Rapport technique — Assistant intelligent de recommandation d’événements (Puls-Events RAG)

# 1) Objectifs du projet

**Contexte (Puls-Events)**Offrir un assistant capable de répondre à des questions naturelles sur les événements culturels récents en France (2024–2025).

**Problématique (pourquoi RAG ?)**

Les LLM “génériques” n’ont pas les derniers événements.

Un pipeline RAG (Recherche + Génération) permet d’ancrer les réponses dans des données fraîches (OpenAgenda).

**Objectif du POC**

Faisabilité: ingestion → index vectoriel → réponse contextualisée.

Valeur: réponses concrètes (titre, ville, dates, URL).

Perf: temps de réponse raisonnable en local, et simplicité de déploiement (Poetry / Docker).

**Périmètre**

Zone géographique: France (filtres région/ville possibles).

Période: 2024–2025.

Données: dataset Opendatasoft evenements-publics-openagenda.

# 2) Architecture du système

**Flux logique**

Collecte → src/scripts/fetch\_data.py (OpenAgenda via Opendatasoft).

Nettoyage/normalisation → src/scripts/clean\_data.py (+ src/utils.py).

Indexation (chunking + embeddings) → src/scripts/build\_index.py → FAISS persistant (artifacts/index).

Question/Retrieval/Génération → src/scripts/query\_rag.py.

API → src/api/app.py (endpoints /ask, /rebuild).

Frontend → src/streamlit.py.

**Technos**

Python 3.10+, Poetry.

LangChain (embeddings, splitters, FAISS).

FAISS-CPU (vector store).

Mistral API ou Ollama (local) pour LLM/embeddings.

FastAPI / Uvicorn, Streamlit.

Docker (Ollama hors conteneur via host.docker.internal:11434).

# 3) Préparation et vectorisation des données

**Source & filtres**

fetch\_data.py interroge l’API Opendatasoft (dataset evenements-publics-openagenda).

Filtres via .env: REGION\_FILTER, CITY\_FILTER, DATE\_MIN, DATE\_MAX.

Auto-détection du champ date par src/openagenda\_client.py.

**Nettoyage / Normalisation**

clean\_data.py + utils.normalize\_records\_to\_df:

**Colonnes cibles harmonisées:** uid,title,description,conditions,keywords,url,city,region,start,end,text.

Concaténation en champ text (titre + description + conditions + mots-clés + méta).

Suppression de doublons (uid/titre/date).

**Sortie Parquet**: data/clean/events\_clean.parquet (+ version small possible).

**Chunking (pour l’index)**

build\_index.py utilise RecursiveCharacterTextSplitter (LangChain):

CHUNK\_SIZE=2200, CHUNK\_OVERLAP=40 (optimisé pour garder du contexte tout en limitant les tokens).

Ajout de tokens structurés dans le texte (e.g. city:paris, year:2024, season:ete, kw:concert) => meilleur rappel sémantique.

**Embeddings**

Backend configurable :

Ollama (langchain\_ollama.OllamaEmbeddings) avec OLLAMA\_MODEL (ex. mxbai-embed-large) et OLLAMA\_BASE\_URL.

Mistral (langchain\_mistralai.MistralAIEmbeddings) via MISTRAL\_EMBED\_MODEL.

Batching avec barre de progression et “flush” FAISS (paramètres: EMBED\_BATCH, FAISS\_FLUSH) pour stabilité/perf.

# 4) Choix du modèle NLP (génération)

**Backends**

Mistral API (SDK mistralai) — simple, qualité stable.

Ollama local (e.g. mistral, llama3.2) — offline, latence faible, contrôle GPU.

Critères

Coût (0 côté Ollama), latence locale, intégration LangChain, simplicité de bascule (radio dans Streamlit).

Prompting (structure)

query\_rag.py : System prompt anti-hallucination (“utiliser uniquement le contexte”), obligation de lister titre/ville/dates/URL.

Limites

Qualité dépend du contexte récupéré (garbage in, garbage out).

Modèles Ollama légers ≈ un peu moins précis que Mistral hébergé.

# 5) Construction de la base vectorielle

FAISS (langchain\_community.vectorstores.FAISS)

Création via FAISS.from\_embeddings(...) ou add\_embeddings(...) par flush réguliers.

Normalisation L2 activée.

Persistance & nommage

Répertoire: ${INDEX\_DIR} (par défaut artifacts/index).

Fichiers: index.faiss, index.pkl.

