CI/CD : Introduction et Fondamentaux

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-17

Table des matières

1 Contexte : L'évolution du développement logiciel 1
1.1 La problématique centrale 1
1.2 Architecture cible de ce cours 2
2 Rappels essentiels 2
2.1 Tests automatisés et pyramide des tests 2
2.2 Build Maven : cycle de vie et plugins 2
2.3 Conteneurisation avec Docker 3
2.4 Multi-stage builds 3
3 Concepts fondamentaux CI/CD 3
3.1 Qu'est-ce que la Livraison Continue (Continuous Delivery) ? 4
3.2 Qu'est-ce que le Déploiement Continu (Continuous Deployment) ? 4
3.3 Anatomie d'un Pipeline CI/CD 4
4 Qualité logicielle dans le pipeline 5
5 Principe du Feedback rapide : Fail Fast . . . 6
5.1 Les bénéfices de CI/CD 7
5.2 Actions concrètes dans le pipeline . . 7

1 Contexte : L'évolution du développement logiciel

Acronymes (premier usage) - **CI** = Continuous Integration (Intégration Continue) - **CD** = Continuous Delivery / Continuous Deployment (Livraison / Déploiement Continu) - **E2E** = End-to-End (tests bout-en-bout) - **IaC** = Infrastructure as Code (Infrastructure en tant que code) - **SAST** = Static Application Security Testing (analyse statique de sécurité) - **RBAC** = Role-Based Access Control (contrôle d'accès par rôle) - **K8s** = Kubernetes

Du développement isolé au DevOps moderne

- **Années 2000** : Développement en silos
  - Équipes dev vs ops séparées
  - Devs codent, ops déploient
  - Déploiements manuels, rares, risqués
  - « Ça marche sur ma machine » → problème en production
- **Problèmes identifiés** :
  - Déploiements longs et complexes
  - Bugs découverts tard (coût élevé)
  - Conflits d'intégration (« merge hell »)
  - Feedback lent sur la qualité du code
- **Besoin** : Automatisation et collaboration entre dev et ops

1.1 La problématique centrale

Comment passer du code sur ma machine à la production de manière fiable, rapide et répétable ?

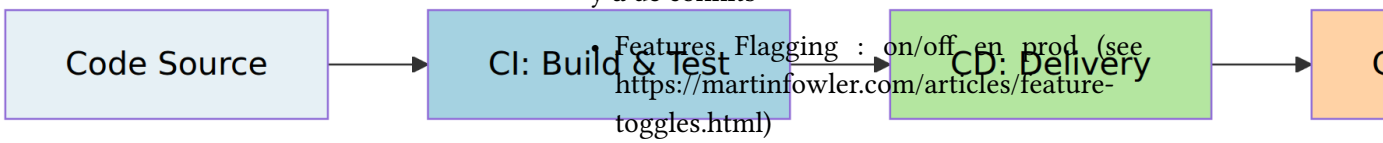
C'est LA question à laquelle CI/CD répond. Aujourd'hui, ce qui prenait des jours ou semaines peut prendre quelques minutes.

Attentes modernes :

- Déployer plusieurs fois par jour
- Revenir en arrière instantanément si problème

- Confiance totale dans les déploiements
- Zéro intervention manuelle

1.2 Architecture cible de ce cours



- Les deux approches nécessitent des tests automatisés solides

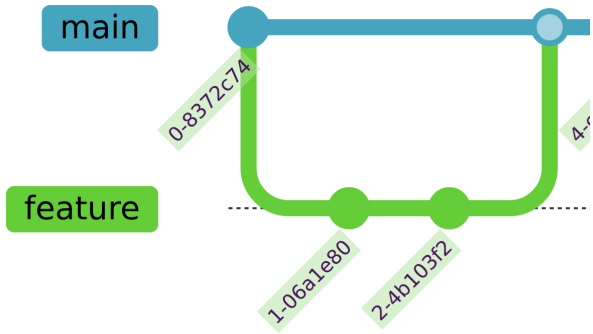
Rappel clé : Plus on intègre souvent, moins il y a de conflits

