1. Objectifs du projet

Contexte

L'entreprise **Puls-Events**, spécialisée dans le développement de solutions numériques pour la culture, souhaite enrichir sa plateforme avec un **assistant intelligent capable de recommander des événements culturels**. L'ambition est de permettre aux utilisateurs de poser des questions en langage naturel et d'obtenir des réponses personnalisées et contextualisées.

Problématique

Aujourd'hui, les plateformes de recherche d'événements reposent essentiellement sur des filtres classiques (mots-clés, lieu, date), qui manquent de souplesse et ne reflètent pas toujours l'intention réelle de l'utilisateur. Un système **RAG (Retrieval-Augmented Generation)** permet de dépasser ces limites en combinant :

- une recherche sémantique sur une base vectorielle d'événements,
- et la génération d'une réponse naturelle grâce à un modèle de langage avancé.

Cela répond directement au besoin métier de Puls-Events : **proposer une expérience utilisateur fluide, intuitive et engageante**, tout en valorisant la richesse de ses données événementielles.

Objectif du POC

Ce projet consiste à développer un **Proof of Concept (POC)** démontrant :

- la **faisabilité technique** de l'approche RAG,
- sa **pertinence métier** pour le domaine culturel,
- et sa **performance** en termes de qualité des réponses générées.

L'API livrée doit permettre de poser des questions simples ("Quels concerts de musique classique en avril 2025 à Paris ?") et de recevoir une réponse enrichie, claire et sourcée.

Périmètre

Le POC se concentre sur un périmètre restreint afin de valider le concept :

- **Zone géographique** : Paris

- **Période** : événements en 2025

- **Source de données** : API publique OpenAgenda (https://public.opendatasoft.com/explore/dataset/evenements-publics-openagenda/table/?disjunctive.keywords_fr&disjunctive.location_region&disjunctive.location_countrycode&disjunctive.location_department&disjunctive.location_city&dataChart=eyJxdWVyaWVzIjpbeyJjb25maWciOnsiZGF0YXNldCI6ImV2ZW5lbWVudHMtcHVibGljcy1vcGVuYWdlbmRhIiwib3B0aW9ucyI6eyJkaXNqdW5jdGl2ZS5rZXl3b3Jkc19mciI6dHJ1ZSwiZGlzanVuY3RpdmUubG9jYXRpb25fcmVnaW9uIjp0cnVlLCJkaXNqdW5jdGl2ZS5sb2NhdGlvbl9jb3VudHJ5Y29kZSI6dHJ1ZSwiZGlzanVuY3RpdmUubG9jYXRpb25fZGVwYXJ0bWVudCI6dHJ1ZSwiZGlzanVuY3RpdmUubG9jYXRpb25fY2l0eSI6dHJ1ZSwicmVmaW5lLnVwZGF0ZWRhdCI6IjIwMjUiLCJyZWZpbmUubG9jYXRpb25fY2l0eSI6IlBhcmlzIn19LCJjaGFydHMiOlt7ImFsaWduTW9udGgiOnRydWUsInR5cGUiOiJsaW5lIiwiZnVuYyI6IkFWRyIsInlBeGlzIjoiYWdlX21pbiIsInNjaWVudGlmaWNEaXNwbGF5Ijp0cnVlLCJjb2xvciI6IiNGRjUxNUEifV0sInhBeGlzIjoibGFzdGRhdGVfZW5kIiwibWF4cG9pbnRzIjoiIiwidGltZXNjYWxlIjoieWVhciIsInNvcnQiOiIifV0sImRpc3BsYXlMZWdlbmQiOnRydWUsImFsaWduTW9udGgiOnRydWV9&location=6,48.21003,7.04224&basemap=jawg.light)

- **Types d'événements** : divers (concerts, expositions, ateliers…), sans distinction thématique initiale

2. Architecture du système

2.1 Vue d'ensemble

Le système repose sur une architecture **RAG (Retrieval-Augmented Generation)** articulée autour de trois briques principales :

1. **Prétraitement et indexation** : les événements culturels provenant de la base OpenAgenda sont collectés, nettoyés, enrichis, puis convertis en vecteurs grâce à un modèle d'embedding. Ces vecteurs sont stockés dans une base de données vectorielle FAISS, optimisée pour la recherche par similarité.

