CI/CD d’un projet e-commerce microservices avec GitHub Actions & Docker

Mohamed Fedi Guizani
Hamdi Ben Ghribi

1. Introduction

L’essor des architectures microservices a transformé la manière dont les applications sont conçues, déployées et maintenues. Contrairement aux architectures monolithiques, les microservices divisent l’application en plusieurs composants indépendants, chacun responsable d’une fonctionnalité métier bien définie. Cette modularité apporte de nombreux avantages comme la scalabilité, la maintenabilité, et la flexibilité technologique. Cependant, elle introduit aussi une complexité accrue au niveau de l’intégration, du déploiement et de la supervision des services.

Dans ce contexte, la mise en place d’une stratégie **CI/CD (Intégration Continue / Déploiement Continu)** devient non seulement bénéfique, mais essentielle. Elle permet d’automatiser l’ensemble du cycle de vie des microservices : depuis la récupération du code source jusqu'à la mise en production. Une bonne stratégie CI/CD garantit des livraisons fréquentes, fiables, traçables et reproductibles.

Le présent rapport s’intéresse à l’implémentation CI/CD dans le projet e-commerce-microservices-architecture, un système de vente en ligne construit avec une architecture de microservices conteneurisés. Ce projet exploite des outils modernes tels que **Spring Boot**, **Docker**, **GitHub Actions**, **Kubernetes** et **Helm**, ce qui en fait un excellent cas d’étude pour comprendre comment orchestrer une chaîne CI/CD complète dans un environnement distribué.

Ce rapport a pour objectifs de :

- Décrire les concepts fondamentaux du CI/CD dans le contexte microservices.
- Présenter les outils utilisés dans le projet et leur rôle.
- Analyser en détail le pipeline CI/CD mis en œuvre via **GitHub Actions**.
- Démontrer comment les services sont testés, buildés, empaquetés, puis déployés automatiquement.
- Identifier les limites du système actuel et proposer des pistes d’amélioration.

Cette analyse détaillée permettra de mieux comprendre l’impact de la CI/CD sur la productivité, la qualité logicielle et la gestion du cycle de vie d’un projet en microservices.

2.Les enjeux DevOps et CI/CD

Dans les environnements de développement modernes, le paradigme **DevOps** est devenu central pour aligner les équipes de développement et d'exploitation autour d'un objectif commun : livrer rapidement et efficacement des logiciels fiables. Le **CI/CD** est le socle technique de cette collaboration, assurant que chaque modification de code peut être intégrée, testée, empaquetée, et déployée sans intervention manuelle.

Pour les projets microservices, ces pratiques ne sont plus une option mais une **nécessité**. Chaque service évoluant de manière indépendante, il devient impératif de disposer :

- d'un pipeline d'intégration automatisé pour valider chaque commit,
- d'une gestion cohérente des versions et des dépendances,
- et d'un processus de déploiement sécurisé et traçable pour éviter les erreurs humaines.

Sans cela, le risque d'avoir des versions incompatibles entre services, des erreurs en production ou des retards de livraison devient très élevé.

3. Présentation du projet

Le projet **e-commerce-microservices-architecture** est une application de type e-commerce construite en suivant les principes de l'**architecture microservices**. L'objectif de ce projet est de fournir une solution modulable, distribuée et facilement maintenable qui reproduit le fonctionnement d'une boutique en ligne moderne : gestion des utilisateurs, des produits, des commandes et des paiements.

Chaque composant fonctionnel est conçu comme un microservice autonome, développé principalement avec **Java 17** et **Spring Boot**, et exécuté dans un conteneur Docker. Ces microservices communiquent entre eux à travers des API REST ou des événements via **Kafka**. L'ensemble du système est orchestré à l'aide de **Docker Compose** pour l'environnement local, et **Kubernetes (via Helm)** pour un déploiement en environnement cloud.

