Assignment: CI/CD Pipeline with Automated Testing and Deployment Strategies

---

Part 1: CI with Automated Testing

Objective:

Set up a simple **Continuous Integration (CI)** pipeline with **automated testing** using **GitHub Actions**.

Step 1: Create a Simple Web Application

1. **Create a simple Node.js application** (or use any language of your choice):
   - Initialize the project:

     ```
     bash
     Copy
     mkdir my-ci-app
     cd my-ci-app
     npm init -y
     npm install express
     ```

   - Create an index.js file for a simple API:

     ```
     javascript
     Copy
     const express = require('express');
     const app = express();
     const port = 3000;

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

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

   - Add a simple unit test using **Mocha** and **Chai**:

     ```
     bash
     Copy
     ```

```
npm install mocha chai --save-dev
```

- Create a test file test/app.test.js:

```
javascript
Copy
const chai = require('chai');
const expect = chai.expect;

describe('GET /hello', () => {
  it('should return "Hello World!"', (done) => {
    const http = require('http');
    http.get('http://localhost:3000/hello', (res) => {
      let data = '';
      res.on('data', chunk => { data += chunk; });
      res.on('end', () => {
        expect(data).to.equal('Hello World!');
        done();
      });
    });
  });
});
```

Step 2: Set Up CI with GitHub Actions

1. **Push your code to GitHub**:
   - Initialize a GitHub repository and push the code.
2. **Create GitHub Actions Workflow**:
   - In your GitHub repository, create a .github/workflows/ci.yml file:

```
yaml
Copy
name: Node.js CI

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Node.js
      uses: actions/setup-node@v2
      with:
        node-version: '14'
    - run: npm install
    - run: npm test
      env:
        CI: true
```

3. **Test the CI Pipeline**:
   - Push the code to the repository and make sure that the GitHub Actions pipeline runs. Check if the tests pass.

Part 2: Continuous Deployment (CD) with Deployment Strategies

Objective:

Set up a **Continuous Deployment (CD)** pipeline that deploys to a cloud service (e.g., **Heroku**) and includes **Canary Releases** and **Rollback** strategies.

Step 1: Set Up Heroku Deployment

1. **Create a Heroku App**:
   - Sign up for Heroku if you don't have an account.
   - Install the Heroku CLI.
   - Log in to Heroku from your terminal:

     ```
     bash
     Copy
     heroku login
     ```

2. **Deploy the App to Heroku**:
   - Create a Heroku app:

     ```
     bash
     Copy
     heroku create my-ci-app
     ```

   - Push your application to Heroku:

bash
Copy
```
git push heroku main
```

Step 2: Set Up Continuous Deployment with GitHub Actions to Heroku

1. **Create a Deployment Workflow in GitHub Actions**:
   - Add a new file in .github/workflows/deploy.yml:

     yaml
     Copy
     ```
     name: Deploy to Heroku

     on:
       push:
         branches:
           - main

     jobs:
       deploy:
         runs-on: ubuntu-latest

         steps:
         - uses: actions/checkout@v2
         - name: Set up Node.js
           uses: actions/setup-node@v2
           with:
             node-version: '14'
         - run: npm install
         - run: npm run build
         - name: Deploy to Heroku
           env:
             HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
           run: |
             git remote add heroku https://git.heroku.com/my-ci-app.git
             git push heroku main
     ```

2. **Add Heroku API Key to GitHub Secrets**:
   - Go to your GitHub repository > Settings > Secrets > New repository secret.
   - Add your **HEROKU_API_KEY** (you can generate it from Heroku CLI).
3. **Test the Deployment Pipeline**:
   - Push the code to trigger the deployment. GitHub Actions should automatically deploy the app to Heroku.

Step 3: Implement Canary Releases and Rollbacks

1. **Canary Releases**:
   - For simplicity, you can simulate **Canary Releases** by deploying to two different environments (e.g., staging and production) and routing a small portion of traffic to the new version.
   - **Bonus**: Use **Heroku Pipelines** or **AWS Elastic Beanstalk** for routing small traffic to a new version.
2. **Rollback**:
   - If the Canary release fails, you can **rollback** to the previous version by running:

     bash
     Copy
     ```
     heroku releases:rollback
     ```

Deliverables:

1. **Web Application Code** (including automated tests).
2. **CI Pipeline** (GitHub Actions configuration file for testing).
3. **CD Pipeline** (GitHub Actions configuration file for deployment to Heroku).
4. **Canary Release Setup** (Documentation or screenshots of Canary strategy, if implemented).
5. **Rollback Strategy** (Documentation or screenshots of rollback procedure).

Evaluation Criteria:

- **CI Setup**: Correct configuration of GitHub Actions for automated testing.
- **CD Setup**: Successful deployment of the app to Heroku using GitHub Actions.
- **Canary Releases**: Canary release configuration, or at least a description of the strategy.
- **Rollback Strategy**: Ability to quickly revert to the previous working version on failure.