- Features Flagging : on/off en prod (see https://martinfowler.com/articles/feature-toggles.html)

2 Rappels essentiels

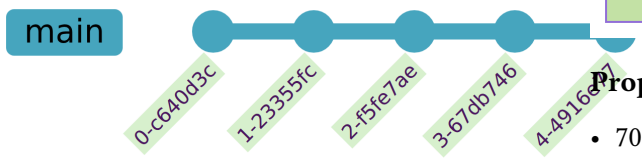
Git Workflows

Feature Branch Workflow



- Branches par fonctionnalité
- Pull/Merge Requests
- Code review avant merge

Trunk-Based Development

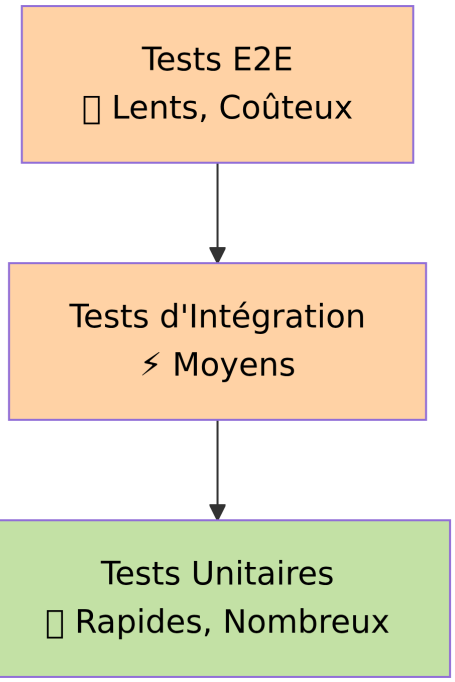


- Commits fréquents sur main
- Branches éphémères (<1 jour)
- Feature flags pour WIP

Pourquoi c'est important pour CI/CD ?

- **Feature Branch** : CI s'exécute sur chaque PR avant merge
- **Trunk-Based** : CI ultra-rapide nécessaire (builds < 10 min)

2.1 Tests automatisés et pyramide des tests



Proportions idéales

- 70% Unitaires
- 20% Intégration
- 10% E2E

Feedback rapide = CI efficace

2.2 Build Maven : cycle de vie et plugins

Cycle de vie par défaut

```
mvn clean      # Nettoie target/
mvn validate   # Vérifie le projet
mvn compile    # Compile sources
mvn test       # Tests unitaires
mvn package    # Crée JAR/WAR
mvn verify     # Tests intégration
mvn install    # Installe local repo
mvn deploy     # Déploie remote repo
```

Plugins essentiels pour CI

- `maven-compiler-plugin`
- `maven-surefire-plugin` (tests)
- `maven-failsafe-plugin` (IT)
- `jacoco-maven-plugin` (coverage)
- `maven-checkstyle-plugin`
- `dockerfile-maven-plugin`

2.3 Conteneurisation avec Docker

Pourquoi Docker en CI/CD ?

Reproductibilité

```
FROM openjdk:17-slim
COPY target/*.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Même environnement partout

Isolation

Chaque build dans un conteneur propre

Pas de pollution d'état

Portabilité

```
docker run -p 8080:8080 \
  mon-app:latest
```

Fonctionne partout

2.4 Multi-stage builds

Optimiser les images Docker pour la production

```
# Stage 1: Build
FROM maven:3.9-openjdk-17 AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package -DskipTests

# Stage 2: Runtime
FROM openjdk:17-slim
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

✅ Image finale légère

- Pas de Maven
- Pas de code source
- Seulement le JAR + JRE

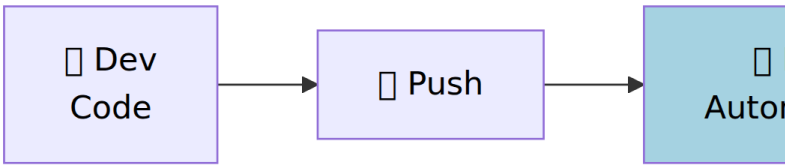
✅ Taille optimisée

- Build stage (with JDK) : ~600 MB
- Runtime stage (with JRE) : ~200 MB
- Quarto with native image : ~50 MB

3 Concepts fondamentaux CI/CD

Qu'est-ce que l'Intégration Continue (CI) ?