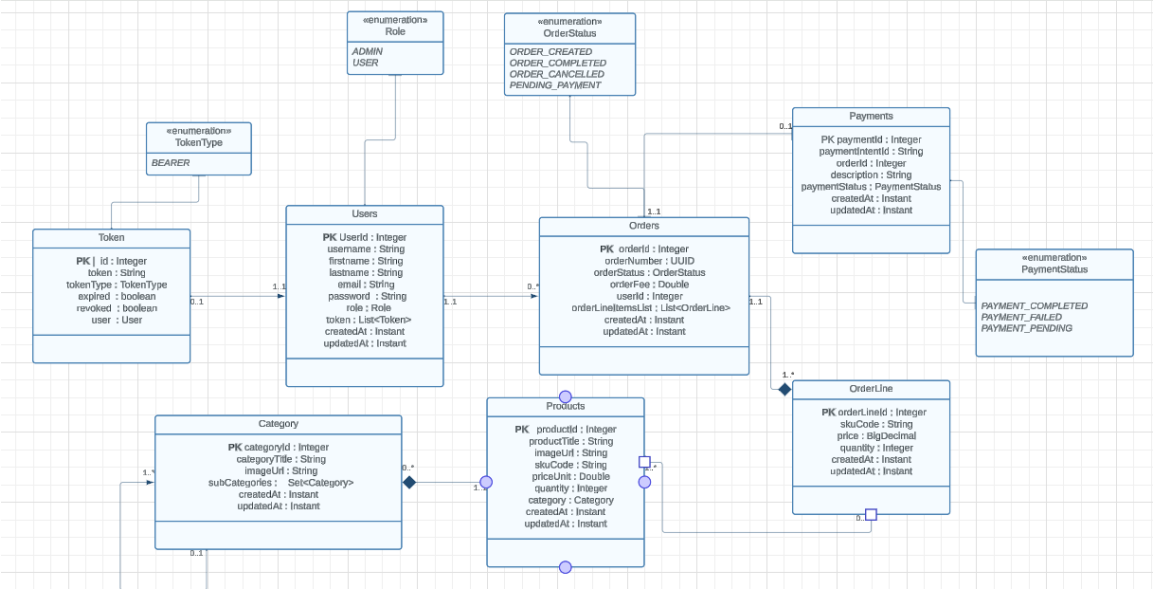
Architecture microservices

Le projet comprend plusieurs microservices, chacun ayant une responsabilité métier claire :

Microservice	Rôle principal

`user-service`	Gestion des utilisateurs (création, login, rôles)
`product-service`	Gestion du catalogue produits et des catégories
`order-service`	Prise en charge des commandes et de leur statut
`payment-service`	Gérer les paiements des commandes de l'application
`config-server`	Serveur de configuration centralisée
`eureka-server`	Service de découverte (service registry)
`api-gateway`	Point d'entrée unique pour les clients (reverse proxy)

Voici le diagramme du classe du projet :



Technologies et outils utilisés

Voici les technologies clés utilisées dans le projet :

- **Langage & Frameworks** : Java 17, Spring Boot, Spring Cloud
- **Conteneurisation** : Docker, Docker Compose
- **Orchestration** : Kubernetes (Helm charts)
- **CI/CD** : GitHub Actions (pour automatiser les étapes build, test, déploiement)
- **Gestion des configurations** : Spring Cloud Config
- **Service Discovery** : Eureka
- **Communication inter-services** : REST et Kafka
- **Base de données** : PostgreSQL

Mode de fonctionnement global

1. L'utilisateur interagit avec l'**API Gateway** qui route les requêtes vers le bon microservice.
2. Les services se découvrent entre eux dynamiquement via **Eureka**.
3. Chaque service a sa propre base de données (architecture **database-per-service**).
4. La configuration des services est centralisée via le **Config Server**.
5. Lorsqu'un utilisateur effectue une commande :
    a. `order-service` envoie un événement via **Kafka**
    b. `payment-service` traite le paiement
    c. Une réponse est retournée à l'utilisateur via l'API Gateway

Environnements d'exécution

Le projet supporte deux environnements principaux :

- **Local** : via Docker Compose pour lancer tous les services localement.
- **Cloud ou cluster K8s** : via Helm pour un déploiement modulaire, scalable et automatisé dans un cluster Kubernetes.

