UVSQ

GESTION DE DONNÉES ET SERVICES DANS LE CLOUD

Rapport TP 2 : Mise en place d'un Système RAG avec FastAPI, Docker et Kubernetes Introduction

Mise en place d'un Système RAG avec FastAPI, Docker et Kubernetes

Élève :
KREIS Amaël
TRILLO Baptiste

Professeur :
TAHER Yehia

Projet réalisé pour le 28 janvier 2025

Table des matières

1 Introduction

Ce TP a pour objectif de concevoir un système **Retrieval-Augmented Generation (RAG)** en exploitant des technologies modernes telles que *FastAPI*, *Docker* et *Kubernetes*.

Un système RAG est composé de deux étapes principales :

— 1. **Récupération d'information (Retrieval)** : Cette étape consiste à rechercher des documents pertinents à partir d'une base de données ou d'un index.

— 2. **Génération de réponse (Generation)** : Un modèle, tel que *GPT*, est utilisé pour générer une réponse enrichie basée sur les documents récupérés.

Ce système repose sur une architecture à base de **microservices**, où chaque service est conçu pour réaliser une tâche spécifique, facilitant ainsi la modularité et l'évolutivité.

2 Architecture services

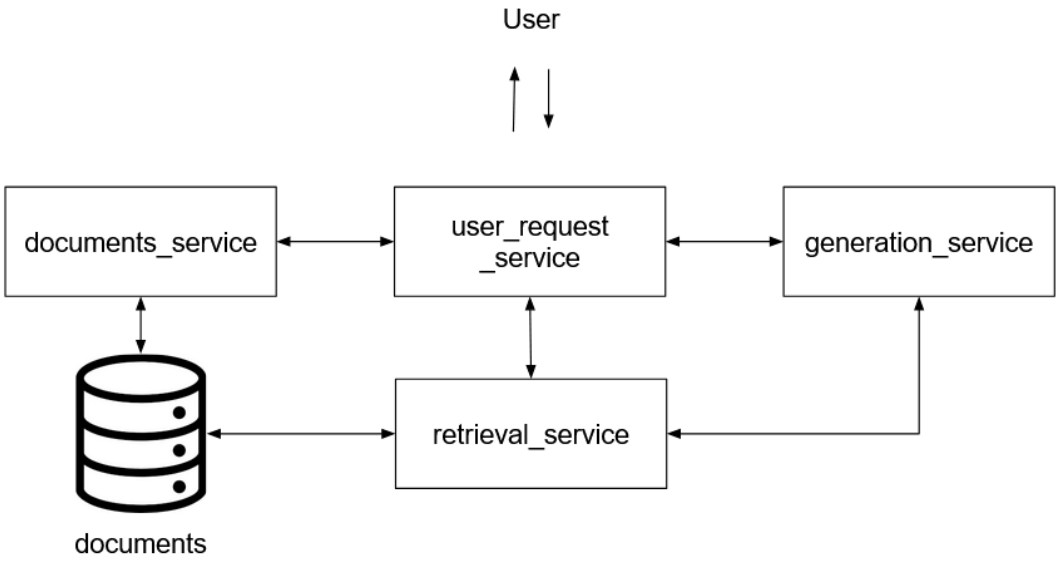

FIGURE 1 – Architécture des services

Il y a 4 services au total :

- **documents_service** : Ce service s'occupe de l'ajout et du retrait de documents dans la base de donnée, il permet aussi de voir tous les documents disponibles
- **retrival_service** : Ce service permet de recupérer à partir d'un prompt de l'utilisateur, les 5 documents les plus pertinant a l'aide de la similarité cosinus
- **generation_service** : Ce service prend en entré un prompt et des embbedings pour génerer la réponse la plus pertinante
- **user_request_service** : Ce service va orchestrer tous les autres services, il va prendre les requêtes de l'utilisateur et va s'occuper de faire les bonnes requêtes et d'envoyer les bonnes informations