Continuous Integration : Intégrer fréquemment pour détecter les problèmes tôt



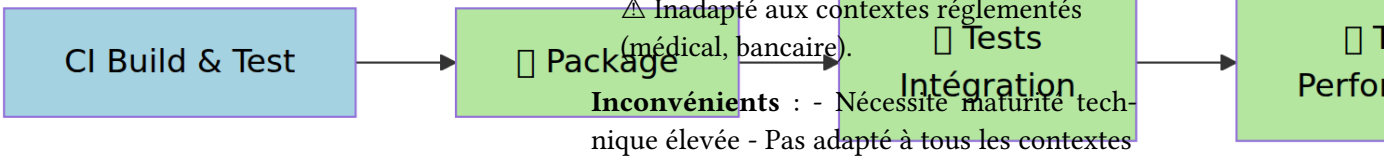
Principes clés

1. Commit fréquents (quotidiens)
2. Build automatique à chaque commit
3. Tests automatiques systématiques
4. Fix immédiat si échec
5. Environnement partagé

Définition : L'Intégration Continue (CI) est une pratique de développement où les développeurs intègrent fréquemment leur code dans un dépôt partagé, chaque intégration étant vérifiée par une build automatisée et des tests.

3.1 Qu'est-ce que la Livraison Continue (Continuous Delivery) ?

Continuous Delivery : Toujours prêt à déployer



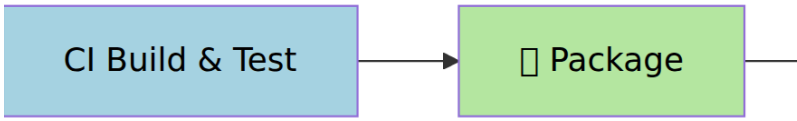
Caractéristiques

✅ Pipeline entièrement automatisé ✅ Chaque commit → artefact déployable ✅ Tests exhaustifs automatiques ⚠ **Déploiement manuel** (bouton) ✅ Déploiement à la demande

Définition : La Livraison Continue (Continuous Delivery) est une extension de CI où chaque changement qui passe les tests est automatiquement préparé pour être déployé en production. **Concept clé** : Le code est **toujours** dans un état déployable. On **peut** déployer à tout moment, mais on **choisit** quand le faire.

3.2 Qu'est-ce que le Déploiement Continu (Continuous Deployment) ?

Continuous Deployment : Zéro intervention humaine



Différence clé

✅ Pipeline 100% automatisé ✅ Chaque commit → production (si tests OK) 🤖 **Zéro intervention humaine** ⚡ Déploiements multiples/jour 🛡 Nécessite grande confiance tests

