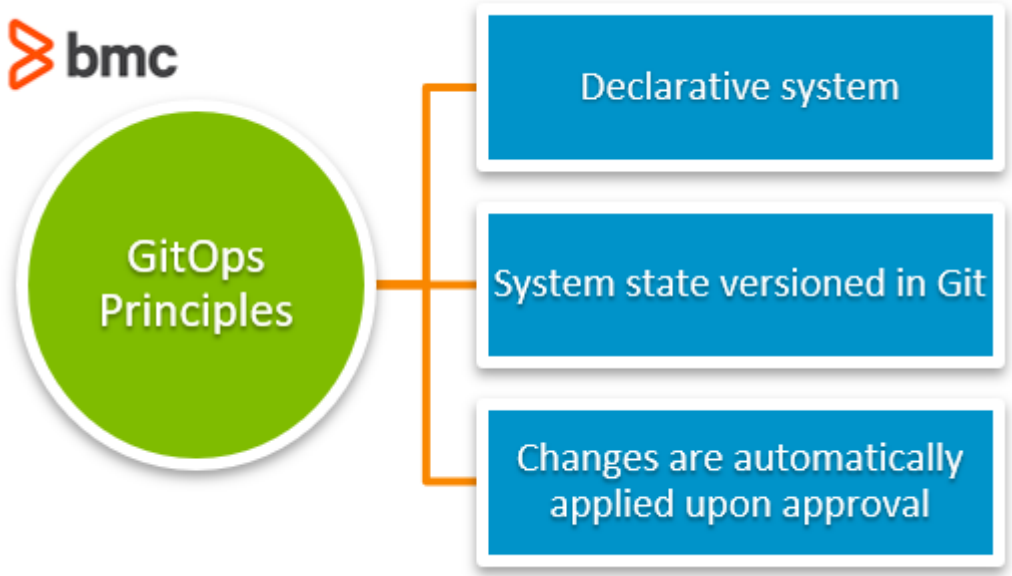
GitOps concept

GitOps is an operational framework that applies DevOps best practices used for application development, such as version control, collaboration, compliance, and CI/CD, to infrastructure automation. It aims to automate the process of provisioning infrastructure, especially modern cloud infrastructure, by using configuration files stored as code (Infrastructure as Code, or IaC).

Key Principles of GitOps



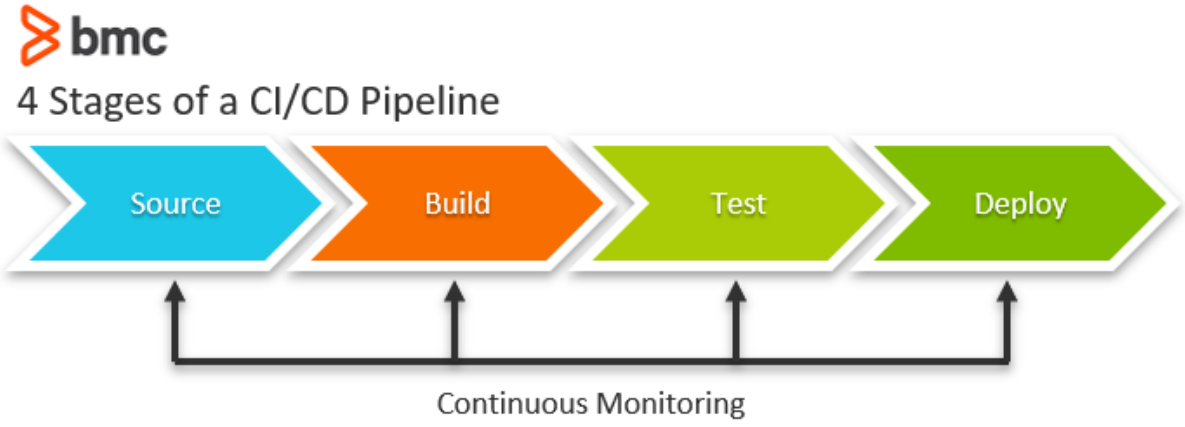
Infrastructure as Code (IaC): GitOps uses a Git repository as the single source of truth for infrastructure definitions. This means that all

infrastructure configurations are stored as code in a Git repository.

Merge Requests (MRs) or Pull Requests (PRs): GitOps uses MRs or PRs as the change mechanism for all infrastructure updates. This allows teams to collaborate via reviews and comments, and formal approvals take place here.

Continuous Integration and Continuous Delivery (CI/CD): GitOps automates infrastructure updates using a Git workflow with CI/CD. When new code is merged, the CI/CD pipeline enacts the change in the environment.

