TD-5 CR

Zhan YE, Jiaqi QI

11 mars 2025

1 Objectif

1. Utilisez Docker et Docker-Compose / Kubernetes pour exécuter vos microservices.
2. Vérifiez l'ordonnancement des événements pour éviter les boucles infinies.
3. Exploitez les logs et outils de monitoring pour diagnostiquer les erreurs.
4. Assurez-vous que votre système est scalable et tolérant aux pannes.

Livrable :

1. Code source des microservices et leurs interactions.
2. Fichier de configuration pour Celery et RabbitMQ/Kafka.
3. Documentation API (Swagger/OpenAPI).
4. Diagramme d'architecture du système.
5. Rapport expliquant les choix techniques et les problèmes rencontrés.

2 Structuration en Microservices et Communication

Le processus d'évaluation des demandes de prêt comprend plusieurs services

1. Décomposer le processus en microservices et proposer une architecture.

   Nous l'avons divisé en fonction du processus BPMN TD3 précédent. Au total, il est divisé en quatre microservices. Le premier est la validation, qui est utilisée pour vérifier la complétude de la demande de prêt du client, par exemple si la date de remboursement et le montant du prêt sont renseignés. Si les deux sont remplis, la demande est envoyée au sujet 'validated_request' de la file d'attente de messages Kafka et le client est notifié par l'intermédiaire de l'SSE. Dans le cas contraire, seul le client est notifié. Le second est l'évaluation, qui est utilisée pour analyser à la fois les biens du client (en appelant la base de données existante, que nous simulons avec un fichier json) et le risque du prêt, et pour donner la décision finale et notifier le client. Nous consommons le premier sujet via Kafka et le poussons vers le second sujet d'acceptation. Le troisième service consiste à générer les documents relatifs au prêt et aux assurances facultatives, comme le montant du remboursement mensuel, etc. Ici, nous lisons d'abord les sujets dans Kafka, puis nous utilisons Celery pour les exécuter en arrière-plan et les faire persister dans Redis avant de les envoyer à Kafka. Nous modélisons la partie boîte blanche de notre BPMN, la partie boîte noire comprend la façon dont le client interagit avec Kafka que nous n'aborderons pas ici, par exemple, à la fin nous lisons le sujet Kafka par défaut qui est signé par le client.

   Plus précisément, notre API est très simple, seul le premier service a des endpoints comme validation pour valider les demandes de prêt soumises par les clients. Le reste de l'API est automatisé, comme l'écoute permanente des sujets Kafka. Il y a également des endpoints comme healthy (pour valider l'état de Kafka) et events (SSE pour l'envoi au client, front-end). La logique du code a été encapsulée dans différentes fonctions plutôt que dans des API, afin de

maintenir une agrégation élevée et un couplage faible (nous ne voulons pas que des API distinctes s'appellent les unes les autres).