4. Concepts CI/CD

Dans le développement logiciel moderne, les concepts de **CI (Continuous Integration)** et **CD (Continuous Delivery / Deployment)** sont au cœur des pratiques **DevOps**. Leur adoption permet d'accélérer les livraisons, d'améliorer la qualité logicielle, et de réduire les risques lors des mises en production.

Dans le contexte du projet **e-commerce-microservices-architecture**, où plusieurs microservices interagissent, la mise en œuvre du CI/CD permet de maintenir une cohérence entre les composants, de détecter rapidement les erreurs, et de garantir un déploiement automatisé et reproductible.

4.1 Intégration Continue (CI)

L'intégration continue est une pratique qui consiste à :

- Intégrer fréquemment les modifications de code (souvent plusieurs fois par jour)
- Compiler automatiquement l'application dès qu'un changement est détecté
- Exécuter automatiquement des **tests unitaires et d'intégration**

- Alerter les développeurs en cas d'erreur

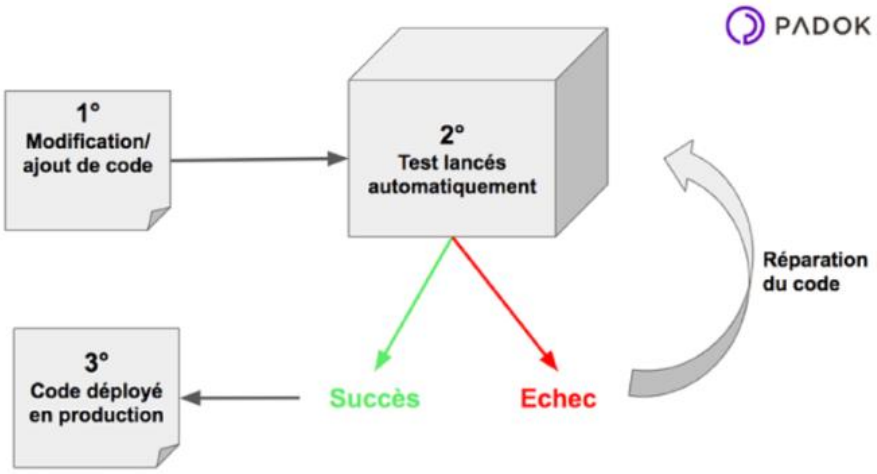
Objectifs :

- Identifier rapidement les erreurs d'intégration
- Maintenir un tronc commun fonctionnel
- Réduire les conflits lors des fusions de branches

Dans ce projet, l'intégration continue est assurée via **GitHub Actions**, qui déclenche le pipeline à chaque **push** sur la branche principale.

Qu'apporte l'intégration continue ?

Le principal apport de l'intégration continue est de garantir la mise en production d'un code de qualité, et donc une meilleure satisfaction des utilisateurs finaux. En effet, l'automatisation des tests de tout le code source à chaque ajout/modification de fonctionnalités permet d'éviter l'introduction de régressions en production. Les développeurs peuvent configurer les notifications du serveur d'intégration et ainsi être prévenus sur le service de leur choix en cas d'anomalies (webhook, email, etc…), gagnant ainsi un temps précieux. En réalité, ce n'est pas la totalité du code qui est testée, car le développement des tests prend du temps. Chez Padok, on considère qu'à partir de 80% on garantit une bonne couverture de test. Mieux vaut avoir les principales fonctionnalités mises à l'épreuves plutôt que toute l'application mal testée. L'intégration continue permet également aux Dev d'avoir un retour plus rapide sur leur développement. Il n'y a donc plus besoin d'attendre plusieurs semaines pour identifier les erreurs et les corriger. Enfin l'intégration continue favorise le travail en équipe. Souvent, plusieurs Dev travaillent sur des tâches séparées dans le cadre d'un même projet. Or, plus les équipes sont importantes, plus le risque d'introduire des erreurs lors de l'intégration est élevé



4.2 Déploiement Continu (CD)

Le déploiement continu est l'étape qui suit l'intégration continue. Il consiste à :

- Construire les **images Docker** des microservices
- Les **pousser vers un registre Docker** (ex : DockerHub)
- **Déployer automatiquement** les images dans un environnement d'exécution (via Docker Compose ou Kubernetes)

Objectifs :

- Accélérer la mise en production
- Supprimer les tâches manuelles de déploiement
- Réduire les erreurs humaines
- Tester plus rapidement en environnement réel

Le CD permet ici d'avoir un **pipeline automatisé de bout en bout**, allant de la récupération du code source à son exécution dans un environnement déployé.

