Tweetoscope

Projet Ingénierie des applications logicielles

CentraleSupélec

Hugo De Bosschere
Valentin Marchesini
Pierre Giraud

Encadrants : Virginie Galtier et Michel Ianotto

Année universitaire 2025–2026

Contents

Introduction

Dans le cadre du cours d'Ingénierie des applications logicielles, nous avons développé Tweetoscope, une application répartie qui collecte, filtre, transforme et analyse en temps réel un flux de tweets. L'objectif du projet est de mettre en œuvre, sur un cas concret, plusieurs notions vues en cours : architecture orientée flux, systèmes distribués, Apache Kafka, Docker, Kubernetes et CI/CD.

L'application fournie au départ est une application Java monolithique qui lit des tweets (réels ou simulés), applique un filtre, extrait les hashtags, les compte et affiche un classement des plus fréquents. Cette version initiale fonctionne, mais elle ne passe pas à l'échelle et n'est pas tolérante aux pannes. Le projet consiste à refactorer cette application en pipeline distribuée basée sur Kafka, puis à la containeriser pour simplifier la gestion des dépendances et à industrialiser son déploiement et gérer les pannes via kubernetes.

Dans ce rapport, nous revenons d'abord sur le fonctionnement de l'application d'origine, puis nous détaillons les tâches demandées dans l'énoncé, en expliquant à chaque fois ce que nous avons effectivement mis en place. Les dernières sections décrivent les choix d'architecture, les expérimentations de tolérance aux pannes, la mise en place de la CI/CD, ainsi que notre retour d'expérience, y compris sur l'utilisation d'outils d'intelligence artificielle.

La vidéo explicative est disponible ici : `https://filesender.renater.fr/?s=download&token=ed1dc72e-7129-4dde-8ffd-cbf20f741c4e`.

1 Présentation de l'application de départ

1.1 Pipeline initial

Le code de départ est organisé autour de la classe `TweetoscopeApp`, qui crée et connecte plusieurs composants via le patron observateur (interfaces `Publisher` et `Subscriber`). Le pipeline principal est le suivant :

- un producteur de tweets (`tweetsProducer`) qui génère ou lit les tweets
- un filtre de tweets (`TweetFilter`) qui ne conserve que les messages répondant à certains critères
- un extracteur de hashtags (`HashtagExtractor`)
- un compteur de hashtags (`HashtagCounter`) qui maintient un classement
- un visualiseur (`Visualizor`) qui affiche le top des hashtags sous forme d'histogramme

Le point d'entrée `TweetoscopeApp.main` lit trois arguments :

1. la source des tweets, par exemple `random`, `scenario` ou, dans les anciennes versions, un flux Twitter en direct
2. la stratégie de filtrage (`none`, `language`, etc.)
3. la taille du tableau des meilleurs hashtags.

Une fois ces paramètres lus, l'application instancie les différents composants, les relie entre eux, puis laisse tourner la boucle de traitement.

1.2 Producteurs de tweets

Plusieurs implémentations de producteurs sont fournies :

- `MockTwitterStreamRandom`, qui génère un flux infini de tweets aléatoires en combinant des hashtags prédéfinis
- `MockTwitterStreamScenario`, qui rejoue un petit scénario de dix tweets, utile pour des tests reproductibles
- des producteurs liés à l'API Twitter réelle, marqués comme dépréciés dans l'énoncé, que nous n'avons pas utilisés

Tous ces producteurs implémentent l'interface `java.util.concurrent.Flow.Publisher` afin de pouvoir notifier leurs abonnés à chaque réception ou génération d'un tweet.

1.3 Filtre, extraction et comptage

Le composant `TweetFilter` s'abonne au producteur et décide, pour chaque tweet, s'il doit être transmis ou non. Dans le mode `FILTER_LANGUAGE`, il ne garde que les tweets dont la langue détectée (`lang`) est `"en"`.

Le composant `HashtagExtractor` s'abonne au filtre et utilise la classe `com.twitter.twittertext.Extractor` pour récupérer les hashtags présents dans le texte du tweet. Chaque hashtag devient un événement transmis au composant suivant.

Le composant `HashtagCounter` maintient alors une structure de données de type liste ou dictionnaire associant à chaque hashtag le nombre de fois où il est apparu. Après chaque mise à jour, il calcule le top des hashtags les plus fréquents et notifie le `Visualizor` si ce top a changé.

Enfin, le `Visualizor` affiche le classement sous forme de barres, mises à jour à chaque modification.

