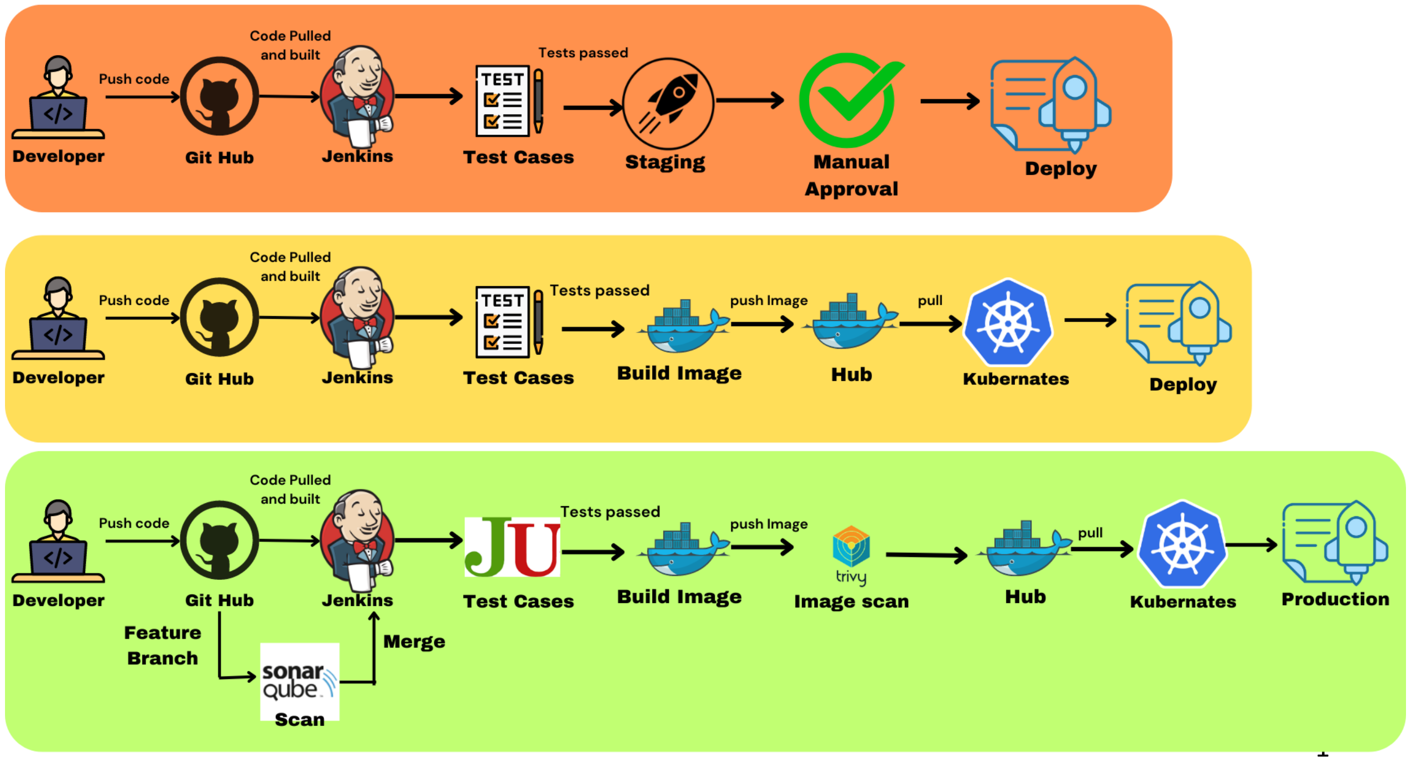
DevOps Shack

Understanding CI/CD Pipelines

1. Introduction

Continuous Integration (CI) and Continuous Deployment (CD) are practices that automate the process of software development, enabling teams to deliver code changes more frequently and reliably. CI focuses on integrating code changes into a shared repository frequently, while CD ensures that these changes are automatically deployed to production.