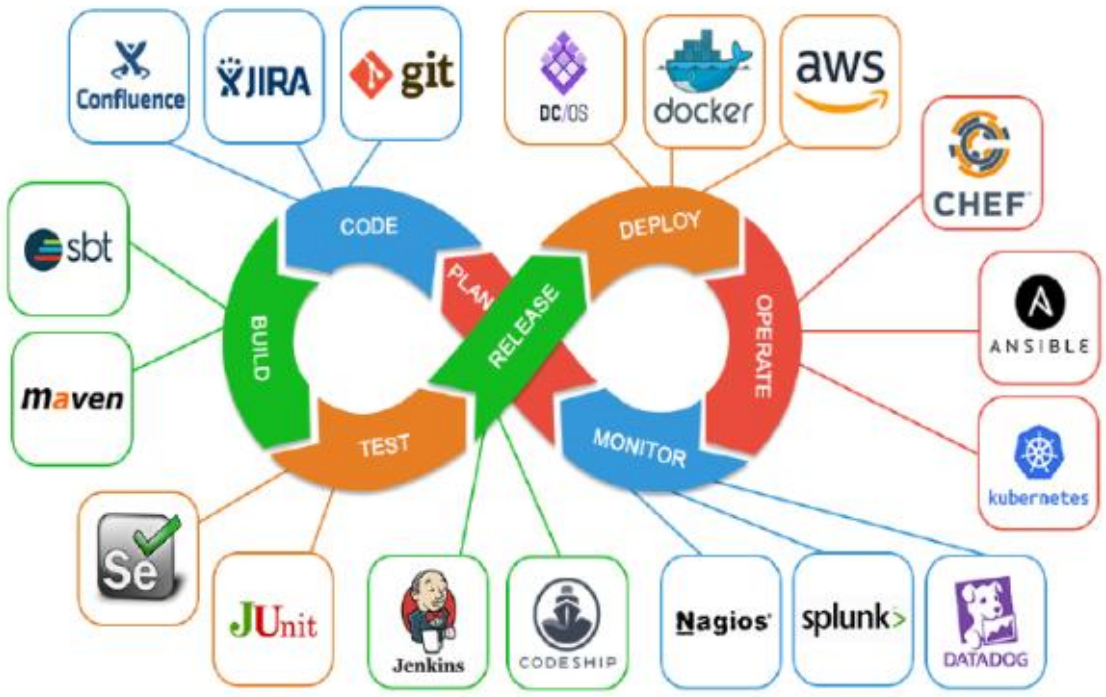
CI/CD dans un environnement microservices

Dans une architecture microservices, chaque service peut évoluer indépendamment. Sans CI/CD, cela entraîne :

- Des risques d'incompatibilité entre services
- Des délais de mise à jour
- Une complexité manuelle importante

Le CI/CD dans ce projet permet :

- De tester et builder chaque service isolément
- De garantir que les services sont compatibles au moment du déploiement
- De déployer uniquement les services modifiés (si nécessaire)



4. Outils utilisés pour la CI/CD

La réussite d'un pipeline CI/CD moderne repose sur une combinaison d'outils spécialisés, chacun jouant un rôle précis dans l'automatisation du processus.
Le projet **e-commerce-microservices-architecture** adopte un ensemble d'outils open source robustes et bien intégrés dans l'écosystème DevOps, permettant une livraison fluide et rapide du code source jusqu'au déploiement final.

