CI/CD Pipeline for Online Store Microservice

Introduction to CI/CD

Continuous Integration (CI) and Continuous Delivery/Deployment (CD) are essential practices for modern software development. CI focuses on integrating code changes frequently, triggering automated builds and tests to detect errors early. CD automates the deployment process, ensuring that the code is always in a deployable state, with Continuous Deployment (CD) pushing changes to production automatically after passing tests. While CI ensures code quality, CD ensures faster, reliable delivery.

State-of-the-Art Practices and Tools

1. **GitOps & IaC**: Infrastructure as Code (IaC) and GitOps enable version-controlled infrastructure changes. Tools like Terraform are commonly used to manage infrastructure alongside application code.
2. **Cloud-Native CI/CD**: Cloud-native CI/CD platforms such as GitHub Actions and GitLab CI/CD integrate with Kubernetes, supporting scalable build systems.
3. **Pipeline as Code**: Pipelines defined as code, such as Jenkinsfiles, are stored alongside application code, making them reusable, version-controlled, and self-documented.
4. **Observability & Feedback**: Monitoring tools like Prometheus and Grafana allow for real-time performance monitoring and production feedback.

User Stories for the Microservice

The Online Store Microservice was developed based on user stories that define different functionalities, including customer account management, order placement, and admin control. It contains features such as HATEOAS navigation, error handling, pagination and date filtering.

Architecture

The application uses a layered architecture in Spring Boot:

- **Controller Layer**: Handles HTTP requests and implements REST API endpoints.
- **Service Layer**: Contains the business logic and manages transactions.
- **Repository Layer**: Interacts with the database via Spring Data JPA.
- **Model Layer**: Defines domain entities (e.g., `Customer`, `Order`).
- **DTO Layer**: Transforms data for API responses, decoupling the internal model.

Test Strategy

Following the Test Pyramid approach, the strategy involves:

- **Unit Tests**: Focus on testing individual components in isolation.
- **Integration Tests**: Validate interactions between components (e.g., between services and repositories).
- **End-to-End Tests**: Test complete workflows and user journeys.

Example of a unit test for the `CustomerService`:

```
@Test
void createCustomer_WithValidCustomer_ShouldReturnSavedCustomer() {
    // Given
    Customer inputCustomer = new Customer();
    inputCustomer.setName("New Customer");
    inputCustomer.setEmail("new@example.com");
    inputCustomer.setAddress("456 New Street");

when(customerRepository.save(any(Customer.class))).thenReturn(testCustomer)
;
    // When
    Customer savedCustomer = customerService.createCustomer(inputCustomer);
    // Then
    assertNotNull(savedCustomer);
    assertEquals(testCustomer.getId(), savedCustomer.getId());
    verify(customerRepository, times(1)).save(any(Customer.class));
}
```

CI/CD Pipeline Implementation

The CI/CD pipeline automates the build, test and deployment processes. Key stages include:

1. **Code Checkout**: Fetches the latest code from GitHub.

```
stage('Checkout Code') {
    steps {
        git branch: 'main', url:
'https://github.com/michaeljosephroddy/online-store-service.git'
    }
}
```

2. **Build and Test**: Uses Maven to compile the application and run all tests.

```
stage('Build and Package') {
    steps {
        sh "mvn clean package verify"
    }
}
```

3. **Static Code Analysis**: Uses SonarQube to analyse code quality and security issues.

```
stage('Static Code Analysis') {
    steps {
        withSonarQubeEnv('SonarQube') {
            sh "mvn sonar:sonar -Dsonar.projectKey=sonar-project-local
-Dsonar.projectName='sonar-project-local'"
```

```
        }
    }
}
```

4. **Containerisation**: Packages the application into a Docker container.

```
stage('Build and Push Docker Image') {
    steps {
        sh 'docker build -t michaelroddy04/online-store-service .'
        withDockerRegistry([credentialsId: 'dockerhub-credentials',
url: '']) {
            sh 'docker tag online-store-service michaelroddy04/online-
store-service'
            sh 'docker push michaelroddy04/online-store-service'
        }
    }
}
```

5. **Deployment**: Automates the deployment using Ansible.

```
stage('Run Ansible for Automated Deployment') {
    steps {
        sh 'sudo chmod 400 lab1webserverkeypair.pem'
        sh 'ansible-playbook -i inventory.ini deploy.yml'
    }
}
```

Tool Selection Evaluation

- **Jenkins**: The primary automated server platform, chosen for its flexibility and plugin ecosystem.
- **Maven**: Used for compiling and building code.
- **SonarQube**: Provides static code analysis to ensure code quality.
- **Docker**: Ensures consistent environments across all stages by containerising the application.
- **Ansible**: Automates the deployment process with agentless architecture.

Pipeline Evaluation

The CI/CD pipeline executes in about a few minutes, with minimal manual intervention. Processes are automated, from code check in to deployment enhancing consistency and reliability. The pipeline is efficient, but further improvements like automated performance testing and security scans could enhance it.

Conclusion

The CI/CD pipeline automates the build, test, and deployment processes for the Online Store Microservice, supporting fast, reliable software delivery. The selected tools Jenkins, Maven, SonarQube, Docker, and Ansible are suitable to the project needs. While the pipeline is already efficient, further automation in areas like performance testing and security scanning would strengthen its reliability and safety.