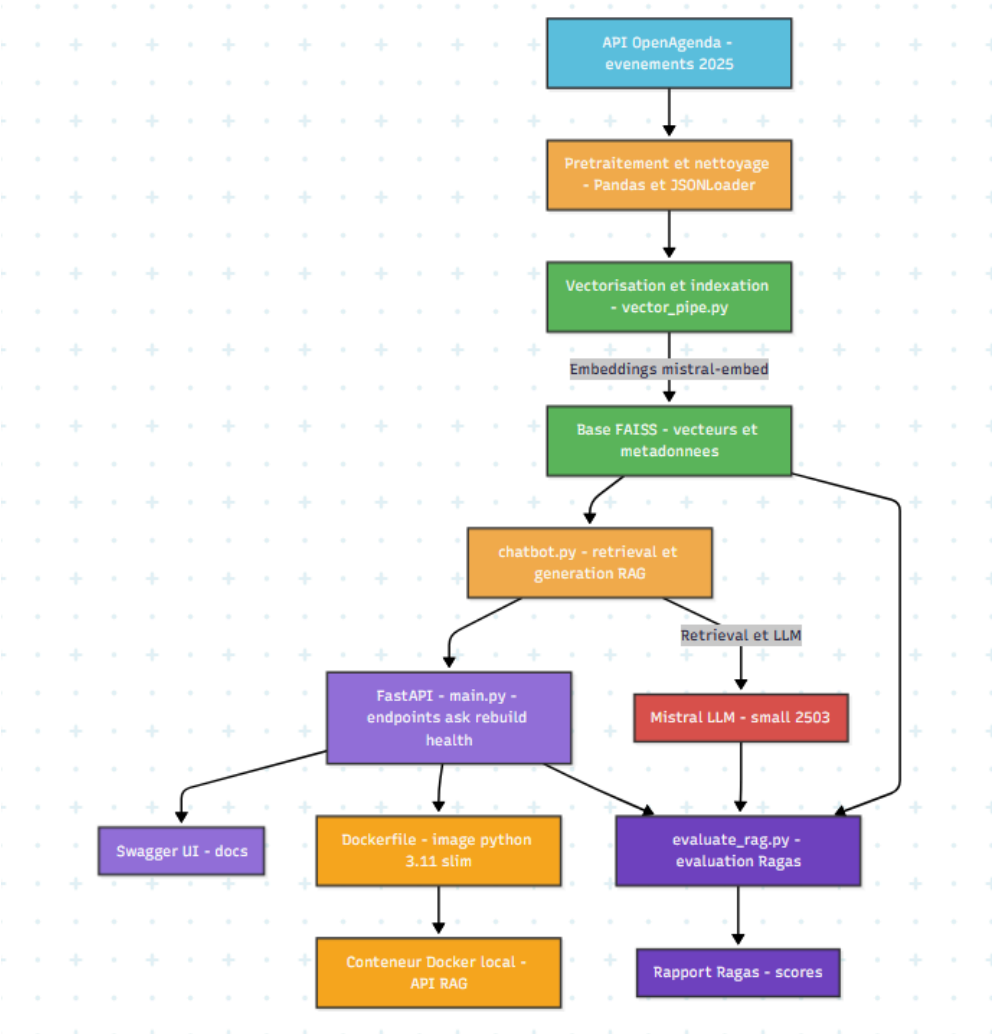
2. **Moteur de recherche et génération** : lorsqu'un utilisateur pose une question, celle-ci est vectorisée, puis comparée aux vecteurs stockés dans FAISS. Les événements les plus pertinents sont extraits et fournis en contexte au modèle de langage (LLM Mistral), qui génère une réponse fluide et synthétique.