Voici les principaux outils utilisés dans ce projet :

1. GitHub Actions – *Orchestration du pipeline CI/CD*

GitHub Actions est le moteur d'automatisation natif de GitHub, utilisé ici pour :

- Déclencher les pipelines à chaque `push` ou `pull request`
- Exécuter les étapes de compilation, test, build d'images Docker, et déploiement
- Gérer les secrets (tokens, credentials) de manière sécurisée
- Permettre des workflows parallèles ou conditionnels

✅Avantage : Tout est centralisé dans le dépôt GitHub — pas besoin de serveur externe comme Jenkins.

2. Maven – *Build, dépendances et tests Java*

Maven est utilisé dans chaque microservice pour :

- Compiler le code Java
- Résoudre les dépendances (`pom.xml`)
- Lancer les tests automatisés (`mvn test`)
- Générer les artefacts `.jar` nécessaires à la construction des images Docker

Il est intégré dans le pipeline GitHub Actions via une étape `run: mvn clean install`.

3. Docker – *Conteneurisation des microservices*

Docker est utilisé pour isoler chaque service dans un conteneur. Chaque microservice dispose :

- D'un `Dockerfile` personnalisé
- D'une image construite automatiquement à chaque build CI
- D'une image poussée vers un registre Docker (ex. DockerHub)

Cela permet d'avoir des environnements **identiques entre développement, test, staging et production**.

4. Docker Compose – *Orchestration locale*

Lors du développement ou des tests d'intégration, Docker Compose est utilisé pour :

- Démarrer tous les services ensemble (`docker-compose.yml`)
- Simuler un environnement complet (DB, Kafka, Config Server, etc.)
- Faciliter les tests end-to-end en local

5. Kubernetes + Helm – *Déploiement cloud*

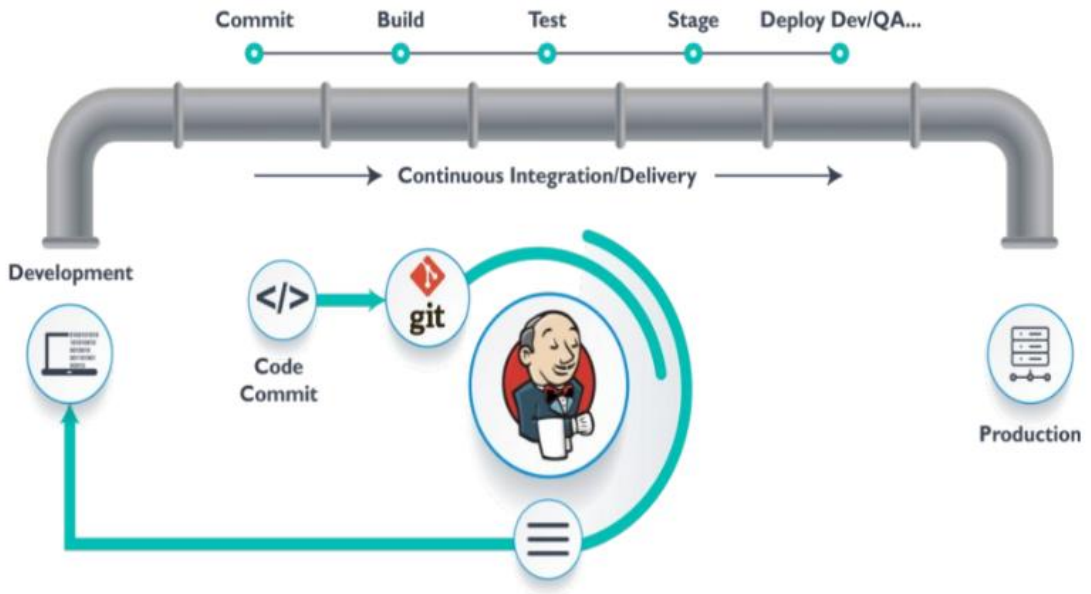
Pour les environnements de production ou staging, Kubernetes est utilisé avec Helm, qui facilite :

- Le déploiement déclaratif des microservices
- La gestion des mises à jour (`helm upgrade`)

- L’automatisation de la scalabilité, des redémarrages, et de la résilience

Chaque service est décrit via un **chart Helm**, qui peut être déployé via une commande dans le pipeline GitHub Actions.