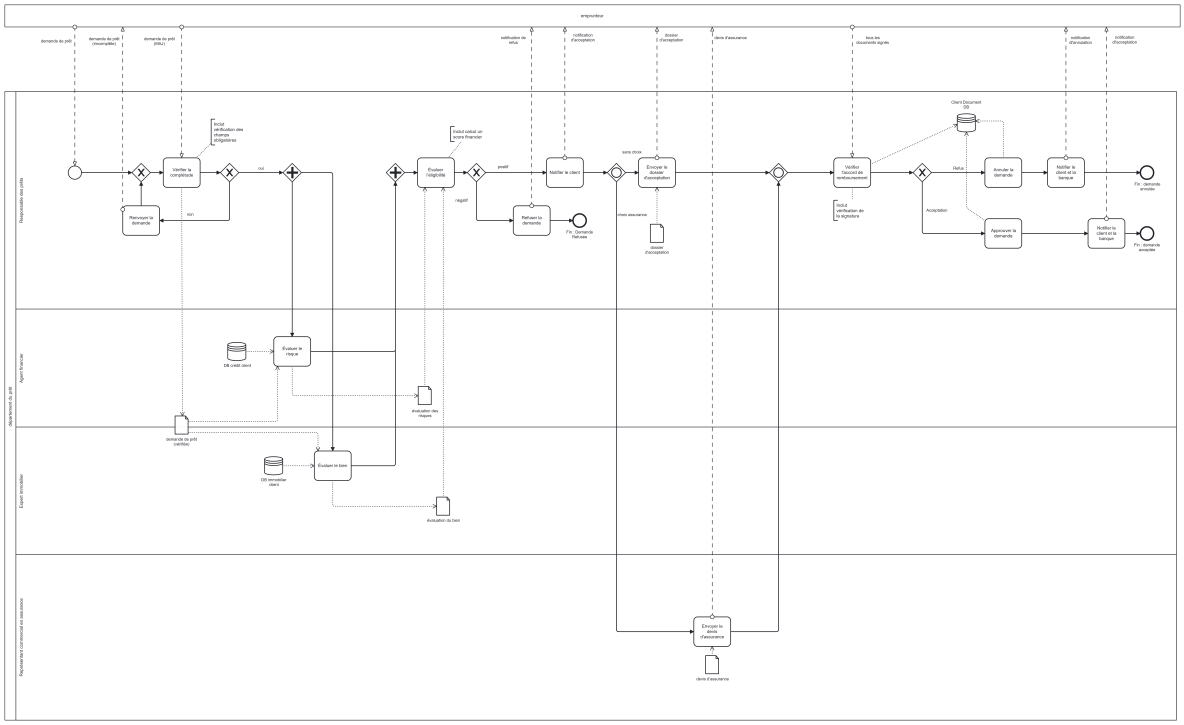

FIGURE 1 – BPMN Process

Dans la figure ci-dessous, le jaune représente chaque service FastAPI que nous avons encapsulé. Le Endpoint ou le composant requis y est connecté. Nous n'avons pas montré ici les fonctions internes de chaque service. Mais nous les avons installées selon la logique indiquée dans la figure ci-dessus.