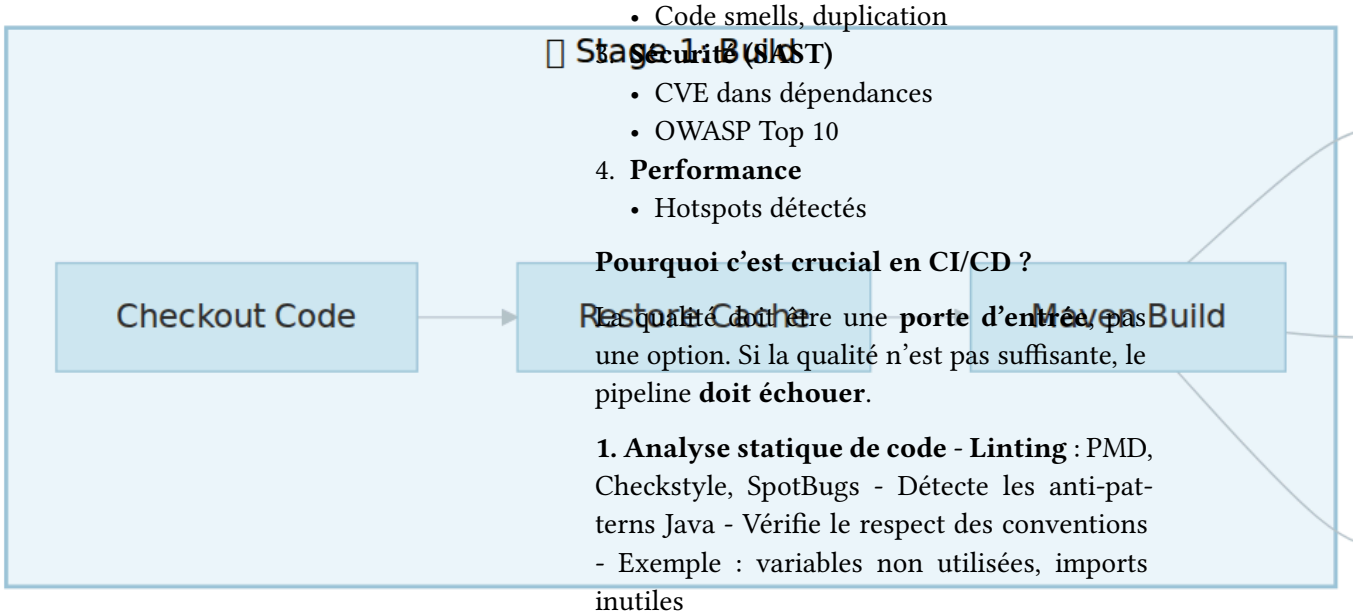
Pas d'étape manuelle ! Du commit au production : 100% automatique.

⚠ Inadapté aux contextes réglementés (médical, bancaire).

Inconvénients : - Nécessite maturité technique élevée - Pas adapté à tous les contextes (médical, bancaire avec validation réglementaire)

3.3 Anatomie d'un Pipeline CI/CD

Structure typique d'un pipeline moderne



4 Qualité logicielle dans le pipeline

La qualité ne se limite pas à la couverture de code

Métriques de qualité

```
Quality Gates:
  ✓ Couverture tests: >80%
  ✓ Code smells: <50
  ✓ Complexité: <15
  ✓ Duplication: <3%
  ✓ Vulnérabilités: 0 critiques
  ✓ Bugs: 0 bloquants
  ✓ Dette technique: <5 jours
```

Outils : SonarQube, CodeClimate

Dimensions de qualité

1. **Analyse statique**
   - Linting (PMD, Checkstyle)
   - Code style & conventions
2. **Maintenabilité**
   - Complexité cyclomatique
   - Code smells, duplication
3. **Sécurité (SAST)**
   - CVE dans dépendances
   - OWASP Top 10
4. **Performance**
   - Hotspots détectés

Pourquoi c'est crucial en CI/CD ?

La qualité doit être une **porte d'entrée**, pas une option. Si la qualité n'est pas suffisante, le pipeline **doit échouer**.

1. Analyse statique de code - Linting : PMD, Checkstyle, SpotBugs - Détecte les anti-patterns Java - Vérifie le respect des conventions - Exemple : variables non utilisées, imports inutiles

- **Code style** : Formatters (google-java-format)
  - Cohérence du codebase
  - Évite les débats de style en PR

2. Métriques de maintenabilité - Complexité cyclomatique : Nombre de chemins dans le code - Seuil recommandé : < 10 par méthode - Haut = difficile à tester et maintenir

- **Code smells** :
  - Méthodes trop longues (> 50 lignes)
  - Classes avec trop de responsabilités
  - Couplage excessif
- **Duplication** :
  - Code copié-collé = maintenance cauchemar
  - Seuil : < 3% du codebase
- **Dette technique** :
  - Temps estimé pour corriger tous les problèmes
  - Visualisation de la tendance (augmente ou diminue ?)

