2025

Déploiement CI/CD Tests de performance

Preuve de Concept système d'allocation de lits d'urgence

LOUIS ZEPHIR

Table des matières

Intégration et Déploiement Continus (CI/CD)

Le projet MedHead implémente un pipeline CI/CD via GitHub Actions, conformément au principe B3 (Intégration et livraison continues) des principes d'architecture du consortium.

Ce pipeline garantit la qualité du code et automatise les processus de build et de test.

Architecture du Pipeline

Le fichier de configuration du pipeline est situé dans `.github/workflows/ci-feature.yml`.

Il est déclenché automatiquement lors de chaque push ou pull request sur les branches de développement.

Étapes du Pipeline

Étape	Actions	Objectif
Build & Test	Configuration JDK 17, lancement PostgreSQL (Docker), compilation Maven, exécution tests unitaires (Surefire), tests d'intégration (Failsafe), rapport JaCoCo	Valider la compilation et l'exécution des tests automatisés
Code Quality	Analyse OWASP Dependency Check pour détecter les vulnérabilités dans les dépendances	Garantir la sécurité des dépendances (Principe B5 : Sécurité shift-left)
Build Artifact	Packaging du fichier JAR, upload en tant qu'artifact GitHub	Produire un livrable déployable et traçable

Déclencheurs du Pipeline

Le pipeline est déclenché automatiquement dans les cas suivants :

- Push sur les branches feature/* et develop
- Création d'une Pull Request vers develop ou main
- Déclenchement manuel via l'interface GitHub (workflow_dispatch)

Environnement de Test

L'environnement CI reproduit les conditions de production :

Composant	Configuration
Runtime Java	JDK 17 (Temurin)
Base de données	PostgreSQL 15 (service Docker)
Build Tool	Maven 3.x
Couverture de code	JaCoCo avec rapport XML/HTML

Rapports et Métriques

Chaque exécution du pipeline génère les rapports suivants, consultables dans l'onglet Actions de GitHub :

- Rapport de tests unitaires : target/surefire-reports/
- Rapport de tests d'intégration : target/failsafe-reports/
- Rapport de couverture JaCoCo : target/site/jacoco/
- Rapport OWASP Dependency Check : target/dependency-check-report.html

Conformité aux Principes d'Architecture

Principe	Mise en œuvre dans le Pipeline
B3 : Intégration continue	Pipeline automatisé déclenché à chaque commit, feedback rapide aux développeurs
B4 : Tests automatisés	Exécution systématique des tests unitaires et d'intégration, rapport de couverture
B5 : Sécurité shift-left	Analyse OWASP des vulnérabilités dès le build, avant déploiement

Gestion de Versionnage

Le projet utilise Git comme système de contrôle de version, hébergé sur GitHub.

La stratégie de versionnage suit le modèle Git Flow adapté, permettant un développement parallèle tout en maintenant la stabilité du code de production.

Stratégie de Branches

Branche	Rôle	Protection
main	Code de production stable, déployable à tout moment	Protégée, merge via PR uniquement
develop	Branche d'intégration, regroupe les features terminées	Protégée, merge via PR
feature/*	Développement de nouvelles fonctionnalités	Non protégée
hotfix/*	Corrections urgentes en production	Non protégée
release/*	Préparation des versions	Non protégée

Convention de Nommage des Branches

Les branches suivent une convention de nommage structurée pour faciliter la traçabilité :

- **`feature/`** : nouvelles fonctionnalités (ex: `feature/poc-bed-allocation-setup`)
- **`bugfix/`** : corrections de bugs (ex: `bugfix/fix-distance-calculation`)
- **`hotfix/`** : corrections urgentes (ex: `hotfix/security-patch`)
- **`release/`** : préparation de version (ex: `release/v1.0.0`)

Convention de Commits

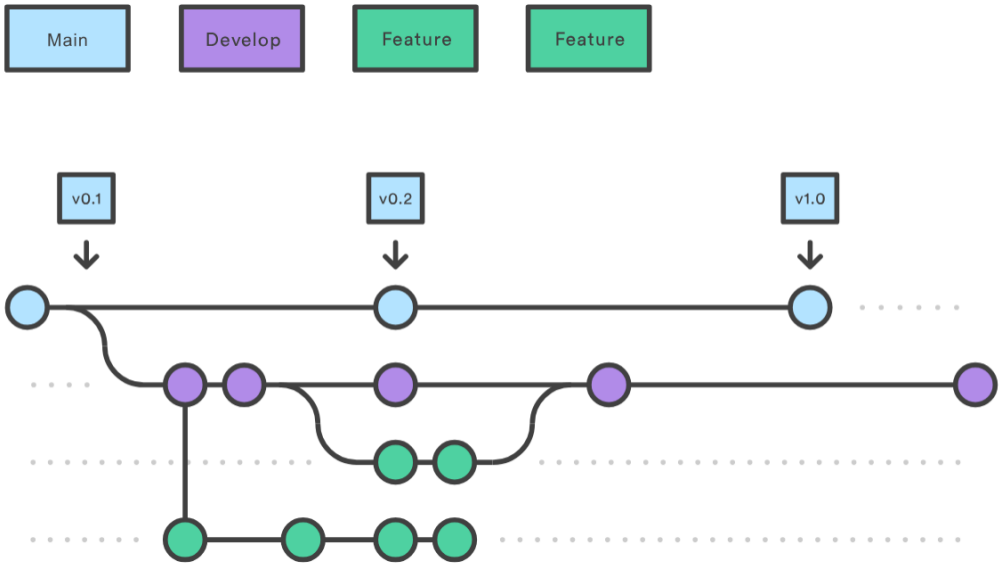
Les messages de commit suivent la convention Conventional Commits pour une meilleure lisibilité de l'historique :