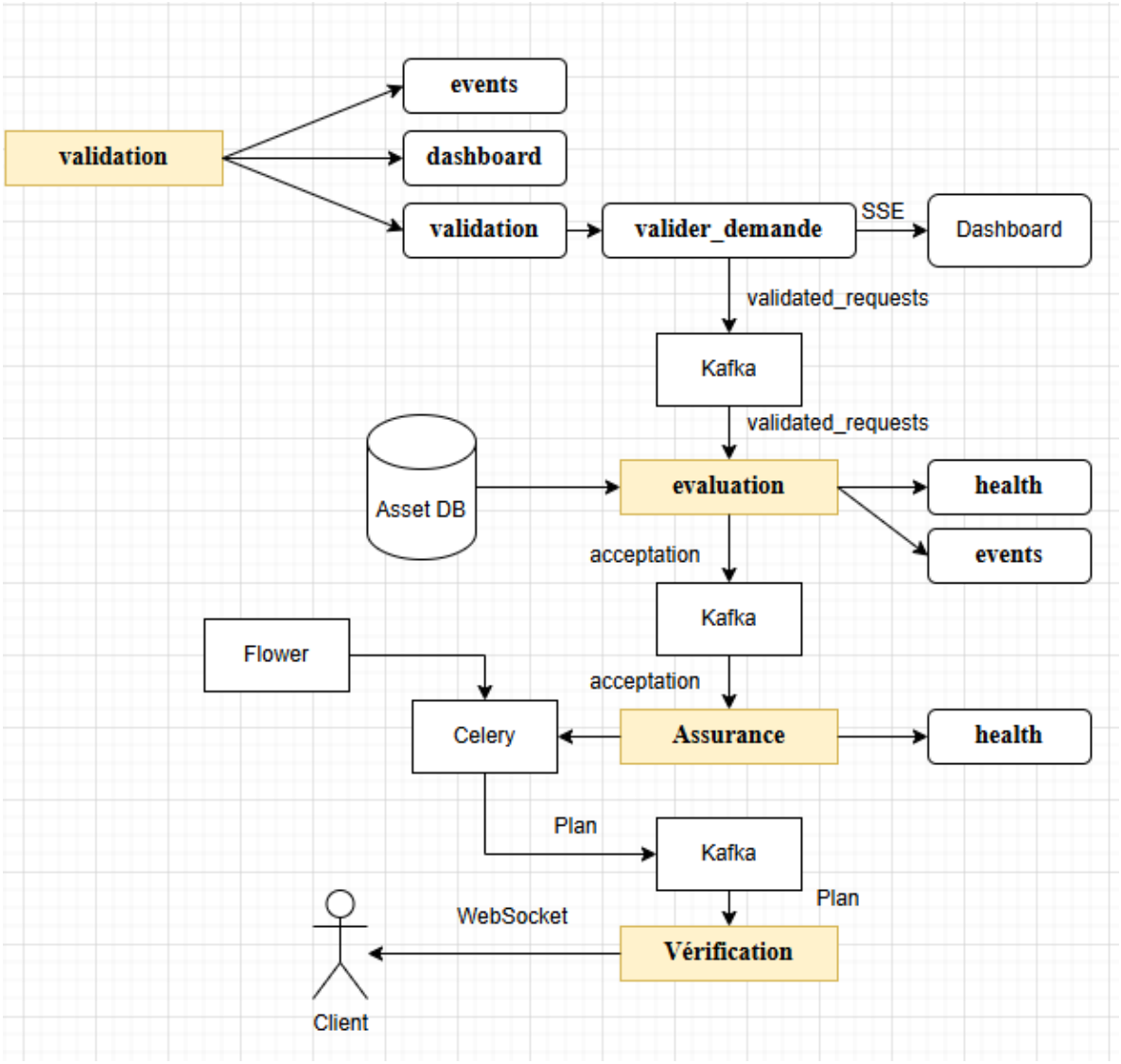

FIGURE 2 – Architecture des Microservices

2. Modéliser la communication entre les services en utilisant une file de messages (RabbitMQ, Kafka).

Nous simulons le flux de messages à l'aide de Kafka, une file d'attente de messages distribués qui gère le flux de données en temps réel. Dans ce cas, le producteur peut envoyer des messages à un sujet spécifique, et nous pouvons également spécifier la méthode de partitionnement (par exemple, aléatoire ou RoundRobin). Les consommateurs peuvent s'abonner à un ou plusieurs thèmes et consommer les messages dans un ordre partitionné. Nous pouvons également utiliser des groupes de consommateurs, où plusieurs consommateurs d'un groupe peuvent consommer plusieurs partitions en parallèle pour améliorer l'efficacité de la consommation. En ce qui concerne le stockage, un cluster Kafka se compose de plusieurs Brokers, chacun d'entre eux étant une instance de Kafka. En outre, chaque thème peut avoir plusieurs partitions, chacune étant stockée sur un Broker différent afin d'améliorer la concurrence et l'évolutivité. Les producteurs peuvent écrire des messages dans plusieurs partitions, et les consommateurs peuvent lire des messages provenant de plusieurs partitions. Chaque message d'une partition possède un offset unique, qui identifie l'emplacement du message et permet au consommateur de suivre la progression du message par l'intermédiaire de l'offset. C'est le paramètre que nous devons connaître pour garantir l'atomicité par la suite.

Plus précisément, le processus commence par une requête soumise par le client, que nous simulons avec une requête Http. Nous la validons ensuite et, si elle est acceptée, elle est envoyée au sujet « validated_request ». Nous la consommons ensuite dans un deuxième microservice et si elle passe l'évaluation, elle est envoyée au sujet « acceptation ». Immédiatement après, le troisième service consomme ce topic et envoie le fichier généré via Celery au topic « plan » dans Kafka. Enfin, le quatrième microservice consomme et notifie au client la fin du processus.

En termes d'implémentation du code, Kafka est installé sous Linux (WSL) mais les appels sont effectués en Python sous Windows, nous devons donc trouver le bon port externe WSL au lieu de localhost, et nous devons ajuster l'écoute IPv6 en IPv4 ainsi que rendre Kafka auto-créable (auto.create. topics.enable=true). Malheureusement, ceci est désactivé dans notre version de Kafka. Nous utilisons toujours la commande pour créer de nouveaux sujets, par exemple :

```
bin/kafka-topics.sh --create --topic validated_requests --bootstrap-server 172.28.146.43:9092 #

docker exec -it kafka kafka-topics.sh --create --topic validated_requests --bootstrap-server ka
```

De plus, toutes les commandes de test nécessaires (Kafka, Zookeeper, Docker Compose, Flower, Celery, Redis, etc.) ont été placées sous forme de commentaires au bas des fichiers correspondants. Par example :

```
sudo service redis-server start # démarrer le redis server
celery -A tasks worker --loglevel=info
celery -A celery_app worker --loglevel=info
celery -A celery_app flower --port=5555
```

Enfin, nous voulions que notre conteneur soit un environnement complètement indépendant sans dépendre du Kafka ou du Zookeeper de notre hôte. Nous avons donc ajouté les composants kafka et zookeeper à docker-compose.yaml. Il en résulte que l'adresse pour lier le kafka interne et le zookeeper doit également être modifiée.

3. Implémenter une API REST pour chaque microservice en Python (FastAPI ou Flask).

Nous avons mis en œuvre quatre microservices avec FastAPI en suivant l'architecture précédente. Parmi ceux-ci, seul le premier service dispose d'une véritable API, les trois autres ressemblant davantage à des applications automatisées. En effet, le premier service nécessite que le client soumette une demande de prêt pour le déclencher, tandis que les autres sont des processus automatisés, tels que Kafka qui écoute toujours le port pour consommer des sujets.

Enfin, nous avons conteneurisé l'ensemble du processus, y compris les composants requis, à l'aide de Docker. Nous avons également ajouté des composants tels que Kafka, Zookeeper, Flower et Volume (plusieurs conteneurs peuvent accéder au même volume pour le partage et la persistance des données) à l'environnement. Le fichier docker spécifique se trouve dans notre code, comme nous l'avons compris et expliqué dans le TD précédent.

4. Définir des événements pour la transition entre les services (ex. : événement "Crédit vérifié").

Comme nous l'avons décrit précédemment. Le premier microservice transmet les demandes de prêt vérifiées (montants de prêt non négatifs, etc.) au deuxième microservice via Kafka. Le deuxième service transmet la demande de prêt évaluée au troisième service, et nous utilisons toujours Kafka. Le troisième service transmet les documents de prêt générés et l'assurance (facultative) au quatrième service.

5. Tester l'enchaînement des services en envoyant des requêtes HTTP et en observant la propagation des événements.

   Nous envoyons une demande de prêt à l'aide de curl et obtenons le résultat suivant.