3 Description des services

3.1 documents_service

- **Ajout de documents** (`POST /documents`) : Les utilisateurs peuvent ajouter un document via une requête POST à l'API en fournissant un document dans le corps. Le contenu du document est transformé en un vecteur d'embedding à l'aide du modèle `sentence-transformers/all-MiniLM-L6-v2`, et un identifiant unique est généré pour le document. Ce dernier est ensuite sauvegardé dans un fichier JSON.
- **Récupération de documents** (`GET /documents/{doc_id}`) : Les utilisateurs peuvent récupérer un document spécifique en fournissant son identifiant.
- **Suppression de documents** (`DELETE /documents/{doc_id}`) : Un document peut être supprimé à l'aide de son identifiant. Une fois supprimé, la base de données est mise à jour.
- **Liste des documents** (`GET /documents`) : Une requête GET permet de lister tous les documents disponibles, sans inclure leurs embeddings.

3.2 retrieval_service

- Une requête est reçue au point d'entrée POST `/retrieve_documents`. La requête doit contenir un texte de recherche (`query_text`).
- Les étapes suivantes sont réalisées pour retourner les résultats :
  1. Un vecteur d'embedding est généré pour le texte de la requête à l'aide du modèle **SentenceTransformer**.
  2. Les similarités cosinus entre l'embedding de la requête et les embeddings des documents chargés sont calculées à l'aide de **scikit-learn**.
  3. Les documents sont triés par ordre décroissant de similarité.
  4. Les cinq documents les plus pertinents sont extraits et renvoyés, incluant uniquement leur identifiant et leur contenu.

3.3 generation_service

Nous n'avons pas vraiment pu faire de RAG car l'api GPT est payante et nous n'avons pas de pc assez puissant pour faire tourner les LLM open source. Nous avons donc utilisé un modèle plus petit et beaucoup plus rapide mais ce dernier ne prend pas en charge le RAG, nous lui donnons donc le contenu des documents en temps que context dans le prompt pour qu'il génère une réponse en prenant en compte ces derniers.

- Une requête POST au point d'entrée `/generate_response` doit contenir les données suivantes :
  - `query` : une chaîne de caractères représentant la question posée.
  - `documents` : une liste de dictionnaires où chaque dictionnaire inclut un champ `content` représentant le texte d'un document.

Génération de réponse :

- Les contenus des documents fournis dans la requête sont concaténés pour former un contexte global.
- Une chaîne d'entrée est préparée sous la forme suivante : `Use the context to answer the question, question: <question> context: <context>`.
- Cette entrée est transformée en vecteurs numériques (*tokenization*) à l'aide du *tokenizer*.
- Une réponse est générée à partir du modèle `T5-small` avec les paramètres suivants :
  - `max_length=250` : la longueur maximale de la réponse.
  - `repetition_penalty=2.0` : pénalité pour limiter les répétitions dans la réponse.

- `length_penalty=1.5` : favorise des réponses plus longues et détaillées.
- `num_beams=5` : utilise une recherche par faisceau (*beam search*) pour améliorer la qualité de la génération.

— La réponse générée est décodée en texte clair et renvoyée au client dans un format JSON :
- `{"response": <réponse générée>}`.

3.4 user_request_service

3.4.1 Ajout d'un document

Le point d'entrée `POST /add_document` permet d'ajouter un document à la base de données des documents. Le contenu du document est envoyé au service de gestion des documents via une requête HTTP `POST`.

- `content` : Le contenu du document à ajouter.
- Le service renvoie une réponse JSON si l'ajout est réussi, sinon une exception est levée en cas d'erreur.

3.4.2 Lister les documents

Le point d'entrée `GET /list_documents` permet de lister tous les documents enregistrés. Une requête `GET` est envoyée au service de gestion des documents pour obtenir la liste des documents disponibles.