3. **API REST** : une API basée sur FastAPI expose le système aux utilisateurs métier.

2.2 Conteneurisation et exécution

- L'API est conteneurisée avec **Docker**.
- Le `Dockerfile` définit une image basée sur `python:3.11-slim`.
- Les dépendances (LangChain, FAISS, FastAPI, Mistral client, Ragas, etc.) sont installées via `requirements.txt`.
- Le service est lancé avec Uvicorn sur `0.0.0.0:8000`.
- Le conteneur peut être exécuté localement avec la commande :
    - `docker build -t rag-api .`
    - `docker run -p 8000:8000 --env-file .env rag-api`

2.3 Schéma UML



2.4 Légende de l'architecture du système RAG

- **API OpenAgenda** : source externe contenant les événements publics culturels (titre, description, lieu, date, etc.), récupérés automatiquement via un appel HTTP.

- **ingest_openagenda.py** : script d'extraction et de nettoyage initial des données OpenAgenda (filtrage sur Paris et 2025, suppression des doublons, normalisation du texte).

- **vector_pipe.py** : pipeline de transformation du texte en vecteurs sémantiques grâce au modèle *Mistral-embed*, puis création et sauvegarde de l'index **FAISS** pour la recherche rapide.

- **FAISS (Vector Database)** : base de données vectorielle permettant la recherche par similarité sémantique entre les questions utilisateur et les événements indexés.

- **chatbot.py** : cœur du système RAG. Il interroge FAISS via LangChain, construit le contexte, et utilise le modèle **Mistral LLM** pour générer des réponses en langage naturel.

- **Mistral LLM** : grand modèle de langage utilisé pour comprendre la question et formuler une réponse cohérente et fluide à partir du contexte extrait.

- **api/main.py (FastAPI)** : interface REST exposant deux endpoints principaux — /ask pour poser une question et /rebuild pour régénérer l'index FAISS.

- **Docker** : conteneurise l'ensemble du projet (scripts, modèles, API) afin de garantir une exécution locale stable, reproductible et facilement partageable.
- **evaluate_rag.py** : script d'évaluation des réponses générées par le chatbot/api en utilisant la bibliothèque RAGAS

3. Choix technologiques et modèles utilisés

FAISS (Facebook AI Similarity Search)

Nous avons choisi **FAISS** comme base vectorielle car elle est :

- optimisée pour la recherche par similarité sur de grands volumes de données,

- rapide et efficace pour le traitement de plusieurs milliers de vecteurs,

- facilement intégrable via LangChain.

Mistral AI

Deux modèles fournis par **Mistral** ont été utilisés :

- **mistral-embed** pour générer les vecteurs (embeddings) des textes et des questions, garantissant une bonne représentation sémantique,
- **mistral-small-2503** (et ponctuellement `mistral-large`) pour la génération des réponses, offrant un équilibre entre performance et coût d'exécution.

LangChain

LangChain a été utilisé comme framework d'orchestration afin de :

- gérer le découpage des documents en chunks,
- interagir avec FAISS,
- connecter les embeddings et le LLM de manière fluide,
- construire facilement une chaîne RAG (retrieval + génération).

FastAPI

Le choix de **FastAPI** pour exposer l'API s'explique par :

- sa simplicité d'utilisation,
- sa rapidité d'exécution avec Uvicorn,
- la génération automatique de documentation Swagger (`/docs`),
- son adoption répandue dans des environnements de production.

Docker

La conteneurisation via **Docker** permet :

- de rendre le système portable et indépendant de l'environnement local,
- de simplifier le déploiement et la démonstration,
- d'assurer une reproductibilité lors des tests et de la soutenance.

4. Préparation et vectorisation des données

4.1 Source des données

Les événements proviennent de l'**API publique OpenAgenda**, qui met à disposition un catalogue riche d'événements culturels et professionnels.
Pour ce POC, nous avons restreint le périmètre à :

- la **ville de Paris**,
- les **événements de 2025** (période récente et pertinente pour la démonstration).