1.4 Limites de cette version

Cette première version fonctionne bien pour des volumes modestes, mais elle présente plusieurs limites :

- couplage fort entre les composants, qui rend difficile la mise à l'échelle
- absence de persistance des messages entre services
- manque de tolérance aux pannes : si un composant s'arrête, tout le pipeline est perturbé
- difficulté à répartir la charge sur plusieurs machines

Les tâches suivantes se concentrent justement sur ces aspects : séparation en services, introduction de Kafka, tests, puis containerisation et déploiement.

2 Task 5 : étude du code et gestion des secrets

2.1 Compréhension du code fourni

La première étape a consisté à cloner le dépôt fourni, à ouvrir le projet dans l'environnement de développement, puis à exécuter l'application dans sa configuration d'origine. Nous avons commencé par lancer la version la plus simple :

```
mvn exec:java \
  -Dexec.mainClass="tweetoscope.TweetoscopeApp" \
  -Dexec.args="scenario none 5"
```

Cette commande lance le scénario prédéfini de dix tweets sans appliquer de filtrage particulier, et affiche un top 5 des hashtags. Nous avons ensuite testé la source `random` ainsi que le filtrage par langue en utilisant `language` comme second argument.

Ces premiers tests nous ont permis de vérifier que le pipeline initial fonctionnait bien et de mieux comprendre la manière dont les différents composants interagissent. Cela a aussi servi de point de comparaison pour les étapes suivantes, notamment lorsque nous avons introduit Kafka.

2.2 Secrets and Git

L'énoncé pose explicitement la question de la gestion des secrets, en particulier du jeton d'accès à l'API Twitter. Nous avons étudié ce point et abouti aux conclusions suivantes.

L'intégration directe des secrets (clé API, jeton d'accès) dans un dépôt Git expose l'application à plusieurs risques majeurs.

Risques liés aux secrets dans Git

- En cas de dépôt public, toute personne ayant accès au dépôt peut récupérer le secret et l'utiliser pour effectuer des appels vers le service concerné. Dans le cas de Twitter, cela permettrait de consommer le quota de requêtes d'un autre, voire de publier du contenu en son nom.
- Même si le fichier contenant la clé est supprimé dans un commit ultérieur, Git conserve l'historique complet. Un attaquant peut donc remonter dans les versions précédentes pour retrouver le secret.
- Même pour un dépôt privé, un problème de droits mal configurés ou un compte compromis peut conduire à une fuite des secrets.

Effacer un secret déjà commité demande de réécrire l'historique (par exemple avec `git filter-branch`), ce qui est délicat à réaliser sur un projet partagé.

Bonnes pratiques retenues

Pour éviter ces risques, nous avons adopté plusieurs bonnes pratiques :

- Le jeton d'accès n'apparaît jamais dans le code source ni dans les fichiers suivis par Git. L'application le lit à partir d'une variable d'environnement (par exemple `BEARER_TOKEN`).
- Lors du développement local, les variables d'environnement sont définies dans le système ou via des fichiers de configuration non suivis par Git. Ces fichiers sont ajoutés dans `.gitignore`.
- Les exemples de configuration fournis dans le dépôt contiennent des valeurs factices ou des variables symboliques, mais jamais de clé réelle.

Conclusion sur les secrets

En résumé, nous avons considéré que stocker un jeton ou une clé d'accès dans un dépôt Git est une mauvaise pratique de sécurité. Les secrets doivent rester à l'extérieur du code versionné et être injectés dans l'application au moment de l'exécution, via des mécanismes adaptés (variables d'environnement, gestionnaire de secrets, configuration du cluster, etc.).

3 Task 6 : utilisation de tweets enregistrés

3.1 Objectif de la tâche

L'objectif de la Task 6 est de s'affranchir de la variabilité d'un flux Twitter en temps réel en utilisant un jeu de données enregistré. Cela permet de tester l'application de manière reproductible, et c'est aussi la seule option réaliste sans accès payant à l'API.

L'énoncé fournit trois fichiers :

- `scenarioTestBase.txt`, qui contient les dix tweets du scénario utilisé par `MockTwitterStreamScenario`
- `miniTestBase.txt`, qui regroupe neuf tweets réels
- `largeTestBase.txt`, qui contient cinq cent mille tweets et sert à des tests plus lourds

3.2 Classe MockTwitterStreamRecorded

L'énoncé demande d'écrire une classe `MockTwitterStreamRecorded` capable de lire l'un de ces fichiers et de publier les tweets sous la forme attendue par le reste de l'application.

Dans notre projet, la classe correspondante était déjà présente dans la base de code fournie. Nous nous sommes donc surtout concentrés sur sa compréhension et sur la manière de l'utiliser.

