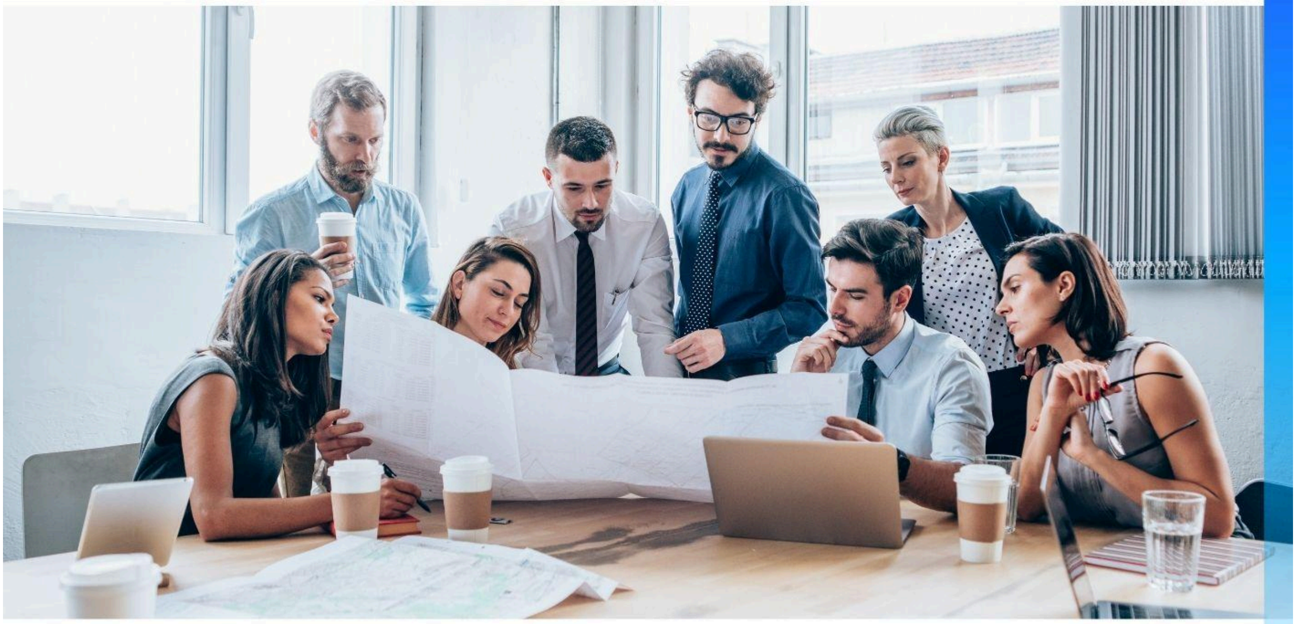
✅ Zero downtime deployment
✅ Immediate rollback in case of failure
✅ Ensures stable releases in production
✅ Traffic can be gradually shifted for controlled testing

Real-World Use Case:

A **banking application** that must be **available 24/7** can use this approach to deploy updates **without disrupting customers** and quickly rollback in case of an issue.

→ 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

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