Les données ont été exportées sous format **JSON** et contiennent pour chaque événement :

- un identifiant unique,
- un titre,
- une description courte et longue,
- des mots-clés,
- des informations de lieu (ville, région, coordonnées),
- des dates de début et de fin,
- un lien vers la page officielle.

4.2 Nettoyage et structuration

Le prétraitement des données a été réalisé avec **Pandas** :

- suppression des doublons et des événements hors périmètre (autres villes ou autres années),
- gestion des valeurs manquantes (par exemple, certaines descriptions absentes),
- concaténation des colonnes **titre** et **description longue** pour obtenir un texte enrichi,

- création d’un format propre (JSON/CSV) avec les champs utiles : titre, description enrichie, dates, lieu, lien.

Un contrôle qualité a ensuite été effectué avec des tests unitaires (`pytest`) afin de vérifier :

- que tous les événements appartiennent bien à Paris,
- que toutes les dates de début concernent l’année 2025,
- que chaque événement dispose d’un identifiant et d’un titre.

4.3 Découpage en chunks

Afin d’améliorer la recherche sémantique, les textes enrichis (titre + description) ont été découpés en **chunks de 500 caractères avec un chevauchement de 50**.
Ce choix permet :

- d’éviter que des descriptions longues dépassent la limite du modèle d’embedding,
- de préserver le contexte sur les zones de chevauchement,
- d’augmenter la granularité lors des recherches (un utilisateur peut poser une question précise qui correspond à une partie seulement d’une description).

4.4 Vectorisation avec Mistral

La vectorisation a été effectuée grâce au modèle **mistral-embed**, proposé par Mistral AI.

- Chaque chunk textuel est transformé en vecteur numérique de grande dimension.
- Ces vecteurs capturent la **sémantique** du texte, permettant de retrouver des événements proches même si les mots utilisés diffèrent.
- La vectorisation est réalisée par lot de 50 chunks pour respecter les limites de l’API et éviter les erreurs de quota.

4.5 Indexation dans FAISS

Les vecteurs obtenus sont ensuite stockés dans un index **FAISS** :

- chaque vecteur est lié à ses **métadonnées** (titre, date, lieu, lien),
- l'index permet une recherche par similarité très rapide,
- l'ensemble est sauvegardé dans le dossier `data/faiss_store` pour être réutilisé par l'API sans devoir relancer toute la pipeline.

5. Choix du modèle NLP

5.1 Objectifs du modèle NLP

- Il génère des **embeddings** pour représenter les textes et les questions sous forme vectorielle.
- Il produit des **réponses en langage naturel** à partir du contexte extrait par la base vectorielle.

Le choix du modèle devait donc répondre à deux contraintes principales :

1. Offrir une **bonne qualité sémantique** pour les embeddings et la génération.
2. Être **performant et rapide** dans un contexte de Proof of Concept.

5.2 Modèles utilisés

Deux modèles fournis par **Mistral AI** ont été intégrés :

- **mistral-embed** :
  Utilisé pour générer les embeddings des événements et des questions.
  Ce modèle est optimisé pour capturer la sémantique des textes et permet d'obtenir des représentations vectorielles robustes pour la recherche par similarité.
- **mistral-small-2503** (et ponctuellement **mistral-large-2411** lors de tests) :
  Utilisé pour la génération des réponses en langage naturel.
  Ce modèle offre un bon compromis entre **coût d'utilisation** et **qualité des réponses**, tout en respectant les contraintes de quotas d'API (limite d'une requête par seconde).

5.3 Justification du choix

Le recours aux modèles Mistral s'explique par plusieurs raisons :

- Leur **qualité reconnue** sur les tâches de génération et d'embedding.
- Leur **compatibilité avec LangChain**, facilitant l'orchestration des appels.
- Leur **souplesse d'utilisation** via API, permettant un déploiement simple sans infrastructure lourde.
- La possibilité d'évoluer : on peut facilement passer de `mistral-small` à `mistral-large` si le besoin métier impose des réponses encore plus détaillées.