Le principe est le suivant :

- le constructeur reçoit le nom du fichier à lire
- les lignes du fichier sont parcourues et transformées en objets `Tweet`
- chaque tweet est publié vers les abonnés, comme le ferait un producteur connecté à l'API réelle

Nous avons vérifié que cette classe permettait d'alimenter le pipeline avec les différents fichiers de test, en particulier `largeTestBase.txt`.

3.3 Validation du pipeline et compteur de tweets

Pour nous assurer que l'ensemble du pipeline traitait bien toutes les données, nous avons ajouté un compteur de tweets dans le visualiseur. Cette modification a été réalisée dans la classe `Visualizor.java` : à chaque mise à jour du classement, nous incrémentons un compteur interne et nous affichons le nombre total de tweets pris en compte jusqu'à présent.

Pour lancer cette version, nous avons utilisé la commande :

```
mvn exec:java \
  -Dexec.mainClass="tweetoscope.TweetoscopeApp" \
  -Dexec.args="recorded"
```

Les tests réalisés avec `scenarioTestBase.txt` et `miniTestBase.txt` nous ont permis de vérifier que :

- les tweets sont correctement lus depuis les fichiers
- le filtre transmet les bons tweets à l'extracteur
- le compteur de hashtags met bien à jour les valeurs affichées
- le nombre de tweets pris en compte par le visualiseur correspond à ce que l'on attend

Avec `largeTestBase.txt`, nous avons surtout vérifié que l'application tenait la charge et que le pipeline ne se bloquait pas, même lorsque le nombre de tweets augmente fortement.

4 Task 7 : création d'un nouveau filtre de tweets

4.1 Objectif et choix du filtre

La Task 7 demande de créer un nouveau filtre de tweets, par exemple en ne conservant que les messages récents ou ceux respectant une certaine contrainte. L'idée est de compléter les filtres déjà présents dans le projet, qui portent par exemple sur la langue ou sur le pays.

Nous avons choisi de mettre en place un filtre basé sur la taille du tweet. L'objectif est de ne garder que les messages contenant au moins un certain nombre de mots, hashtags compris. Cela permet de filtrer les messages très courts qui ne contiennent pas vraiment d'information utile.

4.2 Implémentation du filtre

Nous avons introduit une classe `SizeTweetFilter` qui suit la même structure que les autres filtres du projet. Cette classe possède un attribut `size` indiquant le nombre minimum de mots requis.

Le principe est le suivant :

- le texte du tweet est récupéré
- il est découpé en mots en se basant sur les espaces
- si le nombre de mots est supérieur ou égal au seuil, la méthode de filtrage renvoie vrai ; sinon, le tweet est ignoré

Le filtre est ensuite intégré au pipeline en modifiant la logique de `TweetFilter` pour ajouter un nouveau cas correspondant à cette stratégie. Cela nous permet de choisir ce filtre à l'exécution, de la même manière que les filtres `none` ou `language`.

4.3 Tests avec miniTestBase

Pour tester le nouveau filtre, nous avons utilisé `miniTestBase.txt` en choisissant un seuil relativement bas (par exemple deux ou trois mots) afin de pouvoir contrôler manuellement le résultat.

Nous avons vérifié, en comparant le contenu de `miniTestBase.txt` et les messages affichés par le visualiseur, que seuls les tweets suffisamment longs étaient conservés. Ces tests nous ont aussi servi de base plus tard, lors de la mise en place d'un test unitaire pour ce filtre dans la Task 14.

5 Task 8 : architecture Kafka et analyse de risques

5.1 Objectif de la refactorisation

La Task 8 marque le passage de l'application locale monolithique à une architecture distribuée basée sur Kafka. L'idée est de découpler chaque étape du pipeline dans un service indépendant, qui communique avec les autres uniquement via des topics Kafka.

Cela permet :

- de faire évoluer chaque service séparément
- de répartir la charge sur plusieurs instances et plusieurs machines
- de rejouer les données en cas de besoin
- de rendre le système plus tolérant aux pannes

5.2 Découpage en services

Nous avons découpé le pipeline en cinq services principaux :

- *ReplayerProducer*, qui lit les tweets enregistrés et les publie dans un premier topic Kafka
- *FilterService*, qui consomme ce topic, applique le filtre choisi, et republie les tweets retenus
- *ExtractorService*, qui extrait les hashtags des tweets filtrés
- *CounterStreams*, une application Kafka Streams, qui compte les occurrences de chaque hashtag en continu
- *VisualizerService*, qui consomme les résultats de comptage et les affiche

Chaque service est désormais un programme Java autonome, avec sa propre classe `main`. La seule dépendance commune est l'accès à un cluster Kafka.