3.4.3 Suppression d'un document

Le point d'entrée `DELETE /delete_document/doc_id` permet de supprimer un document en fonction de son identifiant. Une requête `DELETE` est envoyée au service de gestion des documents pour supprimer le document spécifié.

3.4.4 Récupérer des documents pertinents

Le point d'entrée `POST /retrieve` permet d'envoyer une requête de l'utilisateur à un service de récupération des documents (`RETRIEVE_URL`). Le service renvoie une liste de documents pertinents en fonction de la question fournie.

3.4.5 Traitement d'une requête utilisateur

Le point d'entrée `POST /process_query` permet de traiter une requête utilisateur en deux étapes :

1. **Récupération des documents** : La question de l'utilisateur est envoyée au service de récupération pour obtenir une liste de documents pertinents.
2. **Génération de la réponse** : Les documents récupérés sont envoyés au service de génération pour créer une réponse pertinente à la question de l'utilisateur.

4 Docker

4.1 Dockerfile

Le `Dockerfile` ci-dessous est utilisé pour créer une des image Docker permettant de déployer un service Python basé sur **FastAPI**. Tous les Dockerfile ont la meme architecture, voici les étapes définies dans le `Dockerfile` :

4.1.1 Base de l'image

— `FROM python:3.9-slim` : L'image de base est une version minimale de Python 3.9, appelée `python:3.9-slim`. Cette image offre un environnement léger et optimisé pour les applications Python.

4.1.2 Installation des dépendances

— `RUN pip install -no-cache-dir fastapi uvicorn sentence-transformers python-dotenv` : Cette commande installe les bibliothèques nécessaires pour exécuter le service, sans conserver les fichiers temporaires (`-no-cache-dir`) :
  — **FastAPI** : Framework pour créer l'API web.
  — **uvicorn** : Serveur ASGI pour exécuter l'application FastAPI.
  — **sentence-transformers** : Bibliothèque pour la génération d'embeddings de texte.
  — **python-dotenv** : Utilitaire pour charger les variables d'environnement à partir d'un fichier `.env`.

4.1.3 Copie des fichiers nécessaires

— `COPY documents_service.py /app/` : Le fichier `documents_service.py` contenant l'application FastAPI est copié dans le répertoire `/app/` du conteneur.
— `COPY .env /app/` : Le fichier `.env` contenant les variables d'environnement est également copié dans le répertoire `/app/`.

4.1.4 Configuration du répertoire de travail

— `WORKDIR /app` : Le répertoire de travail du conteneur est défini sur `/app`, ce qui signifie que toutes les commandes suivantes seront exécutées dans ce répertoire.

4.1.5 Exposition du port

— `EXPOSE 8000` : Le port 8000 est exposé pour permettre l'accès à l'application FastAPI via ce port.

4.1.6 Commande de démarrage

— `CMD ["uvicorn", "documents_service:app", "-host", "0.0.0.0", "-port", "8000"]` : Cette commande est exécutée au démarrage du conteneur. Elle lance `uvicorn`, un serveur ASGI pour exécuter l'application FastAPI. L'application est accessible sur l'adresse `0.0.0.0` (toutes les interfaces réseau) et le port 8000.

4.2 Docker-compose

Le fichier `docker-compose.yml` ci-dessous permet de définir et de gérer plusieurs microservices dans un environnement Docker. Ce fichier spécifie les services à déployer, leurs configurations respectives, ainsi que la manière dont ils interagissent entre eux. Voici une description détaillée

de chaque service et des configurations.

Services définis :

4.2.1 documents_service

- `build:` Le service est construit à partir du contexte actuel (`context: .`) et d'un `Dockerfile` spécifique (`Dockerfile.documents_service`).
- `ports:` Le service expose le port 8000 à l'extérieur du conteneur, permettant l'accès via le même port sur l'hôte (`8000:8000`).
- `networks:` Le service est connecté au réseau `microservices_net`, permettant la communication avec d'autres services sur le même réseau.
- `volumes:` Le fichier `documents.json` est monté en tant que volume pour persister les données de documents, ce qui permet de conserver l'état des documents même après un redémarrage du conteneur.