5.4 Prompting

Afin d'orienter le modèle vers des réponses pertinentes et structurées, un **prompt de base** a été utilisé.
Celui-ci impose :

- de répondre uniquement à partir du **contexte fourni par FAISS**,
- de donner un résumé clair incluant : **titre, date, lieu et lien** de l'événement,
- de ne jamais inventer d'événements en dehors du contexte.

Ce prompting garantit la **fidélité au contexte** et la **lisibilité** des réponses.

5.5 Limites du modèle

Malgré ses performances, l'utilisation de Mistral présente certaines limites :

- **Quota d'utilisation** : la version actuelle impose une limite d'1 requête par seconde, ce qui ralentit la vectorisation de gros volumes de données.
- **Réponses variables** : comme tout modèle génératif, Mistral peut produire des formulations différentes d'une exécution à l'autre, ce qui influe légèrement sur la reproductibilité des résultats.
- **Absence de raisonnement multi-pas** : le modèle se contente de générer à partir du contexte fourni sans effectuer de raisonnement complexe ou multi-documents avancé.
- **Dépendance au contexte fourni** : si le chunking ou l'indexation FAISS est incomplet, le modèle ne peut pas inventer et doit se contenter de dire qu'aucun résultat n'existe.
- **Coût lié à l'API** : contrairement à un modèle open source déployé localement, l'usage de Mistral implique des coûts liés au nombre de tokens traités.

6. Construction de la base vectorielle

6.1 Découpage des documents

Les textes des événements (titre + description longue) ont été **découpés en chunks de 500 caractères avec un chevauchement de 50**.

6.2 Vectorisation

Chaque chunk a été transformé en vecteur numérique grâce au modèle **mistral-embed**.

- La vectorisation a été effectuée par lots de 50 chunks pour respecter les limites de l'API.
- Chaque vecteur est associé à ses métadonnées.

6.3 Métadonnées associées

Pour chaque chunk, les informations suivantes sont conservées :

- **id** : identifiant unique de l'événement,
- **title** : titre de l'événement,
- **url** : lien officiel OpenAgenda,
- **date_start** et **date_end** : bornes temporelles,
- **city** et **region** : localisation,
- **keywords** : thématiques associées.

Ces métadonnées permettent d'enrichir la réponse du chatbot et d'améliorer la pertinence lors de la recherche.

6.4 Indexation avec FAISS

Les embeddings ont été indexés dans une base **FAISS** :

- structure de recherche optimisée pour la similarité cosinus,
- stockage local dans `data/faiss_store`,
- possibilité de recharger l'index sans relancer la vectorisation.

6.5 Résultat

- Nombre d'événements initiaux : environ 2 149.
- Nombre total de chunks générés : 4 728.
- Base vectorielle prête pour une recherche rapide et réutilisable via l'API.

7. API et endpoints exposés

7.1 Framework utilisé

Le framework **FastAPI** a été retenu pour :

- sa simplicité de mise en place,
- sa performance grâce à Uvicorn,
- la génération automatique de la documentation interactive (**Swagger UI** disponible sur `/docs`).

7.2 Endpoints clés

- **`/ask`** *(POST)*
    - Rôle : permet à l'utilisateur de poser une question en langage naturel.
    - Fonctionnement :
        - La question est transmise au **retriever FAISS**,
        - Les chunks les plus pertinents sont extraits,

        - Le modèle **Mistral (mistral-small-2503 ou mistral-large-2411)** génère une réponse augmentée.
    - Retour :
        - une réponse textuelle,
        - la liste des sources avec leurs métadonnées (titre, date, ville, URL, page_content).

- **`/rebuild`** *(POST)*
    - Rôle : permet de **reconstruire la base vectorielle FAISS** à partir du fichier `events_clean.json`.
    - Utilité : recharger ou mettre à jour l'index sans redéployer tout le système.

- **`/health`** *(GET)*
    - Vérifie l'état de l'API.
    - Retourne un simple message `{"status": "ok"}` si tout est fonctionnel.