5.3 Topics et structure des messages

Nous avons défini quatre topics principaux :

- `tweets.raw`, qui contient les tweets bruts rejoués par le producteur
- `tweets.filtered`, qui contient les tweets ayant passé le filtre
- `hashtags`, qui contient les hashtags extraits
- `counts`, qui contient les paires (hashtag, nombre d'occurrences)

Service	Type	Lit	Écrit	Clé	Valeur	Identifiant
Replayer	Producteur	–	tweets.raw	tweetId	Tweet JSON	–
FilterSvc	Consommateur / producteur	tweets.raw	tweets.filtered	tweetId	Tweet filtré	filter-service
ExtractorSvc	Consommateur / producteur	tweets.filtered	hashtags	hashtag	hashtag	extractor-service
CounterSvc	Kafka Streams	hashtags	counts	hashtag	nombre (Long)	hashtag-counter
Visualizer	Consommateur	counts	–	hashtag	nombre (Long)	visualizer-service

Table 1: Rôles des services et topics utilisés dans notre architecture Kafka.

Le tableau suivant résume qui lit et écrit quoi, ainsi que les clés et valeurs utilisées :

Le choix de la clé n'est pas anodin. Pour les topics `tweets.raw` et `tweets.filtered`, nous utilisons l'identifiant du tweet. Pour les topics `hashtags` et `counts`, la clé est le hashtag lui-même, ce qui permet à Kafka Streams de regrouper facilement toutes les occurrences d'un même hashtag.

5.4 Représentation schématique du pipeline Kafka

5.5 Risk analysis / initial risk analysis

L'énoncé demande d'indiquer l'impact de la défaillance de chaque composant. Nous avons identifié les cas suivants.

ReplayerProducer Ce service est la source de données. S'il s'arrête, plus aucun tweet n'est injecté dans le pipeline. Les autres services continuent à fonctionner mais se retrouvent en attente de nouveaux messages. L'application ne plante pas, mais le visualiseur cesse de recevoir des mises à jour.

FilterService Ce service consomme les tweets bruts et publie ceux qui respectent les critères. S'il tombe en panne, les messages restent bloqués dans le topic `tweets.raw`. Les services situés en aval (extracteur, compteur, visualiseur) ne reçoivent plus rien. L'affichage se fige, mais le système peut repartir correctement dès que le service est relancé, puisqu'il suffit de reprendre la consommation du topic là où elle s'était arrêtée.

ExtractorService Si ce service s'arrête, les tweets filtrés continuent d'arriver dans `tweets.filtered`, mais aucun hashtag n'est produit. Le compteur ne reçoit donc plus de données et le visualiseur n'affiche plus de nouvelle valeur. Là encore, le redémarrage du service permet de rejouer les messages en retard.

CounterStreams Le compteur Kafka Streams agrège les occurrences de hashtags à partir du topic `hashtags`. En cas de défaillance, les hashtags continuent d'être produits, mais le topic `counts` n'est plus alimenté. Lorsque le service redémarre, il peut reconstruire son état à partir des données présentes dans Kafka, grâce au mécanisme de changelog et au stockage d'état associé à Kafka Streams.

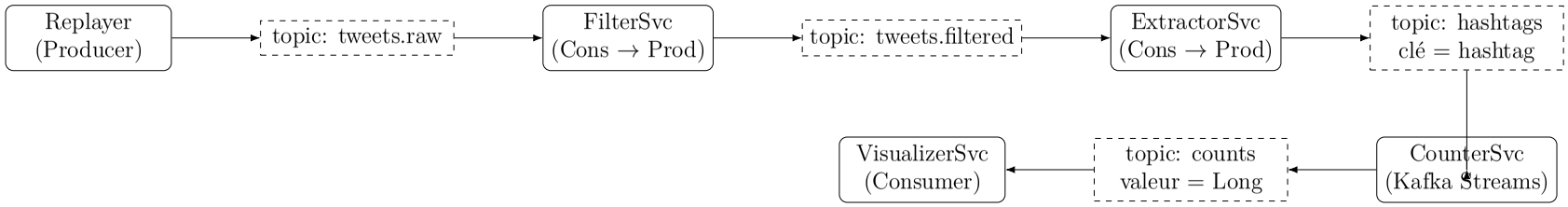

Figure 1: Vue d'ensemble du pipeline Kafka que nous avons mis en place.

VisualizerService Si le visualiseur s'arrête, le pipeline de traitement continue de fonctionner normalement. Les messages continuent d'être écrits dans le topic `counts`, mais ils ne sont plus affichés. Au redémarrage du visualiseur, il est possible de relire les derniers messages pour reprendre l'affichage.

