Module : Suivi de la qualité logicielle assisté par l'IA

Partie 3 — Tests de Performance, Dette Technique et CI/CD

Prérequis : Partie 2 — IA et génération de tests

Outils : solutions open source

Projet support : Python / FastAPI

Table des matières

1 Introduction Générale

Ce troisième jour est consacré à l'étude de la **qualité logicielle avancée**, abordée sous l'angle de trois piliers essentiels :

— **La dette technique** : comprendre son impact, mesurer son évolution et identifier les leviers de réduction.

— **Les tests de performance** : analyser la résistance d'un système sous charge, et anticiper les dégradations de service.

— **Le pipeline CI/CD** : automatiser les tests, l'analyse qualité et le déploiement continu du logiciel.

L'IA occupe une place centrale : elle devient assistante dans l'analyse, l'interprétation des métriques, la génération de tests et la construction du pipeline de livraison continue.

Objectifs du jour

— Comprendre, mesurer et interpréter la dette technique.

— Découvrir et utiliser des outils légers d'analyse (flake8, radon, pylint).

— Réaliser des tests de performance avec JMeter et k6.

— Construire un pipeline CI/CD intégrant qualité et tests.

— Utiliser l'IA pour interpréter, corriger et automatiser.

2 Dette Technique

2.1 Définition et nature du problème

La dette technique désigne le **coût futur** généré par des compromis à court terme lors du développement d'un logiciel. Elle peut être intentionnelle ou accidentelle.

Analogie

Développer vite revient à faire un *emprunt*. Le remboursement correspond au temps passé à corriger des erreurs, restructurer du code et stabiliser le système.

2.2 Sources principales de dette

— Duplication de logique

— Absence de documentation ou tests

— Conception précipitée

— Nommage peu explicite

— Fonctions trop longues ou trop complexes

2.3 Impacts majeurs

— Augmentation du nombre de bugs

— Lenteur dans les évolutions

— Fragilité accrue du système

— Coûts de maintenance plus élevés

2.4 Indicateurs de dette technique

Les métriques suivantes constituent les signaux d'alerte les plus courants :

— **Complexité cyclomatique** : mesure du nombre de chemins indépendants dans une fonction.

— **Taux de duplication** : proportion de code répété.

— **Code smells** : éléments suspects (variables inutilisées, fonctions trop longues, classes surchargées).

— **Lignes de code par fonction** : un indicateur simple mais puissant.

Conseil de bonne pratique

Il est souvent moins coûteux de limiter la dette technique à mesure qu'elle apparaît plutôt que d'attendre la fin du projet pour la traiter.

3 Suivi Simplifié de la Dette Technique

Même sans SonarQube, il est possible d'obtenir une vision claire de la qualité du code.

3.1 Outils légers

— **flake8** : détecte violations PEP8, erreurs de style et incohérences.

— **pylint** : analyse poussée, notation générale du code.

— **radon** : mesures de complexité cyclomatique.

— **eslint** (JavaScript) et **phpstan** (PHP) pour d'autres stacks.

3.2 Exemple d'installation et d'exécution

```
pip install flake8 radon pylint

flake8 app/
radon cc app/ -a
pylint app/
```

3.3 Rôle de l'IA dans l'interprétation

Prompt IA recommandé

"Voici les résultats de flake8/pylint/radon. Analyse les indicateurs, détecte les risques principaux et propose un plan de refactoring en trois priorités."

4 Tests de Performance

4.1 Objectifs

Les tests de charge permettent d'anticiper :

— saturation du système,

— dégradations de temps de réponse,

— erreurs sous volume,

— goulets d'étranglement.

4.2 Indicateurs clés

— **Latence moyenne**

— **Temps de réponse maximal**

— **Taux d'erreurs**

— **Throughput** (requêtes/seconde)

4.3 Outils

— **Apache JMeter** : interface graphique complète.

— **k6** : outil CLI rapide, idéal pour automatisation.

4.4 Rôle de l'IA

— générer des scénarios de charge réalistes ;

— interpréter des tableaux ou graphiques de performance ;

— proposer des seuils de tolérance ;

— identifier les goulets d'étranglement.

Prompt IA

"Voici les résultats d'un test de charge (latence, throughput, erreurs). Analyse les faiblesses et propose des seuils cibles pour une API critique."