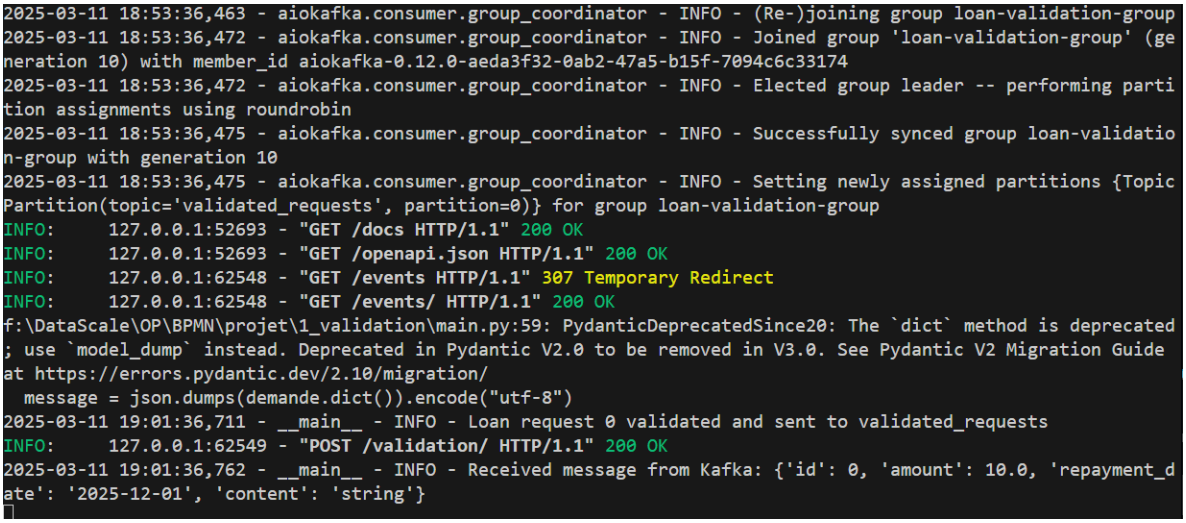

FIGURE 3 – Service 1 : Validation

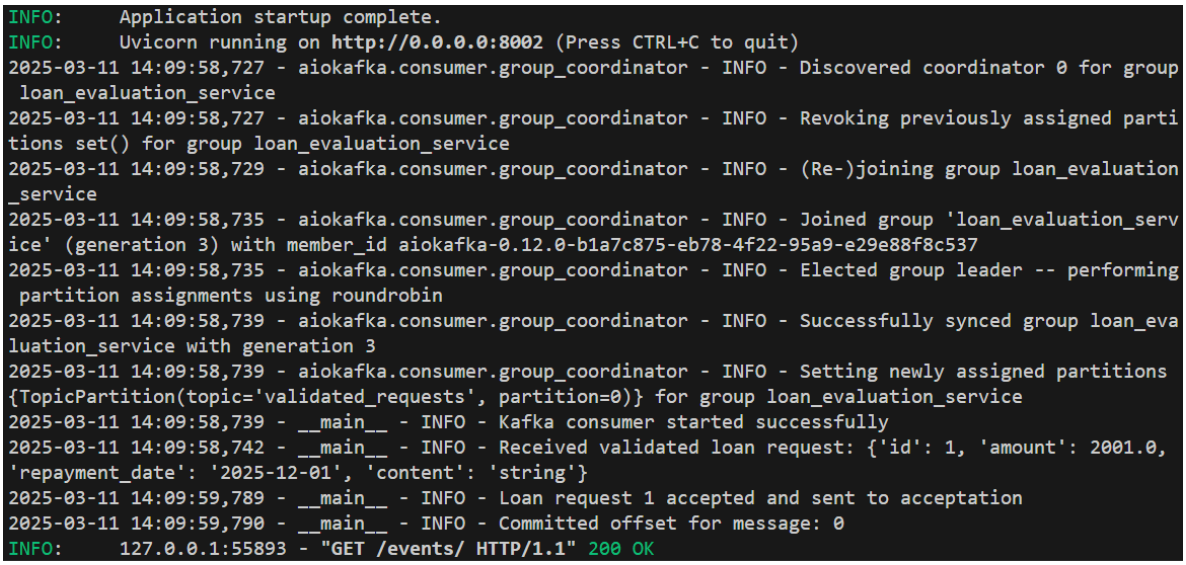

FIGURE 4 – Service 2 : Evaluation

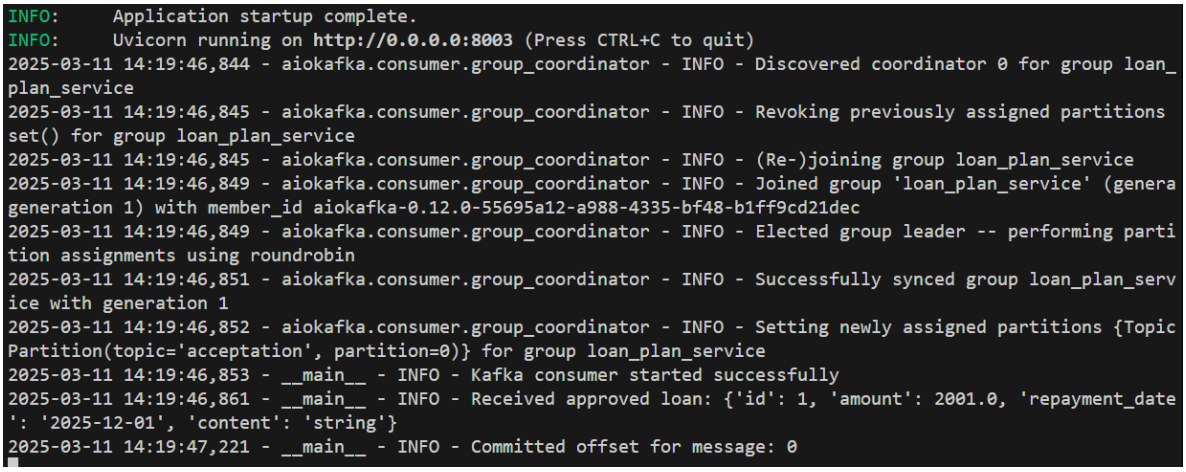

FIGURE 5 – Service 3 : Assurance

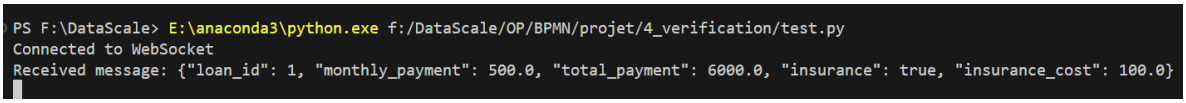

FIGURE 6 – Service 4 : Vérification

Pour l'ensemble du processus, nous avons écrit un fichier test.py dans le paquet du quatrième service, dédié à la connexion à la WebSocket avec l'ID correspondant et à la lecture des messages.

3 Tâches Asynchrones et Traitements en Arrière-Plan

Certaines étapes du processus métier nécessitent du temps (ex. : analyse du crédit, estimation immobilière). Il est inefficace d'attendre la réponse synchrone, ce qui justifie l'utilisation de tâches asynchrones.

Nous utilisons des fonctions asynchrones (async/await), en particulier dans le service consommateur Kafka. Les consommateurs Kafka doivent traiter plusieurs messages et communiquer avec Kafka sur le réseau pour effectuer plusieurs tâches en même temps (par exemple, l'évaluation des biens, l'évaluation des risques). Si vous utilisez l'approche synchrone, chaque fois qu'un message est traité, il occupe des threads, alors que l'utilisation de l'approche asynchrone permet à l'unité centrale de traiter des demandes multiples plus efficacement. En effet, avec l'approche synchrone traditionnelle, si une opération prend beaucoup de temps, l'ensemble du processus sera bloqué jusqu'à ce que l'opération soit terminée. L'approche asynchrone évite le blocage en permettant l'exécution d'autres tâches pendant la période d'attente.

1. Utiliser Celery pour exécuter des tâches en arrière-plan : Vérification du score de crédit, Estimation immobilière, Génération et envoi de documents (ex. : calendrier de remboursement)