Kafka broker Le broker Kafka est le point central de communication. En cas de panne du broker, les services ne peuvent plus produire ni consommer de messages. L'ensemble du pipeline est alors indisponible. La reprise nécessite la remise en route du cluster Kafka, éventuellement avec plusieurs brokers pour limiter ce type de risque.

Cette analyse de risques nous a servi de base pour les expérimentations de la Task 10, où nous avons volontairement arrêté certains services pour vérifier que le comportement observé correspondait à ce qui était prévu.

6 Task 9 : validation de l'architecture refactorisée

6.1 Objectif de la tâche

La Task 9 consiste à valider concrètement l'architecture refactorisée mise en place avec Kafka. L'énoncé demande en particulier :

- d'utiliser un consommateur en ligne de commande pour vérifier que chaque producteur Kafka émet bien les bons messages
- de vérifier que la solution fonctionne lorsque les services sont répartis sur plusieurs machines

6.2 Tests avec miniTestBase et console consumer

Nous avons d'abord lancé Kafka en local, puis démarré le *ReplayerProducer* avec le fichier `miniTestBase.txt` comme source. Nous avons utilisé l'outil `kafka-console-consumer` pour observer le contenu de chaque topic :

- sur `tweets.raw`, nous avons contrôlé que les tweets bruts étaient bien présents
- sur `tweets.filtered`, nous avons vérifié que seuls les tweets passant le filtre étaient relayés
- sur `hashtags`, nous avons inspecté la liste des hashtags générés par l'extracteur
- sur `counts`, nous avons observé l'évolution des compteurs au fil du temps

Ces tests nous ont permis de vérifier pas à pas le bon fonctionnement de chaque maillon du pipeline, en comparant les sorties obtenues avec les données d'entrée.

6.3 Exécution sur plusieurs machines

Dans un second temps, nous avons réparti les services sur plusieurs machines appartenant au même réseau, en conservant un seul cluster Kafka accessible depuis ces machines. Concrètement, Kafka tournait sur une machine, tandis que certains services (par exemple le visualiseur et l'extracteur) étaient exécutés sur une autre.

Les modifications nécessaires pour cela se limitent essentiellement à la valeur de l'adresse du broker Kafka, qui n'est plus `localhost:9092` mais une adresse IP du réseau. Le reste du code ne dépend pas de la machine sur laquelle il tourne.

Ces tests ont montré que :

- la communication entre services était bien assurée par Kafka
- l'architecture ne dépend pas d'un déploiement monolithique

- il est possible de déplacer ou dupliquer certains services sans changer l'architecture globale

Toutes les modifications apportées dans le cadre de cette tâche ont été intégrées dans le dépôt Git du projet.

7 Task 10 : comportement en cas de panne

7.1 Objectif de la tâche

La Task 10 demande de vérifier expérimentalement que le comportement de l'application en cas de panne est cohérent avec l'analyse de risques réalisée dans la Task 8. L'objectif est de ne pas rester sur une analyse théorique, mais de provoquer des défaillances contrôlées pour observer leurs effets.

7.2 Scénarios testés

Nous avons mis en place plusieurs scénarios simples :

- arrêt du service de filtrage pendant que le producteur continue à envoyer des tweets
- arrêt temporaire de l'extracteur de hashtags
- arrêt du visualiseur
- redémarrage du service de comptage Kafka Streams après une interruption

Dans chaque cas, nous avons laissé le producteur continuer à publier des tweets, en particulier avec `largeTestBase.txt`, afin de disposer d'un flux suffisant pour observer les effets.

7.3 Résultats observés

Les observations principales sont les suivantes :

- Lorsque le *FilterService* est arrêté, le topic `tweets.raw` continue de se remplir, mais `tweets.filtered` reste vide. Les services situés en aval n'affichent plus de mise à jour. Au redémarrage du filtre, les tweets accumulés sont progressivement traités.
- Lorsque l'*ExtractorService* est arrêté, `tweets.filtered` continue de recevoir des messages, mais le topic `hashtags` reste figé. À la reprise, on observe une rafale de hashtags correspondant aux tweets qui avaient été en attente.
- Lorsque le *VisualizerService* est arrêté, le topic `counts` continue de se remplir. Le redémarrage du visualiseur permet de reprendre la consommation là où elle s'était arrêtée, sans perte de données.
- Pour le service *CounterStreams*, le redémarrage après arrêt montre que Kafka Streams est capable de reconstruire son état à partir des données présentes dans Kafka, ce qui confirme l'intérêt du mécanisme d'état persistant.

Ces tests confirment que le comportement observé est conforme à notre analyse initiale : les pannes de services intermédiaires provoquent des blocages partiels mais pas de perte définitive de données, tant que Kafka reste disponible.

