Rapport technique – Assistant intelligent de recommandation d'événements culturels à Paris

1. Objectifs du projet

Contexte :
Puls-Events est une entreprise technologique qui développe une plateforme spécialisée dans la recommandation d'événements culturels personnalisés.

Notre mission est de mettre en place un nouveau chatbot intelligent capable de répondre à des questions utilisateurs sur les événements culturels à venir à Paris, en s'appuyant sur un système **RAG (Retrieval-Augmented Generation)**.
Le système permet aux utilisateurs d'interroger en langage naturel une base de données de 100 événements réels issus de l'API OpenAgenda et d'obtenir des réponses contextualisées et personnalisées.

- Source de données : **API OpenAgenda** (extraction de 100 événements ayant lieu à Paris)
- Architecture : **Pipeline RAG** complet (préprocessing, vectorisation, retrieval et génération)
- Déploiement : API REST **FastAPI** conteneurisée avec **Docker**
- Évaluation : Évaluation avec **RAGAS** sur un jeu de test de 12 questions annotées

---

2. Architecture du système

Schéma global (schéma UML) : voir PJ

- Données entrantes (API OpenAgenda)
- Prétraitement / embeddings / base vectorielle FAISS
- Intégration du LLM Mistral Large avec LangChain
- Exposition via API FastAPI

Technologies utilisées :

Étape	Choix technologique	Explication
Data Loader	Python (BeautifulSoup, JSON…)	Téléchargement des événements depuis OpenAgenda et preprocessing (nettoyage HTML, création d'un événement avec description enrichie, sauvegarde des événements propres au format JSON).
Index Builder	RecursiveCharacterTextSplitter	Chargement des données avec `data_upload.py`.
Embedding	Mistral Embed	1 — Chunking : découpage en morceaux de 800 caractères (overlap 100), obtention de 253 chunks. 2 — Vectorisation sémantique en 1024 dimensions. 3 — Indexation FAISS (création de la base vectorielle). 4 — Enregistrement de l'index FAISS et des métadonnées.
Vector Store (Retriever)	FAISS — Similarité cosinus	Charge l'index FAISS, `metadata.json`, l'embedding model. Permet la recherche d'événements culturels et renvoie les résultats sous forme exploitable par le chatbot.
Chatbot RAG	LangChain + Mistral Large (latest)	Pipeline RAG : combinaison retrieval + prompt + LLM. - `ChatbotRAG` : classe principale (VectorStore, LLM et pipeline). - `_setup_prompt` : construction du prompt système. - `_format_context` : conversion des événements en contexte lisible. - `log_for_ragas` : journalisation pour l'évaluation RAGAS.
API	FastAPI	Endpoints : `/health` (GET), `/ask` (POST), `/rebuild` (POST), `/rebuild/full` (POST). Documentation Swagger auto-générée.
Déploiement	Docker + docker-compose	Conteneurisation avec image optimisée.
Pipeline CI/CD	GitHub Actions	Tests automatisés et création d'image Docker à chaque push.

3. Préparation et vectorisation des données

Source de données :
Les données sont récupérées via l'API OpenAgenda, avec un filtre permettant de collecter les événements culturels pertinents (100 événements).
La requête API renvoie des champs structurés : titre, description, horaires, lieu, organisateur, etc.

Nettoyage :
Avant la vectorisation, le texte subit un pré-traitement :

- Suppression des balises HTML et caractères spéciaux
- Unification des encodages et normalisation Unicode
- Suppression des champs vides / doublons
- Restructuration du texte lorsque plusieurs champs descriptifs sont fournis séparément
  → Objectif : produire un texte cohérent, informatif et sans bruit lexical.

Chunking :
Les descriptions sont parfois longues ; le découpage en segments améliore la pertinence de la recherche sémantique.
Taille choisie : **800 caractères** pour conserver une granularité suffisante permettant à chaque chunk de porter un sens autonome.

Embedding :

- Modèle utilisé : **Mistral Embed** (vecteurs 1 024 dimensions)
- Principe : création d'un vecteur par chunk (traitement par batch)
- Format final : vecteurs stockés dans **FAISS** + métadonnées associées (titre, date, description, etc.)

---

4. Choix du modèle NLP

Modèle sélectionné : Mistral Large (latest) pour la génération des réponses.

Pourquoi ce modèle ?

