Tweetoscope Project

Bilal Kopp, Nolan Mulder, Alexis Garcias

20 novembre 2025

1 Introduction

Le projet consiste à refactoriser une application de traitement de tweets en une architecture distribuée basée sur Kafka. L'objectif est de transformer le système initial, construit sur un pattern observer monolithique, en un pipeline de microservices communiquant via des topics Kafka, puis de déployer l'ensemble avec Docker et Kubernetes, orchestré par un pipeline CI/CD GitLab.

Le système utilise des tweets enregistrés (miniTestBase.txt, scenarioTestBase.txt, largeTestBase.txt) plutôt que l'API Twitter en direct, ce qui permet un testing reproductible et stable.

2 Secrets et Git

Il ne faut jamais commit de tokens ou d'identifiants sensibles dans un dépôt Git. Les secrets (tokens Docker, registry, potentiels accès API) doivent être stockés exclusivement dans les variables d'environnement GitLab CI/CD.

Cette approche assure une bonne séparation entre code et secrets, limite les risques de fuite, et permet d'injecter les credentials uniquement au runtime dans les jobs concernés.

3 Lecture des tweets et nouveaux filtres

L'implémentation de `MockTwitterStreamRecorded` permet de lire des tweets depuis les fichiers fournis. Les trois bases (`miniTestBase`, `scenarioTestBase`, `largeTestBase`) ont été utilisées pour valider progressivement les différents composants.

Un filtre custom, `RecentTweetFilter`, a été ajouté pour ne conserver que les tweets récents selon une fenêtre temporelle configurable. Un test unitaire JUnit valide son comportement et vérifie que seuls les tweets dans la plage temporelle sont propagés.

4 Architecture Kafka

4.1 Choix d'architecture

L'architecture Kafka repose sur quatre topics principaux :

— `tweets-raw` : tweets bruts provenant du producer

— `tweets-filtered` : tweets passés par le filtre
— `hashtags` : hashtags extraits
— `leaderboard` : top hashtags calculés

Le pipeline est composé de cinq microservices : Producer, Filter, Extractor, Counter et Visualizer. Chaque composant est indépendant et peut être déployé ou mis à l'échelle séparément, ce qui simplifie le debugging et la maintenance.

4.2 Analyse de risques

— **Producer down** : plus de nouveaux tweets, mais le reste du pipeline traite les messages déjà présents.
— **Filter down** : les tweets s'accumulent dans `tweets-raw`.
— **Extractor down** : les tweets filtrés s'accumulent dans `tweets-filtered`.
— **Counter down** : il reconstruit son état en relisant `hashtags`.
— **Visualizer down** : le leaderboard continue d'être mis à jour mais n'est plus affiché.

4.3 Implémentation et tests

Chaque stage a été vérifié individuellement via un console consumer Kafka et la base `miniTestBase.txt`. Les formats JSON, offsets, et consumer groups se comportent comme attendu. Kafka gère naturellement le scale-out : plusieurs instances d'un même service se répartissent les partitions.

5 Dockerisation

Un Dockerfile simple a été créé pour chaque service (`producer`, `filter`, `extractor`, `counter`, `visualizer`). Le build Maven est inclus dans chaque image, puis les services sont packagés dans une image runtime légère basée sur `eclipse-temurin`.

Les tests locaux ont été réalisés avec `docker-compose`, permettant de lancer rapidement Kafka et l'ensemble des microservices. Cela a permis de valider l'architecture avant le passage à Kubernetes.

6 Déploiement Kubernetes

6.1 Déploiement simple

Le déploiement a été réalisé dans le namespace `student30-ns`. Chaque microservice est décrit via un Deployment avec un replica par défaut (plusieurs replicas utilisés ponctuellement pour expérimenter le scaling).

Les images sont push par la CI dans le GitLab Container Registry, et les manifests Kubernetes utilisent directement ces images (`tweetoscope-<service>:latest`). Les variables d'environnement nécessaires (ex. `KAFKA_BOOTSTRAP_SERVERS`) sont définies dans les Deployments.

6.2 Fault Tolerance

Les tests de fault tolerance montrent que Kubernetes redémarre automatiquement les pods en échec. Grâce à Kafka, aucun message n'est perdu : les consumers rattrapent le retard en consommant les messages accumulés pendant la panne. Le Counter reconstruit correctement son état à partir du topic `hashtags`.

6.3 Mitigation via Kubernetes

Kubernetes ajoute un niveau supplémentaire de résilience grâce aux Deployments, ReplicaSets, restart policies et au découplage complet entre les services.

7 CI/CD Pipeline

7.1 LaTeX build

Le job `build_report` compile le rapport via `latexmk`. Le job `pages` publie automatiquement le PDF sur GitLab Pages. Les artifacts sont conservés une semaine.

7.2 Tests

Le job `test-recent-filter` exécute les tests JUnit du filtre custom. Les rapports JUnit sont collectés et visibles directement dans GitLab.

7.3 Docker build et push

Le job `docker-build-images` build les images des cinq services et les push dans le GitLab Container Registry de CentraleSupélec. Les images taggées `latest` sont ensuite utilisées dans les manifests Kubernetes.

7.4 Code Quality

Un job Checkstyle a été ajouté au pipeline. Il génère un rapport XML stocké comme artifact. Le pipeline ne fail pas en cas de violations, mais cela permet d'améliorer progressivement la qualité du code.

8 Vidéo de démonstration

La vidéo montre :
— l'exécution du pipeline CI/CD après un commit,
— le build et le push des images Docker,
— l'application manuelle des manifests sur Kubernetes,
— le flow complet du Producer jusqu'au Visualizer,
— un test de fault tolerance (arrêt d'un pod puis rattrapage),
— un redeploy après modification du code.

9 Conclusion

Ce projet a permis de travailler sur une architecture distribuée complète : Kafka, microservices, Docker, Kubernetes, et CI/CD. La séparation en services indépendants facilite le développement, la maintenance, et les tests.

La partie la plus difficile a clairement été la création des manifests Kubernetes. Ayant raté le premier TD pour des raisons techniques, j'ai dû apprendre en autonomie et réécrire plusieurs fois mes fichiers. L'approche qui a finalement fonctionné a été de partir de pods simples, puis de monter progressivement vers des Deployments et Services plus complexes.

Les outils d'IA (ChatGPT, Cursor) ont été utiles pour générer du boilerplate, diagnostiquer des erreurs, et accélérer certains aspects du développement. Cependant, chaque réponse devait être adaptée manuellement pour s'intégrer dans l'architecture réelle. L'IA a été un accélérateur, pas un remplaçant : la compréhension du système est restée essentielle.

Sur des projets classiques, avec des technologies bien établies comme ici, les outils d'IA sont réellement efficaces pour gagner du temps. Malgré cela, je continue de penser que coder soi-même reste plus satisfaisant et formateur. C'est souvent en écrivant le code "à la main" que j'ai le plus appris et que je suis resté engagé dans le projet.