2. CI/CD Pipeline Concepts

Before diving into specific scenarios, let's establish a clear understanding of the key concepts in CI/CD pipelines:

- **Version Control System (VCS)**: A system like Git that helps manage changes to code.
- **Continuous Integration (CI)**: The practice of merging all developers' working copies to a shared mainline several times a day.
- **Continuous Deployment (CD)**: The practice of deploying every change that passes automated tests to production.
- **Build Automation**: Automating the process of compiling code into binaries or packages.
- **Testing Automation**: Automatically running tests to verify code quality.
- **Deployment Automation**: Automatically deploying applications to different environments (e.g., staging, production).

3. Basic CI/CD Pipeline (Build-Test-Deploy)

3.1 Key Steps

A basic CI/CD pipeline consists of several key steps:

1. **Code Commit**: Developers commit their code changes to a version control system like Git.
2. **Build**: A CI server retrieves the latest code and builds the application.
3. **Testing**: Automated tests are executed to ensure the code works as expected.
4. **Deployment**: The application is deployed to a staging environment for further testing and then to production.

3.2 Example Code Implementation

In this example, we will use **Jenkins** as our CI/CD tool and a simple Node.js application to illustrate the basic CI/CD pipeline.

3.2.1 Setting Up the Node.js Application

Create a simple Node.js application. Below is a sample app.js file:

```
// app.js
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(port, () => {
  console.log(`App listening at http://localhost:${port}`);
});
```

3.2.2 Setting Up Jenkins

1. **Install Jenkins**:
   - Follow the Jenkins installation guide to set up Jenkins on your machine or server.
2. **Create a New Job**:
   - Open Jenkins in your browser and create a new job by clicking on "New Item."

   - Select "Freestyle project" and give it a name (e.g., "Basic-CI-CD-Pipeline").
3. **Configure Source Code Management**:
   - In the job configuration, select "Git" as the source code management option and provide the URL of your Git repository.
4. **Build Triggers**:
   - Select "GitHub hook trigger for GITScm polling" to trigger builds on code changes.
5. **Add Build Steps**:
   - In the build section, add the following shell command to install dependencies and run tests:

```
#!/bin/bash
npm install
npm test
```

6. **Post-build Actions**:
   - Add a post-build action to deploy the application to your staging environment (you can use a shell script or another tool for deployment).

3.2.3 Running the Pipeline

1. **Commit Code**: Make changes to your application and commit them to the Git repository.
2. **Build Trigger**: The Jenkins job is triggered automatically when changes are pushed.
3. **Build and Test**: Jenkins builds the application and runs the tests.
4. **Deployment**: If all tests pass, Jenkins deploys the application to the staging environment.

4. CI/CD Pipeline with Docker and Kubernetes

4.1 Key Steps

Incorporating Docker and Kubernetes into the CI/CD pipeline allows for containerization and orchestration, leading to better scalability and management.

1. **Code Commit**: Developers commit code changes to a Git repository.
2. **Docker Image Build**: The CI server builds a Docker image from the code.
3. **Push to Container Registry**: The built image is pushed to a container registry (e.g., Docker Hub).
4. **Kubernetes Deployment**: The CI/CD pipeline triggers the deployment of the Docker image to a Kubernetes cluster, first in a staging environment.
5. **Testing**: Automated tests are run in the staging environment to validate functionality.
6. **Production Deployment**: After passing tests and obtaining approvals, the application is deployed to the production environment.

4.2 Example Code Implementation

In this example, we will use **GitHub Actions** for CI/CD, **Docker** for containerization, and **Kubernetes** for orchestration.

4.2.1 Setting Up the Node.js Application

You can use the same Node.js application as described in the previous section.

4.2.2 Dockerfile

Create a Dockerfile in the root of your project:

```
# Dockerfile

FROM node:14
```

```
# Set the working directory
WORKDIR /usr/src/app

# Copy package.json and install dependencies
COPY package*.json ./
RUN npm install

# Copy the rest of the application code
COPY . .

# Expose the port the app runs on
EXPOSE 3000

# Command to run the application
CMD ["node", "app.js"]
```