   Comme décrit précédemment, nous utilisons Celery pour que le service Generate Loan Documentation s'exécute en arrière-plan et persiste dans Redis. Étant donné que l'évaluation des prêts implique des tâches chronophages telles que la vérification des actifs et l'analyse des risques (que nous simulons dans le code via un délai), elle ne devrait pas bloquer le thread principal de l'interface FastAPI. Si l'API exécute ces tâches directement, l'utilisateur peut attendre 5 à 10 secondes pour obtenir une réponse. Celery laisse ces tâches s'exécuter en arrière-plan, l'API les renvoie immédiatement et l'utilisateur peut interroger les résultats ultérieurement. En outre, l'état d'exécution de la tâche peut être stocké dans Redis, ce qui permet à la tâche de se poursuivre après une panne de courant et un redémarrage. Et si l'exécution échoue, Celery peut automatiquement réessayer.

Plus précisément, pour la fonction process_plan_task.apply_async qui s'exécute dans celery. Au lieu d'être exécutée directement, elle enverra la tâche à la file d'attente des tâches (Redis) et laissera le travailleur Celery la traiter. Le processus de Celery est le suivant : tout d'abord, Celery doit confirmer que process_plan_task est une tâche enregistrée. Ensuite, il sérialise la tâche, par exemple en convertissant data='id' : 1, 'amount' : 2001.0 au format JSON, ce qui génère également un identifiant de tâche. Enfin, Celery agit en tant que producteur et publie la tâche dans la file d'attente Redis. Du côté du travailleur Celery (le processus qui exécute celery -A tasks worker), le processus du travailleur écoute continuellement Redis, et lorsque le travailleur trouve une nouvelle tâche dans la file d'attente Redis, il extrait la tâche et la traite. Plus précisément, le travailleur commence par désérialiser, par exemple en analysant le champ de la tâche pour qu'il corresponde à la fonction de tâche Python tasks.process_plan_task. Par exemple, il analyse les kwargs et trouve data = « id » : 1, « amount » : 2001.0. Ensuite, la tâche est exécutée et Celery stocke la valeur de retour de la tâche dans Redis (backend=« redis ://localhost :6379/0 »). Enfin, le travailleur envoie un ACK pour confirmer que la tâche est terminée, et Redis supprime la tâche.