5 Automatisation CI/CD

5.1 Définition d'un pipeline

Un pipeline CI/CD classique comporte les étapes suivantes :

1. Build
2. Tests (unitaires, intégration)
3. Analyse statique et qualité
4. Tests de charge (optionnel mais recommandé)
5. Déploiement

5.2 Outils

— **GitHub Actions**

— **GitLab CI**

5.3 Apports de l'IA

— générer les fichiers YAML complets ;

— proposer les dépendances et versions ;

— optimiser les règles de déclenchement ;

— intégrer automatiquement tests qualité + tests de charge.

Note pédagogique

Le pipeline devient un **contrôle automatique de la qualité**. Plus il est complet, moins la dette technique s'accumule.

6 Travaux Pratiques

6.1 TP 3.1 — Mesurer la dette technique

Objectif

Installer des outils d'analyse statique, interpréter les résultats et construire un plan de réduction de dette.

Étapes :

1. Installer flake8, radon et pylint
2. Collecter les scores principaux
3. Demander à l'IA un plan de refactoring priorisé

4. Comparer avec une capture SonarQube

Livrables :

— Tableau d'indicateurs avant/après refactoring

— Rapport IA (plan d'action)

6.2 TP 3.2 — Test de charge avec JMeter ou k6

Objectif

Exécuter un test de charge simple, analyser les résultats et interpréter les métriques avec l'aide de l'IA.

Étapes :

1. Choisir un outil :
   - **JMeter** : test HTTP via interface graphique.
   - **k6** : script JavaScript exécuté en CLI.
2. Préparer un test minimal :

```
# Exemple k6 : test.js
k6 run test.js
```

```
import http from 'k6/http';
import { sleep } from 'k6';

export default function () {
  http.get('http://localhost:8000/items');
  sleep(1);
}
```

3. Exécuter et collecter les métriques :
   - Latence moyenne
   - Taux d'erreur
   - Requêtes/seconde
   - Temps de réponse max
4. Interpréter les résultats avec l'IA :

Prompt IA

"Voici les résultats d'un test de charge. Analyse les goulets d'étranglement, les seuils critiques et propose des améliorations réalistes."

Livrables :

- Capture graphique (JMeter ou sortie k6)
- Rapport IA de synthèse

6.3 TP 3.3 — Pipeline CI/CD guidé par IA

Objectif

Créer un pipeline GitHub Actions intégrant linters, tests, analyse qualité et tests de charge.

Pipeline minimal fourni :

```yaml
name: QualityPipeline
on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'
      - name: Run linters
        run: flake8 app/
      - name: Run tests
        run: pytest --maxfail=1 --disable-warnings -q
```

Étapes :

1. Exécuter le pipeline fourni.
2. Demander à l'IA des améliorations :

Prompt IA amélioré

"Optimise ce pipeline CI/CD pour inclure : - analyse de dette technique (radon, pylint), - exécution d'un test de charge k6, - upload des résultats en artefacts, - conditions de déclenchement intelligentes."

3. Intégrer la version finale générée par l'IA.
4. Vérifier l'exécution sur GitHub.

Livrables :

— Pipeline YAML fonctionnel

— Capture du workflow exécuté dans GitHub Actions

7 Livrables du Jour 3

Objectif	Livrable IA	Format
Dette technique	Analyse + plan d'action	Markdown / PDF
Tests de charge	Interprétation + recommandations	Texte + Capture
CI/CD	Pipeline complet fonctionnel	YAML + capture GitHub

8 Résumé du Jour 3

Aujourd'hui, vous avez appris à combiner analyse qualité, performance et automatisation pour construire un système logiciel robuste et maintenable.

— La dette technique se mesure, se traite et se suit grâce à des outils légers comme flake8, pylint et radon.

— Les tests de charge sont essentiels pour valider la scalabilité d'une API.

— Un pipeline CI/CD bien structuré garantit une qualité continue.

— L'IA devient un assistant précieux pour accélérer l'analyse, écrire des scripts ou générer des pipelines.

Bonne pratique essentielle

Un pipeline CI/CD doit évoluer en même temps que le projet : adaptez les tests, surveillez la dette et intégrez les retours utilisateurs dans les seuils de performance.

9 Annexes

9.1 Ressources Utiles

— Documentation JMeter : `https://jmeter.apache.org/`

— Documentation k6 : `https://k6.io/docs/`

— GitHub Actions : `https://docs.github.com/actions`

— Pylint : `https://pylint.pycqa.org/`

— Radon : `https://radon.readthedocs.io/`

9.2 Installation rapide

```
# Install tools for debt analysis
pip install flake8 pylint radon

# k6 installation
brew install k6   # MacOS
```

```
sudo apt install k6   # Linux

# Run a simple load test (k6)
k6 run test.js
```