8 Task 11 : containerisation avec Docker

8.1 Objectif de la tâche

La Task 11 demande de containeriser chaque service de l'application, d'abord pour une exécution locale en utilisant des images locales, puis en poussant ces images sur un registre en vue d'exécutions à distance.

8.2 Images Docker pour les services

Nous avons choisi une approche assez classique pour les images applicatives :

- une image de base contenant un JDK pour construire le projet avec Maven
- une image finale plus légère contenant un JRE et les fichiers `jar` et dépendances nécessaires

Chaque service (replayer, filtre, extracteur, compteur Kafka Streams, visualiseur) est ensuite lancé en donnant la classe `main` à exécuter via la ligne de commande. Les connexions à Kafka sont configurées via des variables d'environnement, ce qui permet de réutiliser les mêmes images en local et en déploiement.

8.3 Image Kafka et script d'initialisation

Pour Kafka, nous nous appuyons sur une image existante fournie par un distributeur, ce qui évite de gérer la configuration du broker à la main. Un script d'initialisation supplémentaire crée les topics nécessaires (`tweets.raw`, `tweets.filtered`, `hashtags`, `counts`) s'ils n'existent pas déjà.

8.4 Fichier docker-compose

Nous avons ensuite écrit un fichier `docker-compose.yml` permettant de lancer, d'un seul coup, le broker Kafka, le script de création des topics et l'ensemble des services applicatifs. Ce fichier décrit les dépendances entre services, de sorte que Kafka soit prêt avant que les services ne tentent de se connecter.

Cette étape nous a permis d'obtenir un environnement reproductible : une seule commande suffit pour démarrer le pipeline complet, ce qui facilite les tests et les démonstrations.

9 Task 12 : déploiement Kubernetes et tolérance aux pannes

9.1 Objectif de la tâche

La Task 12 consiste à déployer l'application sur un cluster Kubernetes, à la place ou en complément de l'environnement Docker Compose utilisé en local. L'idée est de profiter des mécanismes de Kubernetes pour la tolérance aux pannes et la montée en charge.

9.2 Manifests de déploiement

Pour chaque service, nous avons défini :

- on ne déploie sur kubernetes que les fichiers yaml puisque l'image docker est pull de dockerhub. Par ailleurs puisque le mot de passe est le même pour tous les élèves, push tout notre code sur le cluster permettrait à tous nos camarades curieux de voir notre travail.
- un objet `Deployment`, décrivant l'image à utiliser, le nombre de réplicas et les variables d'environnement (notamment l'adresse de Kafka)
- éventuellement un `Service` pour exposer le composant à d'autres pods ou à l'extérieur du cluster

Le cluster Kafka lui-même peut être déployé dans Kubernetes ou accessible depuis l'extérieur. Dans nos tests, nous avons principalement utilisé un Kafka externe au cluster, ce qui limitait la complexité de la configuration.

9.3 Risk mitigation using Kubernetes

L'utilisation de Kubernetes apporte plusieurs mécanismes de mitigation des risques identifiés dans la Task 8 :

- le redémarrage automatique des pods en cas de crash
- la possibilité d'augmenter le nombre de réplicas pour un service qui devient un goulot d'étranglement
- le maintien de l'adresse logique d'un service, même si les pods sous-jacents sont recréés

Dans nos essais, nous avons par exemple augmenté le nombre de réplicas du service d'extraction pour vérifier que la charge pouvait être répartie sur plusieurs instances. Nous avons aussi observé que la suppression d'un pod de visualisation entraînait son redémarrage automatique, sans intervention manuelle.

10 Task 13 : pipeline GitLab CI pour le rapport

10.1 Objectif et choix techniques

La Task 13 demande de mettre en place une pipeline GitLab CI capable de compiler automatiquement les sources LaTeX du rapport que vous avez sous les yeux et de publier le PDF résultant sur GitLab Pages. L'idée est que toute modification poussée sur le dépôt déclenche une recompilation du rapport et, si tout se passe bien, une mise à jour de la version publiée. Nous avons décidé de travailler d'abord sur une branche séparée nomée ci-test pour éviter les problèmes de compatibilité.

10.2 Configuration de base

Nous avons ajouté un fichier `.gitlab-ci.yml` contenant un job latex build (composé de deux stages : build et deploy) qui a pour but de créer un pdf à partir du latex fourni et de le déployer sur GitLab Pages.