Type	Description	Exemple
feat	Nouvelle fonctionnalité	feat: add nearest hospital endpoint
fix	Correction de bug	fix: correct distance calculation
test	Ajout ou modification de tests	test: add integration tests for API
docs	Documentation	docs: update README
refactor	Refactoring sans changement fonctionnel	refactor: extract service layer
ci	Modification du pipeline CI/CD	ci: add JaCoCo coverage

Workflow de Développement (GitFlow)

Le cycle de développement d'une fonctionnalité suit les étapes suivantes inspiré de la méthode GitFlow :

1. Création de la branche `feature/*` depuis `develop`
2. Développement avec commits réguliers suivant la convention
3. Push sur le repository distant (déclenche le pipeline CI)
4. Vérification du succès du pipeline (tests verts, couverture OK)
5. Création d'une Pull Request vers `develop`
6. Revue de code par un pair (si applicable)
7. Merge de la PR après validation



Cliquez sur le lien pour voir le principe : https://www.atlassian.com/fr/git/tutorials/comparing-workflows/gitflow-workflow

Gestion des Tags et Versions

Le versionnage sémantique (SemVer) est utilisé pour les releases :

- Format : **`vMAJOR.MINOR.PATCH`** (ex: v1.0.0, v1.1.0, v1.1.1)
- **MAJOR** : changements incompatibles avec les versions précédentes
- **MINOR** : nouvelles fonctionnalités rétrocompatibles
- **PATCH** : corrections de bugs rétrocompatibles

Tests de Performance (Stress Tests)

Conformément aux exigences du projet MedHead, des tests de stress ont été réalisés pour valider que le système d'intervention d'urgence respecte les contraintes de temps de réponse.

L'exigence stipule que le temps moyen de traitement d'une urgence doit être inférieur à **800 millisecondes**.

Outil et Méthodologie

Les tests de performance ont été réalisés avec **Apache JMeter 5.6.3**, un outil de référence pour les tests de charge et de stress des applications web.

Endpoint testé : `GET /api/hospitals/search/nearest`

Configuration du Test

Paramètre	Valeur
Nombre total de requêtes	5 000 requêtes
Utilisateurs simultanés	50 threads
Période de montée en charge (ramp-up)	10 secondes
Paramètres de requête	latitude, longitude (aléatoires), specialtyCode
Authentification	Token JWT

Résultats des Tests

Les résultats ci-dessous concernent l'endpoint critique de recherche d'hôpital le plus proche :

Métrique	Résultat	Objectif	Statut
Temps de réponse moyen	89 ms	< 800 ms	✅ CONFORME
Temps de réponse médian	17 ms	< 800 ms	✅ CONFORME
90ème percentile (P90)	301 ms	< 800 ms	✅ CONFORME
95ème percentile (P95)	413 ms	< 800 ms	✅ CONFORME
99ème percentile (P99)	660 ms	< 800 ms	✅ CONFORME
Temps minimum	8 ms	-	-
Temps maximum	1 560 ms	-	⚠️ Pic isolé
Taux d'erreur	0.00%	0%	✅ CONFORME
Débit (Throughput)	49.6 req/sec	-	✅ Bon débit

Voir le rapport

Analyse des Résultats

Points positifs :

- Le temps de réponse moyen (89ms) est très inférieur à l'objectif de 800ms, avec une marge de 711ms
- 99% des requêtes sont traitées en moins de 660ms, garantissant une excellente expérience utilisateur
- Le taux d'erreur de 0% démontre la stabilité du système sous charge
- Le débit de 49.6 requêtes/seconde est satisfaisant pour un environnement de développement

Point d'attention :

- Le temps maximum de 1560ms représente un pic isolé, probablement lié à l'initialisation du pool de connexions ou au garbage collector. Ce cas reste marginal et n'impacte pas les percentiles.

Conclusion

✅ L'exigence de performance est VALIDÉE

Le système d'allocation de lits d'urgence répond aux exigences de performance du projet MedHead.

L'endpoint de recherche de l'hôpital le plus proche traite les requêtes en **89ms en moyenne**, soit **9 fois plus rapide** que l'objectif fixé de 800ms.

Cette marge confortable garantit que le système pourra absorber une montée en charge en environnement de production.

Recommandations pour la Production

- Mettre en place un monitoring continu des temps de réponse (Prometheus/Grafana)
- Configurer des alertes si le P95 dépasse 600ms
- Envisager un cache Redis pour les requêtes fréquentes
- Réaliser des tests de charge réguliers après chaque release majeure