7.3 Format des requêtes/réponses

- **Requête `/ask`**

```
{
  "question": "Quels concerts de musique classique en avril 2025 à Paris ?"
}
```

- Réponse **`/ask`**

```
{
  "answer": "Concert de l'Ensemble Marani le 6 avril 2025 à Paris...",
  "sources": [
    {
      "title": "Concert de l'Ensemble Marani",
      "url": "https://openagenda.com/...",
      "date_start": "2025-04-06T17:30:00",
      "date_end": "2025-04-06T19:30:00",
      "city": "Paris",
      "region": "Île-de-France",
      "keywords": ["musique classique", "concert"],
      "page_content": "Description textuelle du chunk..."
```

```
      }
    ]
}
```

7.4 Exemple d’appel API

- En Python (requests)

```
import requests

resp = requests.post("http://127.0.0.1:8000/ask", json={"question": "concert jazz juin 2025"})
print(resp.json())
```

7.5 Tests effectués et documentés

- **Tests unitaires** avec pytest pour valider le nettoyage des données et la cohérence des dates.
- **Tests d’intégration** : Test de fonctionnement des 3 endpoints
- **Tests de performance** : vérification manuelle que la recherche et la génération se font en quelques secondes.

7.6 Gestion des erreurs et limitations

- Vérification que la question n’est pas vide (400 Bad Request).
- Gestion des erreurs internes du modèle ou de l’index (500 Internal Server Error).
- Protection : les clés API ne sont jamais exposées dans les réponses.
- Limitation : la reconstruction via /rebuild est un endpoint sensible à protéger si l’API était déployée publiquement.

8. Évaluation du système

8.1 Jeu de test annoté

Pour évaluer le système, un **jeu de test annoté** a été construit :

- **Nombre d’exemples** : 10 questions couvrant différents types d’événements (concerts, expositions, ateliers, événements sportifs, etc.).

- **Méthode d’annotation** : pour chaque question, une **réponse de référence (ground truth)** a été définie à partir des données disponibles dans events_clean.json.

    - Exemple :

```
{    "question": "Quels concerts de musique classique en avril 2025 à Paris ?",

    "ground_truth": "Concert de l'Ensemble Marani le 6 avril 2025 à 17h30 à l'Eglise du Val-de-Grâce"
```

8.2 Métriques d’évaluation

L’évaluation a été réalisée avec la bibliothèque **Ragas**, qui fournit des métriques adaptées aux systèmes RAG :

- **Answer Relevancy** : mesure la pertinence de la réponse par rapport à la question.

- **Faithfulness** : vérifie que la réponse est bien supportée par le contexte fourni (évite les hallucinations).

- **Context Precision** : proportion du contexte utilisé par rapport à ce qui est réellement nécessaire.

- **Context Recall** : proportion du contexte pertinent qui a effectivement été retrouvé.

8.3 Résultats obtenus

- **Analyse quantitative (scores globaux)** :
  Les résultats moyens sur le jeu de test sont les suivants :

    - Answer relevancy : **0.56**

    - Faithfulness : **0.59**

    - Context precision : **0.10**

    - Context recall : **0.18**

- Ces scores montrent que le système parvient à fournir des réponses pertinentes et globalement fidèles au contexte, mais que la **sélection et l'exploitation des bons chunks de contexte** restent limitées (précision et rappel faibles).

- **Analyse qualitative (exemples)** :

    - **Réussite** : pour les concerts de musique classique en avril 2025, le système a correctement identifié *le Concert de l'Ensemble Marani* le 6 avril.

    - **Limite** : pour certaines requêtes plus spécifiques (ex. *atelier culinaire en septembre 2025*), le système a répondu *"Aucun événement trouvé"*, alors que des événements pertinents étaient présents dans la base.

9. Recommandations et perspectives

9.1 Ce qui fonctionne bien

- Le système RAG est **fonctionnel de bout en bout** : ingestion des données, nettoyage, vectorisation avec Mistral, indexation dans FAISS, interrogation via un chatbot et exposition d'une API REST.