CI/CD:



CI/CD pipeline is a series of steps that automate the process of software delivery, from code integration to deployment. CI/CD stands for Continuous Integration and Continuous Delivery/Deployment, which are

practices aimed at improving software development and delivery through automation.

Continuous Integration (CI)

Continuous Integration involves developers frequently committing their code changes to a shared repository. Each commit triggers an automated build and test process, ensuring the new code integrates smoothly with the existing codebase. This practice helps in identifying and fixing bugs early, improving software quality, and reducing integration issues

Example of CI Process:

Code Commit: Developers commit code to a version control system like Git.

Build: An automated build process compiles the code and checks for errors.

Unit Tests: Automated tests run to verify the functionality of the code.

Code Review: Tools like SonarQube can be used for automated code reviews.

Continuous Delivery (CD)

Continuous Delivery extends CI by automating the release process. After the code passes the CI pipeline, it is deployed to a staging environment where further tests, such as integration and regression tests, are conducted. This ensures that the code is always in a deployable state

Example of CD Process:

Staging Deployment: The code is deployed to a staging environment.

Integration Tests: Automated tests check the integration of different modules.

Regression Tests: Tests ensure that new changes do not break existing functionality.

Manual Approval: A final manual check before deploying to production.

Create a multistage pipeline with Azure DevOps:

Setting Up a Project

Before diving into creating a pipeline, ensure you have an Azure DevOps account and a project created. Projects are containers for your pipelines, repositories, and other DevOps-related entities.

Creating Your First Pipeline

Defining Build Stages

To create a pipeline, navigate to the 'Pipelines' section in your Azure DevOps project and select 'New pipeline'. Choose the repository hosting your code and then opt for either the YAML or the Classic editor to define your pipeline.

Your first stage will typically be a build stage where your application is compiled:

```yaml
stages:
- stage: Build
  displayName: 'Build Stage'
  jobs:
  - job: BuildJob
    pool:
      vmImage: 'ubuntu-latest'
    steps:
    - script: make build
      displayName: 'Build Application'
```

Adding Test Stages:

```yaml
- stage: Test
  displayName: 'Test Stage'
  jobs:
  - job: TestJob
    pool:
      vmImage: 'ubuntu-latest'
    steps:
    - script: make test
      displayName: 'Run Tests'
```

Deployment Stages

Deployment stages can vary based on the environment (e.g., Development, Staging, Production). For each environment, you'd have a similar setup:

```yaml
- stage: DeployDev
  displayName: 'Deploy to Development'
  jobs:
  - deployment: DeployJob
    pool:
      vmImage: 'ubuntu-latest'
    environment: development
    strategy:
      runOnce:
        deploy:
          steps:
          - script: echo Deploying to Development
            displayName: 'Deploy Development'
```

Approvals and Gates

To ensure quality and control, you can add approvals and gates between stages. This is done from the Environments section in Azure DevOps where you can set checks that must be passed before moving on to the next stage.

Variables and Variable Groups

For managing configurations across stages, use variables and variable groups. These allow you to define environment-specific configurations without hardcoding values into your YAML file.

Tips for Successful Pipeline Creation

Keep your pipeline as simple as possible initially, then iterate and add complexity as needed.

Use templates for reusability across multiple pipelines or stages.

Regularly review your pipeline configurations to refine and optimize them.

Consider security best practices at every step of your pipeline configuration.

In complex scenarios or when looking to scale up rapidly, it may be beneficial to hire Azure DevOps engineers who can bring expertise to fine-tune your CI/CD processes effectively.

Build and deploy to Azure Kubernetes Service with Azure Pipelines:

Setting Up Azure Service Connections for ACR and AKS

To automate this deployment, you'll need two service connections in Azure DevOps:

Azure Container Registry Service Connection: This allows Azure DevOps to push the Docker image to ACR.

Kubernetes Service Connection: This connects Azure DevOps to your AKS cluster, enabling the deployment of the Docker image.

Steps to set up service connections:

In Azure DevOps, go to Project Settings > Service connections.

Create a new connection for Azure Container Registry using your ACR credentials and name it azureContainerSC.

Create a second service connection for Kubernetes that links to your AKS cluster, and name it kubernetesSC.

Understanding the Pipeline YAML File:

A complete Azure DevOps pipeline YAML file for building, pushing, and deploying a Dockerized application to AKS. We'll break down each part to clarify what it does.

trigger: Configures the pipeline to run automatically whenever there's a new commit to the main branch. This is useful for automating deployment every time code is updated.