Outil	Rôle Principal
GitHub Actions	Automatisation du pipeline CI/CD
Maven	Compilation, test et gestion des dépendances
Docker	Conteneurisation de chaque microservice
Docker Compose	Exécution multi-services en local
Kubernetes + Helm	Déploiement modulaire et scalable
GitHub Secrets	Gestion sécurisée des clés et identifiants sensibles



5. Description du pipeline CI/CD

Le pipeline CI/CD du projet **e-commerce-microservices-architecture** est entièrement automatisé via **GitHub Actions**, ce qui garantit une intégration fluide entre la gestion du code source, le build, les tests, la création d’images Docker, et le déploiement.

Le pipeline est défini dans un fichier YAML situé dans le répertoire `.github/workflows/`, typiquement nommé `build.yml` ou `ci-cd.yml`.

🔁 Déclencheur

Le pipeline est déclenché automatiquement lors des événements suivants :

- Push sur la branche principale (`main`)
- Création ou mise à jour d'une Pull Request ciblant la branche principale

Cela permet de s'assurer que seul du code validé passe dans la chaîne de production.

5. Description du pipeline CI/CD (adapté à ta configuration)

Le pipeline CI/CD est défini dans `.github/workflows/ci-cd.yml` et comporte deux jobs principaux, `build-test` et `deploy`.

Job 1 : `build-test`

- Utilise une **stratégie de matrice** (`matrix.service`) pour paralléliser la construction et le test de chaque microservice (`users`, `order`, `products`, `payment`, `api-gateway`, `config-server`, `eureka-server`).
- Pour chaque service :
  - Checkout du code
  - Setup de JDK 17 avec Temurin
  - Build Maven complet (`mvn clean install` au global et `mvn clean verify` spécifique par service)
  - Construction de l'image Docker en se basant sur un Dockerfile spécifique (`docker/${service}.Dockerfile`)
  - Connexion au registre GitHub Container Registry (GHCR) avec un token sécurisé
  - Push de l'image Docker taggée avec le SHA du commit

Cette approche permet une **construction modulaire, rapide et sécurisée**, avec un registre privé GitHub Container Registry.

Job 2 : `deploy`

- Ce job dépend du succès du job `build-test` (`needs: build-test`).
- Il crée un **cluster Kubernetes local avec Kind** (Kubernetes in Docker), ce qui permet de simuler un environnement de production dans GitHub Actions sans besoin d'un vrai cluster cloud.
- Il installe les outils `kubectl` et `helm` pour gérer Kubernetes et le déploiement des services.
- Il lance la commande Helm suivante :

```
bash
CopyEdit
helm upgrade --install shopify ./helm \
  --set global.imageTag=${{ github.sha }} \
  --set global.imageRepository="ghcr.io/${{ github.repository_owner }}"
```

Ce qui déploie ou met à jour la stack Kubernetes en injectant le tag d'image Docker correspondant au commit testé.

6. Déploiement (CD)

Le déploiement, ou **Continuous Deployment (CD)**, est la phase du pipeline CI/CD où les artefacts (ici les images Docker des microservices) sont envoyés dans un environnement d'exécution prêt à être utilisé.

□ Déploiement actuel dans le projet

Dans ta configuration, le déploiement est effectué via un job GitHub Actions qui :

1. **Crée un cluster Kubernetes local avec Kind**
   Kind (Kubernetes in Docker) permet de simuler un cluster Kubernetes complet dans l'environnement CI, sans besoin d'infrastructure cloud externe.
   C'est très pratique pour valider que le déploiement fonctionne correctement.
2. **Configure les outils Kubernetes**
   a. `kubectl` est installé pour gérer les ressources K8s
   b. `helm` est installé pour faciliter le déploiement et la gestion des charts

3. **Déploie la stack Kubernetes via Helm**
   La commande Helm déploie ou met à jour tous les microservices dans le cluster Kind, en injectant :
   a. Le **tag d'image Docker** correspondant au SHA GitHub (garantissant la traçabilité du code déployé)
   b. Le registre d'images GitHub Container Registry (GHCR)