- Les réponses générées sont globalement **pertinentes et fidèles au contexte**, en particulier pour les événements bien représentés dans les données (ex. concerts, expositions).

- L'API **FastAPI** est opérationnelle, avec documentation Swagger intégrée, endpoints principaux (`/ask`, `/rebuild`, `/health`), et gestion des erreurs basiques.

- La conteneurisation avec **Docker** rend le système portable et facilement exécutable localement.

9.2 Limites du POC

- **Volumétrie** : le système a été testé sur un sous-ensemble (~2100 événements). En cas de passage à des millions d'événements, l'index FAISS et les embeddings nécessiteront une gestion plus avancée (sharding, vector DB managée).

- **Performance** : la vectorisation repose sur l'API Mistral avec une limite de requêtes par seconde, ce qui ralentit la construction de l'index.

- **Coût** : l'usage d'API propriétaires (Mistral) implique un coût en production si les appels sont nombreux.

- **Couverture thématique** : les résultats sont bons pour certains types d'événements (musique, expositions), mais limités pour d'autres (sport, culinaire).

9.3 Améliorations possibles

- **Amélioration des embeddings** : tester d'autres modèles (par ex. `sentence-transformers` en local ou des modèles plus spécialisés) pour mieux représenter les textes.

- **Optimisation de l'index** : utiliser une base vectorielle plus robuste (Pinecone, Weaviate, Milvus) avec support natif pour la scalabilité et la mise à jour incrémentale.

- **Meilleure gestion du contexte** : affiner le découpage en chunks et améliorer la sélection de contexte pour augmenter `context_precision` et `context_recall`.

- **Ajout de fonctionnalités** : filtrage par type d'événement, lieu, prix, ou public cible directement dans l'API.

- **Évaluation avancée** : automatiser les tests avec Ragas dans un pipeline CI/CD (GitHub Actions) pour surveiller la qualité en continu.

9.4 Passage en production

- Déployer l'API dans un environnement cloud (AWS, GCP, Azure) avec un orchestrateur comme **Kubernetes** pour la haute disponibilité.

- Intégrer un **système de monitoring** (Prometheus, Grafana) pour suivre les performances (latence, taux d'erreurs, qualité des réponses).

- Prévoir une **stratégie de mise à jour des données** (cron jobs pour recharger les événements OpenAgenda).

- Sécuriser l'API avec authentification et limitation des endpoints sensibles comme `/rebuild`.

10. Organisation du dépôt GitHub

Le dépôt du projet est structuré de manière claire afin de séparer les différents composants du système RAG et de faciliter sa compréhension, son exécution et sa maintenance.

Arborescence du dépôt

```
├── .git/                # Fichiers Git pour le suivi de version
├── api/                 # Contient le code de l'API FastAPI (main.py, endpoints)
├── data/                # Jeux de données nettoyés et index FAISS
├── docker/              # Fichiers liés à la configuration Docker (optionnel)
├── env/                 # Environnement virtuel Python (non versionné)
├── eval/                # Scripts et données pour l'évaluation (Ragas, jeux de test annotés)
├── notebook/            # Notebooks exploratoires (nettoyage, prototypage)
├── rag/                 # Cœur du système RAG (chatbot, vectorisation, ingestion)
├── scripts/             # Scripts utilitaires (ex. build_index.py pour reconstruire FAISS)
├── tests/               # Tests unitaires et fonctionnels (pytest)
├── .dockerignore        # Fichiers/dossiers ignorés lors de la construction Docker
├── .env                 # Variables d'environnement (clé API Mistral, config)
├── .gitignore           # Fichiers/dossiers ignorés par Git
├── Dockerfile           # Image Docker pour exécuter l'API localement
├── README.md            # Documentation principale du projet
├── requirements.txt     # Dépendances Python nécessaires au projet
```

Explication rapide des répertoires

- **api/** : contient le fichier `main.py`, qui expose l'API REST via FastAPI, avec les endpoints `/ask`, `/rebuild`, `/health`.