Le stage build de ce job

- utilise une image Docker contenant un environnement LaTeX complet (en l'occurence texlive)
- exécute la commande `pdflatex` sur le fichier main.tex ce qui a pour effet de le compiler et de produire un document appelé "document.pdf"
- dépose le PDF généré dans le répertoire public crée pour l'occasion (ce répertoire est nécessaire pour la phase suivante du pipeline : deploy)
- déploi

Le stage deploy de ce job

- utilise une image très légère (alpine) puisque il s'agit juste de déplacer un fichier déjà existant.
- a accès au document précédemment crée via artifacts : - public
- dépose le PDF généré dans le répertoire public crée pour l'occasion (ce répertoire est nécessaire pour la phase suivante du pipeline : deploy)
- déploi

11 Task 14 : tests unitaires pour le nouveau filtre

11.1 Objectif du test

La Task 14 demande d'écrire un test JUnit pour SizeTweetFilter, puis d'intégrer ce test dans la pipeline CI. L'objectif est double : valider automatiquement le comportement du filtre et montrer l'utilisation de tests unitaires dans le projet.

11.2 Test du filtre de taille

Nous avons écrit un test JUnit pour le filtre de taille `SizeTweetFilter`. Le principe est le suivant :

- créer un filtre avec un seuil donné, par exemple dix mots
- construire un tweet avec un texte plus court que ce seuil, et vérifier que le filtre le rejette (avec la méthode assertFalse)
- construire un tweet avec un texte plus long, et vérifier que le filtre l'accepte (cette fois-ci, assertTrue)
- créer un filtre avec un seuil de 0 et vérifier qu'il laisse passer les tweets avec un texte vide.

Ce test permet de s'assurer que la logique du filtre ne se dégrade pas au fil des modifications. Il a été ajouté à la suite des autres tests, et la pipeline CI est configurée pour exécuter les tests Maven (`mvn test`) à chaque push.

12 Task 15 : intégration des images Docker dans la CI

12.1 Objectif

La Task 15 demande d'intégrer la construction des images Docker dans la pipeline CI et d'adapter les fichiers de déploiement Kubernetes pour utiliser les images du registre GitLab, plutôt que celles d'un registre public.

12.2 Build automatique des images

Dans le fichier `.gitlab-ci.yml`, nous avons ajouté un job de type build qui :

- se connecte au registre de conteneurs GitLab associé au projet
- construit les images Docker pour les différents services à partir des Dockerfiles du dépôt
- pousse ces images vers le registre

Les noms des images et des tags sont choisis de manière cohérente avec les recommandations de GitLab, en utilisant par exemple la variable `$CI_COMMIT`.

12.3 Adaptation des manifests Kubernetes

Les fichiers de déploiement Kubernetes ont été modifiés afin de pointer vers les images hébergées sur le registre GitLab, plutôt que vers Docker Hub. Cette étape permet de relier directement le cycle de développement (code, CI, build) au déploiement, sans dépendre d'un registre externe.

13 Task 16 : qualité du code et intégration continue

13.1 Objectif

La Task 16 demande de mettre en place un processus de vérification de la qualité du code et de l'intégrer à la CI. L'idée est de détecter au plus tôt les problèmes de style, de complexité ou de bugs potentiels.

13.2 Outils et intégration

Dans notre cas, nous avons privilégié une approche simple et compatible avec l'environnement fourni :

- Utilisation de SonarQube pour repérer au plus tot les code smells
- Verification du passage des tests unitaires à chaque modification, grâce aux tests crées à la Task 14

Cette étape permet de repérer certaines erreurs fréquentes ou mauvaises pratiques (méthodes trop longues, conventions de nommage, etc.) et de les éliminer au fur et a mesure.

14 Task 17 : Retour d'expérience

14.1 Ce que nous avons appris

Ce projet nous a permis de mettre en pratique plusieurs notions abordées en cours :

- Dépoussiérer les vieilles connaissances d'utilisation de git (qui est quand même un must)
- comprendre plus concrètement le fonctionnement d'un pipeline de traitement de flux
- expérimenter une architecture à base de Kafka, avec plusieurs services indépendants
- se servir des outils d'IA actuelles qui permettent d'augmenter grandement la vitesse de travail et de compréhension (peut-être au prix d'une longévité de cette compréhension)
- containeriser une application
- containeriser une application et la déployer dans un environnement orchestré
- mettre en place une pipeline CI/CD simple mais fonctionnelle pour le code et pour le rapport

Nous avons aussi pris conscience de la difficulté à garder une vue d'ensemble lorsque plusieurs technologies différentes sont en jeu (Java, Kafka, Docker, Kubernetes, GitLab CI, LaTeX). Le fait de découper le travail par tâches, comme le propose l'énoncé, nous a aidés à progresser étape par étape.