4.2.3 GitHub Actions Workflow

Create a .github/workflows/ci-cd.yml file in your repository:

```
name: CI/CD Pipeline

on:
  push:
    branches:
      - main
```

```
jobs:
 build:
   runs-on: ubuntu-latest

   steps:
     - name: Checkout code
       uses: actions/checkout@v2
     - name: Set up Node.js
       uses: actions/setup-node@v2
       with:
         node-version: '14'
     - name: Install dependencies
       run: npm install
     - name: Run tests
       run: npm test
     - name: Build Docker image
       run: docker build . -t my-node-app
     - name: Log in to Docker Hub
       run: echo "${{ secrets.DOCKER_HUB_PASSWORD }}" | docker login -u "${{
secrets.DOCKER_HUB_USERNAME }}" --password-stdin
     - name: Push Docker image to Docker Hub
       run: docker push my-node-app
```

4.2.4 Kubernetes Deployment

Create a deployment.yaml file for your Kubernetes deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-node-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-node-app
  template:
    metadata:
      labels:
        app: my-node-app
    spec:
      containers:
      - name: my-node-app
        image: my-node-app:latest
        ports:
        - containerPort: 3000
```

4.2.5 Deploying to Kubernetes

You can deploy to your Kubernetes cluster using the following command:

```
kubectl apply -f deployment.yaml
```

5. Advanced CI/CD Pipeline with Feature Branching and Quality Gates

5.1 Key Steps

This pipeline uses feature branching to isolate code changes and includes quality gates to ensure high standards before merging into the main branch.

1. **Feature Branch Development**: Developers create feature branches to work on isolated changes.
2. **Code Quality Analysis**: SonarQube runs code quality checks on the feature branch.
3. **Pull Request**: Developers submit a pull request for their feature branch to be merged into the main branch.
4. **Merge to Main Branch**: Upon approval, the feature branch is merged.
5. **Automated Build**: The CI server automatically builds the main branch after the merge.
6. **Automated Testing**: A comprehensive suite of tests is run on the main branch.
7. **Staging Deployment**: The application is deployed to a staging environment for further validation.
8. **Acceptance Testing**: The application undergoes acceptance testing in staging.
9. **Production Deployment**: After successful testing and approval, the application is deployed to the production environment.

5.2 Example Code Implementation

In this example, we will use **GitLab CI** for our CI/CD process along with **SonarQube** for code quality analysis.

5.2.1 Setting Up SonarQube

1. **Install SonarQube**: Follow the SonarQube installation guide to set up SonarQube on your local machine or server.
2. **Create a New Project**: Log in to SonarQube and create a new project for your application.

5.2.2 GitLab CI Configuration

Create a .gitlab-ci.yml file in your repository:

```
stages:
  - test
  - build
  - deploy

variables:
  SONAR_USER_HOME: "$CI_PROJECT_DIR/.sonar"

before_script:
  - npm install

test:
  stage: test
  script:
    - npm test
  artifacts:
```

```
  paths:
    - coverage/

sonar:
 stage: test
 script:
   - sonar-scanner -Dsonar.projectKey=my-node-app -Dsonar.sources=. -
Dsonar.host.url=http://your-sonarqube-server -Dsonar.login=$SONAR_TOKEN

build:
 stage: build
 script:
   - docker build -t my-node-app .

deploy_staging:
 stage: deploy
 script:
   - kubectl apply -f deployment.yaml
 environment:
   name: staging
   url: http://staging.yourapp.com

deploy_production:
 stage: deploy
```

```
  when: manual
  script:
    - kubectl apply -f deployment.yaml
  environment:
    name: production
    url: http://yourapp.com
```

5.2.3 Running the Pipeline

1. **Feature Branch Development**: Developers create a new feature branch.
2. **Code Quality Analysis**: As soon as a merge request is created, the SonarQube analysis runs automatically.
3. **Automated Testing**: Tests run on the feature branch.
4. **Merge to Main Branch**: Once all checks pass, the feature branch can be merged into the main branch.
5. **Deployment**: The CI/CD pipeline deploys the application to the staging environment. Manual approval is required to deploy to production.

6. Conclusion

CI/CD pipelines are crucial for modern software development, enabling teams to deliver high-quality software quickly and efficiently. In this document, we explored three different CI/CD pipeline scenarios: the Basic CI/CD Pipeline, the CI/CD Pipeline with Docker and Kubernetes, and the Advanced CI/CD Pipeline with Feature Branching and Quality Gates.

Each scenario highlights different practices and tools that can be used to automate the software delivery process. By implementing these pipelines, organizations can improve collaboration, increase deployment frequency, and reduce the risk of errors in production.