- **data/** : stocke les données brutes et nettoyées (`events_clean.json/csv`) ainsi que l'index vectoriel FAISS sauvegardé (`faiss_store/`).

- **docker/** : éventuellement dédié aux configurations Docker supplémentaires (compose, volumes).

- **env/** : répertoire local pour l'environnement virtuel Python (non versionné sur GitHub).

- **eval/** : inclut le fichier `eval_data.json` (jeu de test annoté) et les scripts d'évaluation avec Ragas (`evaluate_rag.py`, etc.).

- **notebook/** : notebooks Jupyter utilisés lors de la phase exploratoire et de nettoyage initial.

- **rag/** : cœur du projet RAG, avec les modules pour l'ingestion des données (`ingest_openagenda.py`), la vectorisation (`vector_pipe.py`), et le chatbot (`chatbot.py`).

- **scripts/** : scripts utilitaires exécutables, comme `build_index.py` pour reconstruire l'index FAISS avant le déploiement.
- **tests/** : répertoire contenant les tests unitaires et fonctionnels pour valider le nettoyage, la vectorisation et les endpoints.
- **fichiers racine (.env, requirements.txt, Dockerfile, README.md)** :
    - `.env` : stocke les variables sensibles (clé API Mistral).
    - `requirements.txt` : liste des dépendances Python.
    - `Dockerfile` : permet de construire l'image Docker exécutant l'API.
    - `README.md` : documentation projet (installation, usage, contexte).

11. Annexes (exemples)

🔹 Extraits du jeu de test annoté

Le fichier `eval/eval_data.json` contient les questions et les réponses de référence ("ground truth") utilisées pour l'évaluation avec **Ragas**.

Exemple :

"question": "Quels concerts de musique classique en avril 2025 à Paris ?",

"ground_truth": "Concert de l'Ensemble Marani le 6 avril 2025 à 17h30 à l'Eglise du Val-de-Grâce"

"question": "Y a-t-il un hommage à Beethoven en 2025 ?",

"ground_truth": "Hommage à Beethoven le 21 juin 2025 à 20h à la Mairie du 9e"

🔹 Prompt utilisé (RAG – génération de réponse)

Le **prompt système** défini dans `chatbot.py` guide le modèle pour générer des réponses uniquement à partir du contexte :

=> Tu es un assistant culturel qui recommande des événements uniquement à partir du CONTEXTE fourni ci-dessous.

Question : {question}

Contexte :

{context}

Consignes :

- Si le contexte contient des événements pertinents, donne un résumé clair avec :
  - titre(s) d'événement
  - date lisible
  - lieu
  - lien (si disponible)
- Si le contexte est vide ou ne contient rien de pertinent, réponds exactement :
  "Désolé, aucun événement trouvé correspondant à ta recherche."
- N'invente jamais d'événements ou d'informations extérieures.

🔹 Exemple de réponse JSON (endpoint `/ask`)

Requête :

POST http://127.0.0.1:8000/ask

{

"question": "Quels concerts de musique classique en avril 2025 à Paris ?"

}

Réponse :

{

"answer": "Voici les événements de musique classique en avril 2025 :\n\n- **Concert musique classique**\n - **Date :** Dimanche 6 avril 2025 à 17h30\n - **Lieu :** Non spécifié\n - **Entrée :** Libre",
"sources": [
{
"title": "Concert de l'Ensemble Marani",
"url": "https://openagenda.com/pci-ile-de-france/events/concert-de-lensemble-marani",
"date_start": 1743953400000,
"date_end": "2025-04-06T17:30:00+00:00",
"city": "Paris",
"region": "Île-de-France",
"keywords": "[]",
"page_content": "captiver tous les amateurs de musique classique. Ne manquez pas cette occasion de vivre un moment de pure émotion musicale. Le concert se déroulera le dimanche 6 avril 2025 à 17h30. Une entrée libre vous permet de profiter pleinement de cette performance sans contrainte."
}