4.2.2 retrieval_service

- `build:` Le service est également construit à partir du contexte actuel avec un `Dockerfile` dédié (`Dockerfile.retrieval_service`).
- `ports:` Ce service expose le port 8001, accessible à l'extérieur du conteneur sur le même port (`8001:8001`).
- `networks:` Le service est également connecté au réseau `microservices_net` pour communiquer avec les autres services.
- `volumes:` Comme pour `documents_service`, un volume est monté pour persister le fichier `documents.json`.

4.2.3 generation_service

- `build:` Ce service est construit à partir du contexte actuel avec un `Dockerfile` dédié (`Dockerfile.generation_service`).
- `ports:` Le service expose le port 8002, accessible à l'extérieur sur le même port (`8002:8002`).
- `networks:` Ce service est également connecté au réseau `microservices_net`.

4.2.4 user_request_service

- `build:` Le service est construit à partir du contexte actuel avec un `Dockerfile` dédié (`Dockerfile.user_request_service`).
- `ports:` Ce service expose le port 8003, permettant l'accès via ce port sur l'hôte (`8003:8003`).
- `networks:` Il est connecté au même réseau `microservices_net` pour assurer la communication avec les autres services.
- `depends_on:` Ce service dépend des services `documents_service`, `retrieval_service`, et `generation_service`, ce qui signifie qu'il attendra que ces services soient opérationnels avant de démarrer.

5 Kubernetes

Kubernetes est une plateforme open source destinée à automatiser le déploiement, la mise à l'échelle et la gestion des applications conteneurisées. Initialement développé par Google, Kubernetes a été adopté par la Cloud Native Computing Foundation (CNCF) et est devenu un standard de facto pour orchestrer les conteneurs.

5.1 Objectif et Utilité

Dans les environnements modernes, les applications sont de plus en plus souvent déployées sous forme de conteneurs, par exemple à l'aide de Docker. Si un ou deux conteneurs peuvent être gérés manuellement, la situation se complique lorsque des centaines ou des milliers de conteneurs doivent être supervisés. C'est là que Kubernetes intervient.
L'objectif principal de Kubernetes est de fournir une infrastructure unifiée pour gérer des conteneurs à grande échelle. Il facilite :

- Le déploiement cohérent des applications sur différents environnements (cloud, on-premise, ou hybrides).
- La haute disponibilité des services grâce à des mécanismes automatiques de redémarrage, remplacement, et mise à l'échelle des conteneurs.
- La répartition de charge et la gestion des ressources, garantissant une utilisation efficace de l'infrastructure.
- L'automatisation des mises à jour et des rollbacks pour un déploiement sécurisé et fiable.

5.2 Principes de Fonctionnement

Kubernetes repose sur une architecture de type maître-esclave. Voici une vue d'ensemble de ses composants principaux :

- **Control Plane** : Ce composant gère l'ensemble du cluster. Il inclut l'API Server (interface principale), l'Etcd (base de données clé-valeur pour stocker l'état du cluster), et le Scheduler (responsable d'assigner les tâches aux nœuds).
- **Nœuds (Workers)** : Les nœuds sont des machines physiques ou virtuelles qui exécutent les conteneurs. Chaque nœud comprend un kubelet (agent principal), un runtime de conteneur (comme Docker ou containerd), et un proxy réseau.
- **Pods** : Les conteneurs sont groupés en unités appelées "pods", qui représentent la plus petite unité déployable dans Kubernetes.

5.3 Utilisation de Kubernetes

Déploiement d'Applications : Pour utiliser Kubernetes, on commence par décrire l'état désiré d'une application dans des fichiers YAML ou JSON appelés "manifests". Par exemple, un manifest typique inclura la définition des pods, des services, et des règles de mise à l'échelle. Une fois les fichiers appliqués via la commande `kubectl apply`, Kubernetes orchestre automatiquement l'état demandé.