2. Configurer un worker Celery et assurer la persistance des tâches.

   La file d'attente des tâches (Broker), qui utilise ici Redis pour stocker les messages de la file d'attente des tâches. Stockage des résultats de la tâche (Backend), pour stocker les résultats de l'exécution de la tâche, de sorte qu'il soit facile d'interroger ultérieurement l'état de la tâche.

```
task_serializer="json"Les tâches sont stockées au format JSON.
result_serializer="json"Les tâches sont stockées au format JSON.
accept_content=["json"]N'accepte que des données JSON pour une meilleure sécurité.
result_expires=3600Les résultats de l'exécution de la tâche sont stockés pendant une heure, pui
```

```python
from celery import Celery

# Create Celery instance, using Redis as broker and backend storage
celery_app = Celery(
    "loan_plan_service",
    broker="redis://localhost:6379/0",
    backend="redis://localhost:6379/0"
)

# Celery configuration
celery_app.conf.update(
    task_serializer="json",  # Use JSON format for task serialization
    result_serializer="json",  # Store results in JSON format
    accept_content=["json"],  # Accept only JSON data
    result_expires=3600,  # Task results expiration time (in seconds)
)
```

FIGURE 7 – Celery Configuration

3. Implémenter un mécanisme de monitoring pour suivre l'état des tâches (Flower ou Prometheus).

   Nous utilisons Flower pour contrôler l'état d'avancement du processus.Il peut surveiller l'exécution des tâches (succès, échec, en cours), voir le statut des travailleurs Celery (en ligne/hors ligne), observer le temps d'exécution des tâches, la charge de la file d'attente, etc. Flower n'a pas besoin de modifier le code Celery, et tant que Celery fonctionne correctement, Flower peut récupérer le statut d'exécution des tâches par le biais du système d'événements et de la file

d'attente de messages de Celery. Flower s'appuie sur le système d'événements Celery et sur le backend Celery (magasin de résultats de tâches). Chaque fois que l'état d'une tâche change, Celery envoie des événements au backend (Redis), et Flower écoute ces événements pour obtenir l'état d'exécution de la tâche. Flower se connecte au Celery Worker via le broker_url de Celery (une file d'attente de messages, telle que Redis) pour obtenir des informations sur les tâches dans la file d'attente de tâches. Il s'abonne aux événements Celery et capture les changements d'état de la tâche. Enfin, Flower interroge périodiquement le backend de Celery (par exemple, Redis) pour obtenir les résultats de l'exécution de la tâche afin de s'assurer que tous les états sont visualisés.

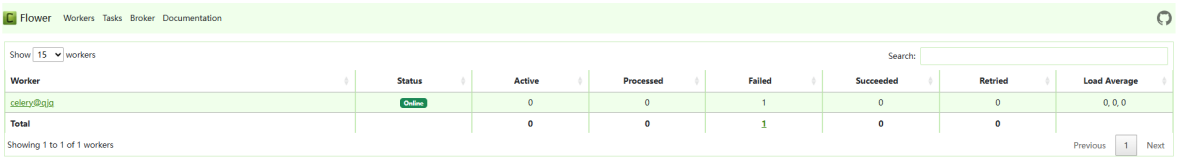

FIGURE 8 – Flower Web UI

4. Gérer les erreurs et la persistance en cas d'échec d'une tâche.

   Nous ajoutons ce code utilisé pour s'assurer que si Celery ne parvient pas à se connecter à un Broker (file d'attente de messages, par exemple Redis) au démarrage, il réessayera automatiquement la connexion au lieu d'échouer et de sortir immédiatement.

   ```
   celery_app.conf.broker_connection_retry_on_startup = True
   ```

   De plus, Celery permet de stocker l'état des tâches dans un backend de persistance (Redis). Cela permet de retrouver et analyser les erreurs après exécution. Celery stocke automatiquement l'état de chaque tâche dans Redis. Nous pouvons récupérer les logs d'erreur à partir de l'ID de la tâche.

4 Messagerie et Gestion des Transactions

Les services doivent échanger des informations de manière fiable et atomique. Une transaction ne doit pas être validée si l'une des étapes échoue.

— Mettre en place RabbitMQ ou Kafka pour la gestion des flux d'information entre les services.

Nous utilisons Kafka, et nous l'implémentons comme expliqué précédemment. Les détails sont dans notre code.

— Implémenter un mécanisme de compensation pour garantir la cohérence en cas d'échec partiel (ex. : si l'évaluation du bien échoue, annuler l'évaluation du crédit).

Comme nous l'avons mentionné précédemment, nous assurons l'atomicité du service en manipulant manuellement l'offset. Nous ne validons pas l'offset automatiquement. En d'autres termes, nous ne devons valider l'offset que si l'ensemble du service réussit. Si l'une des exécutions de la fonction échoue, notre offset ne changera pas. Voici un extrait du code :

```
enable_auto_commit=Flase
```

```python
async def consume_kafka():
    """Kafka consumer listens to `acceptation` topic"""
    global consumer, consumer_status

    while True:
        try:
            consumer = AIOKafkaConsumer(
                ACCEPTED_TOPIC,
                bootstrap_servers=KAFKA_BROKER,
                auto_offset_reset="earliest",
                enable_auto_commit=False,
                group_id="loan_plan_service",
            )

            await consumer.start()
            consumer_status["running"] = True
            consumer_status["error"] = None
            logger.info("Kafka consumer started successfully")
```

FIGURE 9 – Kafka offset pour l'atomicité

— Assurer la durabilité des messages pour éviter leur perte en cas de panne du système.