Métadonnées stockées

uid,title,url,city,region,postalcode,start,end,year,month,season,keywords,...

Utilisées pour filtrer/réaligner le ranking, et pour afficher les infos dans la réponse.

# 6) API & endpoints exposés

**Framework: FastAPI (src/api/app.py)**

Endpoints

**POST /ask**

Body: {"question": "Que faire à Paris cet été ?", "k": 5, "backend": "mistral|ollama"}

Réponse: { "answer": "...", "contexts": [...], "ranking":[...], "k": 5 }

**POST /rebuild**

Reconstruit l’index local (relance build\_index.py).

**GET /health**

return {"status": "ok"}

**Exemples d’appels**

PowerShell (Windows) :

irm http://127.0.0.1:8000/ask -Method POST -Body (@{question="Quels concerts à Paris ce week-end ?"; k=5} | ConvertTo-Json) -ContentType "application/json"

curl (bash) :

curl -s -X POST http://127.0.0.1:8000/ask \

-H "Content-Type: application/json" \

-d '{"question":"Quels concerts à Paris ce week-end ?","k":5}'

**Tests**

tests/test\_api\_smoke.py (API up + /docs).

tests/test\_index.py (présence de l’index).

**Gestion des erreurs**

Réseau Ollama: bascule localhost → 127.0.0.1 (Windows), ou en Docker host.docker.internal.

Index manquant: message explicite (“run clean-data / build-index”).

# 7) Évaluation du système

**Jeu de test**

tests/dataset.jsonl (Q/R de référence, petite taille pour démo).

**Métriques / Outils**

Ragas (démonstration src/scripts/ragas\_ollama\_demo.py) pour context\_precision / context\_recall.

Possibilité d’étendre avec Answer Correctness / Faithfulness.

Résultats (démo)

Exemple mesuré : context\_precision ≈ 1.0, context\_recall ≈ 0.83 (sur mini-jeu de test).

Quali : réponses listant correctement titre/ville/date/URL lorsque le contexte couvre la requête.

# 8) Recommandations & perspectives

**Ce qui fonctionne bien**

Pipeline simple et modulaire (scripts indépendants).

MMR (diversité) + tokens structurés dans le texte → meilleur rappel.

Bascule Mistral↔Ollama et Docker prêt (option A).

**Limites**

Pas d’“entity resolver” avancé (orthographes/tokens ville).

Index unique France (volume croissant = taille FAISS + temps d’embedding).

Qualité dépend des champs OpenAgenda.

**Améliorations**

Filtres sémantiques plus “forts” (ville/genre d’événement) côté requête.

Rerank léger (Cross-Encoder) après FAISS.

Planification: rebuild quotidien + cache des réponses fréquentes.

Auth/API key si mise en prod.



# 9. Organisation du dépôt GitHub

* puls-events-rag-poc/
* ├─ src/
* │  ├─ api/
* │  │  └─ app.py                # API FastAPI (/ask, /rebuild)
* │  ├─ scripts/
* │  │  ├─ fetch\_data.py         # 1. Récupérer CSV 2024–2025 (filtré)
* │  │  ├─ clean\_data.py         # 2. Nettoyage + Parquet
* │  │  ├─ build\_index.py        # 3. Embedding + FAISS
* │  │  ├─ query\_rag.py          # 4. Question en CLI (RAG)
* │  │  └─ ragas\_ollama\_demo.py  # 5. démonstration de raggas en utilisant ollama (Ragas-ready)
* │  ├─ openagenda\_client.py     # Client Opendatasoft (+ auto-détection du champ date)
* │  ├─ config.py                # Chargement .env
* │  └─ utils.py                 # Normalisation des colonnes → texte
* │  └─ streamlit.py             # BONUS : Frontend Opérationel
* ├─ tests/
* │  ├─ test\_api\_smoke.py        # Démarre l'API et check /docs
* │  ├─ test\_index.py            # Vérifie présence index
* │  └─ dataset.jsonl            # Exemple de dataset d'éval
* ├─ data/                       # (créé au runtime) (.parquet)
* ├─ artifacts/                  # (créé au runtime) (index faiss)
* ├─ scripts/
* │  └─ run\_api.sh               # Lance l'API (dev)
* ├─ .env.example
* ├─ pyproject.toml
* └─ README.md

# 10. Annexes (exemples)

https://github.com/Tom-Buzon/P7-Evnt-Rg.git