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

pool: Specifies the agent pool to run the pipeline on. We use a Linux Ubuntu pool, which is compatible with Docker and Kubernetes tasks.

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

variables: Defines reusable values for the pipeline.

repositoryUrl: The Docker repository where the image will be pushed.

tagName: Generates a unique tag for each image using the build ID.

containerRegistrySC and kubernetesSC: Refer to the service connections for Docker Hub and AKS.

```
variables:
  repositoryUrl: "docker.io/iheanacho/webapi"
  tagName: $(Build.BuildId)
  containerRegistrySC: dockerhubSC
  kubernetesSC: eng-aks-demo
  appName: webapi
```

Build and Push Stage

The Build and Push stage builds the Docker image and pushes it to the container registry.

```
stages:
- stage: BuildAndPush
  displayName: "Build and Push Docker Image"
  jobs:
  - job: Build
    displayName: "Build and Push Docker Image to Container Registry"
    steps:
```

Docker@2: Builds and pushes the Docker image to the specified container registry.

container Registry: Uses the service connection for Docker Hub.

repository: Specifies the Docker repository URL.

command: Executes the build and push command.

Docker file: Specifies the path to the Docker file.

tags: Tags the image with the build ID.

```
    - task: Docker@2
      displayName: "Build and Push Docker Image"
      inputs:
        containerRegistry: "$(containerRegistrySC)"
        repository: $(repositoryUrl)
        command: 'buildAndPush'
        Dockerfile: '**/Dockerfile'
        tags: |
          $(tagName)
```

CopyFiles@2: Copies any Kubernetes YAML files to the staging directory. These files define the deployment and service configurations for AKS.

```
- task: CopyFiles@2
  inputs:
    SourceFolder: '$(System.DefaultWorkingDirectory)'
    Contents: '**/*.yaml'
    TargetFolder: '$(Build.ArtifactStagingDirectory)'
```

Running the Pipeline

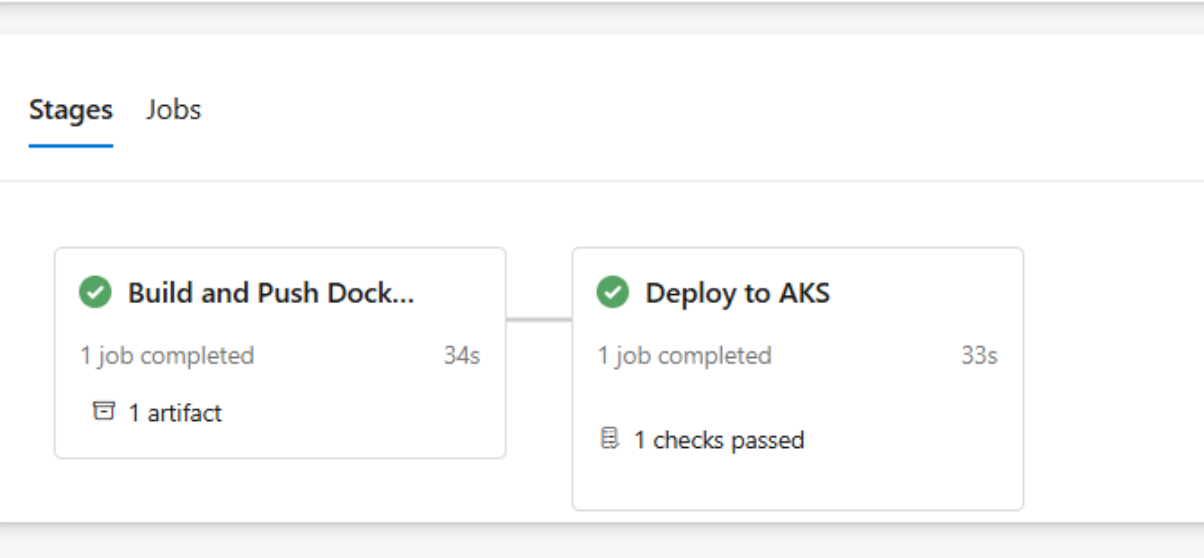
Commit and Push the pipeline YAML file to the main branch in your repository.

In Azure DevOps, navigate to Pipelines and trigger the pipeline.

The pipeline will:

Build and push the Docker image to Azure Container Registry.

Deploy the application to AKS using the Kubernetes YAML manifest.



Verifying the Deployment on AKS

After the pipeline completes, you can verify the application's deployment on AKS:

Check the Deployment

kubectl get all -n webapi

Retrieve the External IP:

kubectl get svc -n webapi

Once the external IP is assigned, open http://<EXTERNAL-IP>/ in your browser to confirm the application is live.