Kafka utilise la réplication (mécanisme de réplique). La durabilité peut être améliorée grâce au paramètre acks, acks=0 : le producteur n'attend pas que Kafka accuse réception du message, ce qui peut entraîner une perte de données (si le producteur se bloque immédiatement après l'envoi). acks=1 (par défaut) : Kafka attend uniquement que le leader accuse réception du message, qui peut être écrit par le leader mais pas encore répliqué par le suiveur, et si le leader se bloque, des données peuvent être perdues. acks=all : Kafka ne renvoie un succès que lorsque le leader et tous les suiveurs ont confirmé la réception des données, ce qui garantit que les données ne seront pas perdues.

En outre, Celery persiste également les tâches dans Redis pour faire face aux échecs.

5 Notifications et Communication en Temps Réel

Les clients doivent être informés de l'état de leur demande en temps réel (approbation, rejet, demande d'informations complémentaires).

— Mettre en place un WebSocket avec FastAPI pour les notifications en direct.

À la fin du dernier microservice, nous informons le client du résultat final du processus et de la fin du processus via WebSocket. Cette technologie permet au serveur et au client d'échanger des données en temps réel, de manière continue, contrairement au modèle traditionnel requête-réponse HTTP. En d'autres termes, le client et le serveur peuvent envoyer des messages à tout moment, sans attendre l'autre partie de la requête. En termes de mise en œuvre, c'est relativement simple. Nous nous connectons à un serveur WebSocket et écoutons les messages

du serveur dans une boucle infinie. Le résultat ressemble à l'illustration précédente.

— Gérer les événements en temps réel en utilisant Server-Sent Events (SSE).

Server-Sent Events (SSE) est une technologie unidirectionnelle de transmission de données en temps réel qui permet aux serveurs de transmettre des données aux clients sur de longues connexions (connexions persistantes HTTP). Contrairement aux WebSockets (communication bidirectionnelle), SSE ne permet qu'une communication unidirectionnelle du serveur vers le client. SSE permet au serveur d'envoyer des données dès qu'elles sont disponibles et repose sur une connexion HTTP standard, sans nécessiter de gestion WebSocket complexe. Ainsi, chaque fois que Kafka consomme des données, il en informe le client via SSE. Nous l'utilisons comme un endpoint API.

— Implémenter un dashboard simple qui affiche les mises à jour en temps réel du statut des demandes.

Nous utilisons Jinja2 pour présenter un tableau de bord sur le front-end qui met à jour le statut des demandes de prêt en temps réel. (Le résultat de SSE)

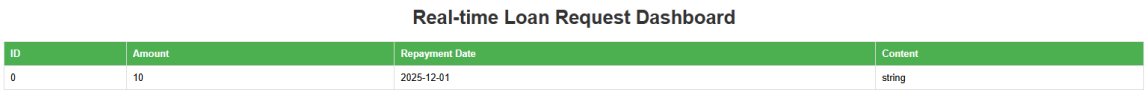

FIGURE 10 – SSE Dashboard

— Documentation Swagger

Comme nous l'avons mentionné précédemment, à l'exception du premier service, les autres services ne contiennent que des points de terminaison EVENTS et HEALTH (pour montrer l'état d'avancement). Seul le premier service dispose d'une API distincte pour recevoir les demandes de prêt des clients vérifiés. Ici, nous ne montrons que l'API de validation. input est le schéma Demande que nous avons défini. id est l'identifiant du client, amount est le montant du prêt, repayment_date est la date de remboursement. content est d'autres remarques comme le fait de savoir si l'assurance est requise ou non. Voici notre test CURL ou directement en Swagger.

```
# {
#     "id": 0,
#     "amount": 10,
#     "repayment_date": "2025-12-01",
#     "content": "string"
# }

# curl -X 'POST' 'http://localhost:8001/validation/' \
#  -H 'Content-Type: application/json' \
#  -d '{"id": 1, "amount": 1000, "repayment_date": "2025-12-01"}'
```