3. Sécurité (SAST - Static Application Security Testing) - Dependency Check : OWASP Dependency Check, Snyk - Scanne `pom.xml`

pour CVE connus - Base de données National Vulnerability Database

- **Code vulnerabilities** :
  - Injection SQL potentielle
  - Cross-Site Scripting (XSS)
  - Hard-coded secrets

4. Architecture & Design - **Metrics Maven Plugin** : - Couplage afférent/efférent - Instabilité des packages

- **ArchUnit** : Tests d'architecture as code
  - « Les controllers ne doivent pas appeler les repositories »
  - « Les entities ne doivent pas dépendre de services »

Exemple de Quality Gate SonarQube :

```
Quality Gate Failed si:
  - Couverture < 80%
  - New bugs > 0
  - Vulnerabilités critiques > 0
  - Code smells > 50
  - Duplication > 3%
  - Complexité cyclomatique > 15
```

Intégration dans le pipeline :

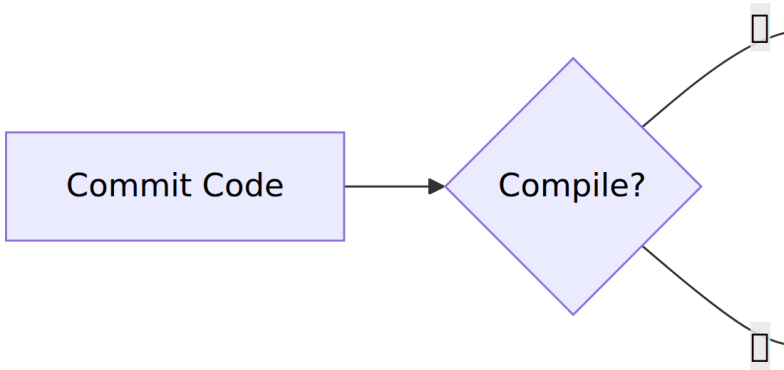
```
mvn          sonar:sonar          -
Dsonar.qualitygate.wait=true
# Échoue si Quality Gate not passed
```

Visualisation : - Badge sur le README - Dashboard d'équipe - Trend over time (la qualité s'améliore-t-elle ?)

Philosophie : « Shift Left » - détecter les problèmes le plus tôt possible, idéalement dans l'IDE, sinon dans le pipeline CI.

5 Principe du Feedback rapide : Fail Fast

Plus vite on détecte un problème, moins il coûte cher



Principe Fail Fast : Conception du pipeline pour détecter les problèmes le plus tôt possible.

Optimisations : - Paralléliser quand possible - Cacher les dépendances - Exécuter les tests les plus susceptibles d'échouer en premier - Utiliser des test selectors (tests modifiés d'abord)

Métrique : « Time to feedback »

Notifications : - Email/Slack sur échec - Badge de build sur README - Dashboard visible de l'équipe

5.1 Les bénéfices de CI/CD

Un pipeline CI/CD bien conçu augmente la vitesse, la qualité et la confiance dans les livraisons.

Vélocité

- Déploiements fréquents et prévisibles
- Réduction du Time-to-market

KPI : Deployment Frequency

Qualité

- Détection précoce des régressions
- Tests automatisés systématiques

KPI : Change Failure Rate

Confiance

- Rollback simple et traçabilité
- Déployable à tout instant

KPI : Lead Time & MTTR

DORA Metrics (essentiel)
- **Deployment Frequency** : fréquence des déploiements.
- **Lead Time for Changes** : temps commit → production.
- **Change Failure Rate** : % de déploiements causant incident.
- **Time to Restore Service (MTTR)** : temps moyen de rétablissement.

5.2 Actions concrètes dans le pipeline

Actions concrètes (intégrer au pipeline)
1. Exiger Quality Gate Sonar pour les PRs (ex.: couverture > 80%).
2. Automatiser tests unitaires + integration avant merge.
3. Mettre en place badges + dashboard DORA pour suivi continu.