14.2 Utilisation des outils d'IA et retour d'expérience

Dans le cadre de ce projet, nous avons utilisé ponctuellement des outils d'assistance basés sur des modèles de langage d'IA générative. Il est important de détailler précisément la manière dont nous nous en sommes servis, non pas comme une source de solutions prêtes à l'emploi, mais comme un appui pour gagner du temps sur certaines tâches répétitives et pour confronter nos propres raisonnements à une seconde opinion technique.

Les interventions concernaient :

- la reformulation ou l'explication de certains extraits de code Java issus de la base existante
- la génération de brouillons de structures de classes pour vérifier la cohérence de notre architecture
- la production d'exemples simples de configuration Kafka ou d'appels d'API Java pour valider notre compréhension

- l'aide à la rédaction (reformulations, corrections de style, structuration de paragraphes)

Nous insistons sur le fait que le *contenu final* du code et des choix d'architecture provient de notre propre travail. Pour chaque élément suggéré par l'IA, nous avons suivi une démarche systématique :

14.2.1 Conception initiale par nous-mêmes

Pour toutes les fonctionnalités, nous avons d'abord élaboré notre propre solution :

- rédaction de pseudocode ou d'un prototype en Python pour valider la logique ;
- schémas d'architecture faits manuellement (topics, clés, partitions) ;
- réflexion personnelle sur la gestion des offsets, des groupes de consommateurs et de la tolérance aux pannes.

L'IA n'intervenait qu'après cette première étape.

14.2.2 Comparaison avec les propositions de l'IA

Lorsque nous demandions une validation ou une alternative, nous analysions systématiquement :

- la cohérence entre ce que nous avions conçu et ce que l'IA proposait ;
- les différences de structure (par exemple dans la gestion des `ProducerRecord`, ou dans la façon de parcourir les messages Kafka) ;
- la présence éventuelle d'erreurs ou d'imprécisions dans la version de l'IA.

Il arrivait souvent que nous corrigions ou rejetions les suggestions proposées, parfois pour la simple et bonne raison que ce que l'IA avait produit ne compilait pas.

14.2.3 Vérification systématique par tests réels

Nous avons testé *toutes* les parties du code indépendamment :

- exécution locale de chaque service Kafka dans des conteneurs séparés ;
- utilisation de `kafka-console-consumer` pour vérifier manuellement le contenu des topics ;
- tests de charge avec le fichier `largeTestBase` ;
- comparaison entre les résultats produits par nos implémentations et nos attentes théoriques.

Lorsque l'IA proposait une structure de code trop abstraite ou incompatible, nous l'avons retravaillée ligne par ligne.

14.2.4 Utilité réelle des outils d'IA dans notre travail

L'IA nous a été utile surtout pour :

- accélérer la rédaction de commentaires ou de descriptions textuelles
- produire un squelette rapide pour une classe Kafka Streams ou un consommateur
- synthétiser une idée pour vérifier que notre compréhension était correcte
- générer la base du document que vous êtes en train de lire

Cependant, sur les aspects techniques, nous avons constaté plusieurs limites :

- certaines suggestions de code pour Kafka étaient obsolètes ou incorrectes
- les exemples fournis ne tenaient pas toujours compte de notre structure de projet
- il a fallu retraduire en Java plusieurs concepts que nous avions initialement formulés en Python ou en pseudocode ;
- les configurations proposées pour Docker ou Kubernetes n'étaient pas directement fonctionnelles

Ainsi, l'IA n'a pas fourni des solutions exploitables immédiatement : elle nous a surtout permis d'explorer des variantes, de clarifier des idées ou d'accélérer certains aspects formels (comme la rédaction).

Conclusion

Le projet Tweetoscope nous a fait passer d'une application Java monolithique à une architecture distribuée complète, s'appuyant sur Kafka, Docker, Kubernetes et GitLab CI. En suivant les différentes tâches de l'énoncé, nous avons progressivement :

- compris et validé le fonctionnement de l'application initiale
- introduit des jeux de données enregistrés pour obtenir des tests reproductibles
- créé un nouveau filtre de tweets (SizeTweetFilter) et l'avons testé
- refactoré l'architecture en pipeline Kafka, avec plusieurs services indépendants
- analysé les risques et testé le comportement en cas de panne
- containerisé les services et mis en place leur déploiement sur un cluster Kubernetes
- configuré une pipeline CI pour le rapport, pour les tests unitaires et pour la construction des images Docker
- réalisé une vidéo de démonstration et pris du recul sur l'utilisation des outils d'IA.

Même si tout n'est pas parfait et que certains aspects pourraient être encore améliorés, ce projet nous a donné une vision concrète des enjeux liés aux systèmes distribués et à leur industrialisation. Il nous a aussi permis de nous confronter à des outils qui sont largement utilisés dans le monde professionnel, ce qui était l'un des objectifs affichés du cours.