- Excellente compréhension du français

- Coût maîtrisé pour une qualité similaire à d'autres modèles premium
- Très bonne compatibilité avec RAG (longs prompts sans perte de contexte)
- Intégration simple via API et backend Python

Prompt Engineering :
Le prompt système intègre :

- Le rôle (assistant culturel)
- Les instructions (répondre uniquement à partir du contexte fourni)
- La question utilisateur et le contexte provenant de FAISS
  Structure : **Question utilisateur → Contexte documentaire → Politique de réponse → Réponse finale**

Limites du modèle :

- Ne "sait" rien sans le contexte FAISS (strict RAG)
- Les données doivent rester synthétiques pour tenir dans la fenêtre de contexte
- Pas de support des contenus multimédias → uniquement texte

---

5. Construction de la base vectorielle

FAISS utilisé :
FAISS (Facebook AI Similarity Search), moteur de recherche vectorielle performant et open-source.

Stratégie de persistance :

- Index FAISS généré lors de l'ingestion puis sauvegardé
- Format : fichier binaire contenant l'index et les vecteurs
- Nommage : **faiss_index.bin** pour faciliter le rechargement automatique

Métadonnées associées :
Chaque vecteur possède un dictionnaire contenant :

- Titre de l'événement
- Date et horaire
- Lieu / adresse
- Description nettoyée
- Mots-clés relatifs à l'événement

---

6. API et endpoints exposés

Framework utilisé : FastAPI

Endpoints :

- **`/ask :`** question utilisateur → réponse du système
- **`/rebuild :`** chargement de la base vectorielle
- **`/rebuild/full :`** reconstruction complète de l'index si nécessaire

---

7. Évaluation RAGAS

Jeu de test : 12 questions annotées.

Métrique	Score	Interprétation
Faithfulness	0.90	Réponses très fidèles au contexte ; quasi-absence d'hallucinations.
Answer Correctness	0.31	Exactitude moyenne ; parfois à côté de la demande utilisateur.
Context Precision	0.43	Contexte fourni partiellement pertinent.
Context Recall	0.62	Une partie importante du contexte réellement utile est retrouvée.

Analyse :

- Points forts : très faible hallucination, recall satisfaisant.

- Points faibles : précision du contexte insuffisante, d'où des réponses incomplètes ou imprécises.

  → Le problème se situe principalement au niveau du **retrieval**, non du LLM.
  → Stratégies d'amélioration : chunking ajusté, optimisation FAISS, reranker, meilleur prompt.

---

8. Recommandations et perspectives

Ce qui fonctionne bien :

- Pipeline RAG complet et fonctionnel (ingestion → réponse)
- Très bonne fidélité des réponses (Mistral Large)
- Vector store FAISS performant
- API claire et modulaire

Limites du POC :

- Volumétrie limitée (253 vecteurs)
- Pas de mise à jour incrémentielle
- Coût dépendant du nombre d'appels au LLM
- Dépendance unique à l'API OpenAgenda

Améliorations possibles :

- Améliorer le retrieval (reranker, réglage de k, re-chunking)
- Améliorer la qualité des données vectorisées (ajout de champs structurants)
- Nouvelles fonctionnalités : filtres, historique utilisateur£
- Production : déploiement cloud, monitoring, scalabilité

---

9. Organisation du dépôt GitHub

```
P9-RAG-CHATBOT/
├── .github/workflows/           # CI (tests automatisés)
├── Ragas/                       # Scripts d'évaluation RAGAS
├── data/                        # Données récupérées / stockage local
├── tests/                       # Tests unitaires / fonctionnels
├── api.py                       # API REST
├── chatbot_rag.py               # Pipeline RAG
├── config.py                    # Paramètres du projet
├── data_upload.py               # Preprocessing des données
├── embedder.py                  # Génération des embeddings
├── index_builder.py             # Construction de l'index FAISS
├── vectore_store.py             # Vector store + recherche
├── dockerfile                   # Image Docker
├── docker-compose.yml           # Orchestration
├── pyproject.toml / poetry.lock # Dépendances
└── README.md                    # Documentation du projet
```

---

10. Conclusion

Le projet **P9-RAG-Chatbot** démontre la faisabilité et la pertinence d'un système RAG appliqué à la recommandation d'événements culturels à partir de données OpenAgenda. L'architecture conçue est claire, modulaire et évolutive, facilitant la maintenance, l'amélioration continue et le passage en production.
Les choix techniques réalisés — ingestion API, embeddings, vectorisation FAISS, pipeline RAG et génération via Mistral Large — sont cohérents avec les objectifs du POC. L'évaluation RAGAS confirme que le système produit des réponses fidèles (faithfulness ≈ 0.90), évitant les hallucinations.

Les scores plus faibles en answer correctness et context precision identifient le levier prioritaire d'amélioration : l'optimisation du retrieval. Le système est fonctionnel, conteneurisé, testé et documenté.
Il constitue ainsi une base solide pour une industrialisation future, avec des perspectives d'évolution réalistes : amélioration du retrieval, reranker, re-chunking, montée en charge des données, automatisation de l'ingestion et déploiement cloud.