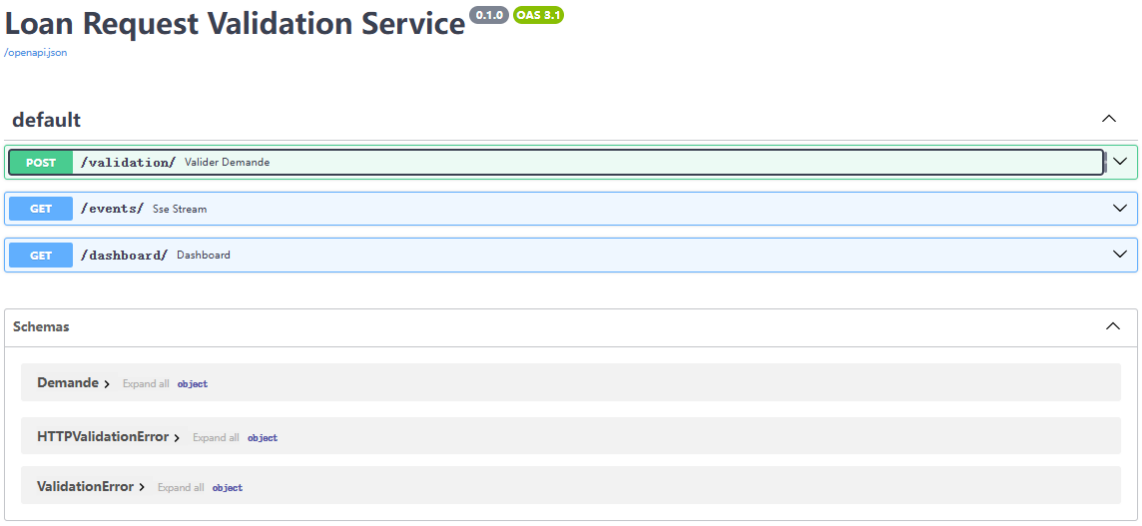

FIGURE 11 – Swagger

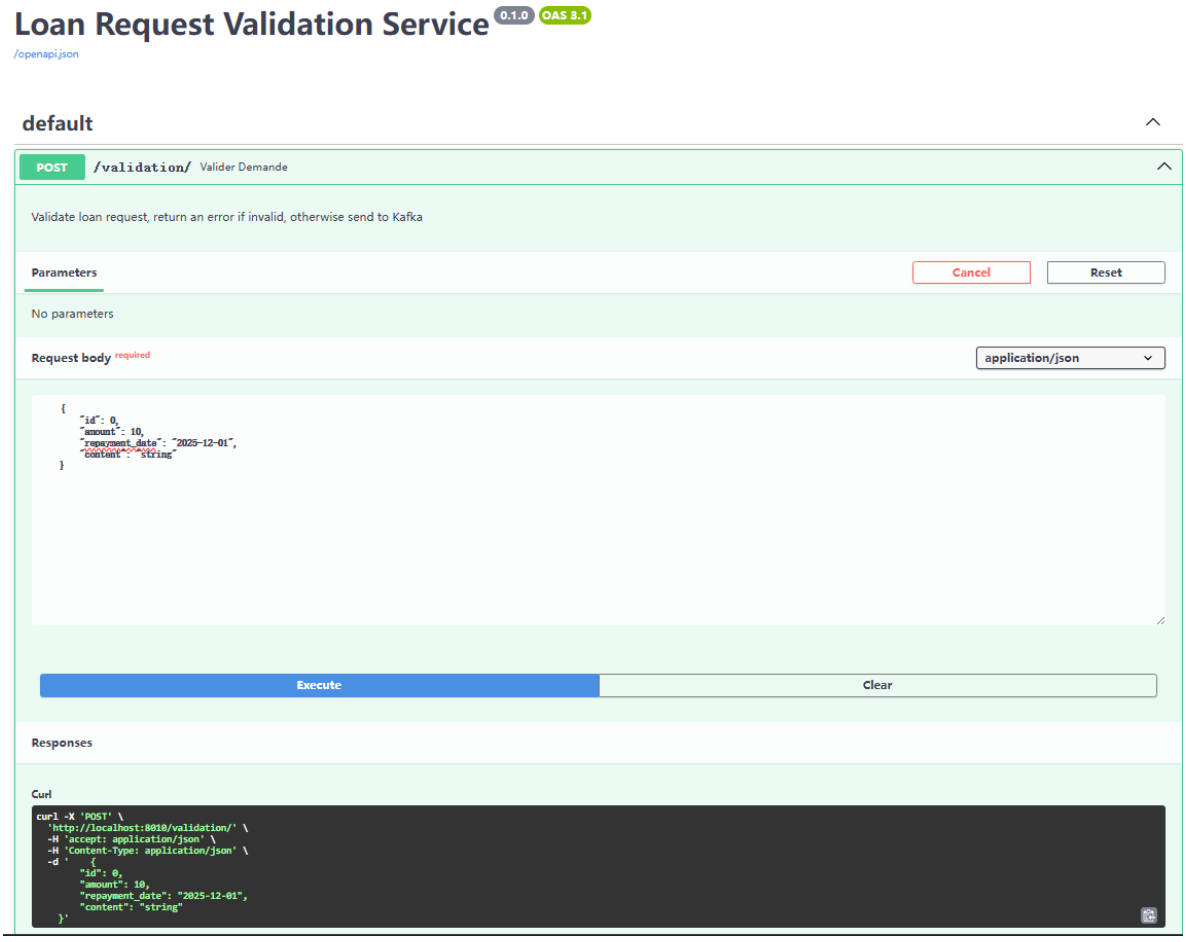

FIGURE 12 – Validation Endpoint Example