Mise à l'Échelle Automatique : Kubernetes surveille constamment l'utilisation des ressources (CPU, RAM) des pods. En fonction de règles prédéfinies, il peut automatiquement ajouter ou retirer des pods pour gérer la charge.

Mises à Jour et Rollbacks : Les déploiements peuvent être mis à jour de manière progressive grâce aux stratégies comme les rolling updates, minimisant les interruptions. En cas de problème, Kubernetes permet un rollback automatique vers une version précédente.

5.4 Avantages pour les Développeurs et les Administrateurs

Kubernetes simplifie la gestion des applications complexes. Les développeurs peuvent se concentrer sur l'écriture de code, tandis que les administrateurs bénéficient d'une solution robuste pour assurer la résilience, la scalabilité, et la sécurité des services. En résumé, Kubernetes est une solution incontournable pour tout environnement nécessitant une orchestration fiable et flexible des conteneurs.

5.5 Déploiement de Kubernetes

Tout les services sauf `user-request-service` sont en CloudIP car ils n'ont pas besoin d'etre vu depuis l'exterieur. Pour `user-request-service`, ce dernier est de type NodePort car il doit être vu depuis l'exterieur pour permettre a l'utilisateur de faire des requêtes.

5.5.1 deployment.yaml

Le fichier `deployment.yaml` déploie un conteneur `documents-service` sur un cluster Kubernetes. Le service est configuré pour :

1. Utiliser une image Docker stockée dans un registre local.
2. Exposer le port 8000 pour accepter les connexions.
3. Monter un fichier `documents.json` à partir d'un `ConfigMap` pour la persistance des documents.
4. Définir des variables d'environnement pour la configuration du service.
5. Garantir que le pod redémarre en cas d'échec.

Le `ConfigMap` assure la persistance du fichier `documents.json` même après des redémarrages du conteneur.

5.5.2 service.yaml

Le fichier `service.yaml` ci-dessous est utilisé pour exposer le service `documents-service` à l'extérieur du cluster Kubernetes. Ce fichier définit un objet de type `Service` qui permet d'accéder à l'application exécutée dans les pods du déploiement `documents-service`. Voici une explication détaillée de chaque section de ce fichier de configuration de service.

Définition du service :

- `apiVersion: v1` : Cette ligne définit la version de l'API Kubernetes à utiliser pour ce service. Ici, nous utilisons la version `v1`, qui est la version stable pour la création de services.
- `kind: Service` : Le type de ressource Kubernetes est un `Service`, qui permet d'exposer un ensemble de pods à l'extérieur du cluster ou à d'autres services à l'intérieur du cluster.
- `metadata:` : Cette section contient les informations de métadonnées du service, notamment :
  - `name: documents-service` : Le nom du service est `documents-service`, ce qui permet de l'identifier dans le cluster.

Spécifications du service :
La section `spec` définit la configuration du service, y compris les ports et le sélecteur pour identifier les pods associés au service.

- `ports:` : Cette section définit les ports utilisés par le service.
  - `name: "8000"` : Le nom du port est `8000`, ce qui est un identifiant arbitraire pour ce port dans la configuration.

  - `port: 8000` : Le service est accessible sur le port 8000 à l'extérieur du cluster Kubernetes.
  - `targetPort: 8000` : Le `targetPort` spécifie le port auquel le service doit rediriger les requêtes, ici il s'agit également du port 8000, qui est le port du conteneur dans lequel le service `documents-service` est en cours d'exécution.
- `selector:` : Le sélecteur est utilisé pour faire correspondre le service aux pods qui doivent être exposés. Ici, il est configuré pour sélectionner les pods qui possèdent l'étiquette `io.kompose.service: documents-service`, ce qui permet d'associer ce service aux pods créés par le déploiement `documents-service`.