Concevez et déployez un système RAG — Rapport technique (POC)

Client (scénario) : Puls‑Events

Projet : Assistant de recommandation d’événements culturels (OpenAgenda → FAISS → LangChain + Mistral)

Auteur : Motasem Abu aqlumbz

Date :

# 1. Résumé exécutif

Ce Proof‑of‑Concept (POC) met en œuvre un assistant **RAG** (Retrieval‑Augmented Generation) qui recommande des événements culturels à partir du dataset public **OpenAgenda**. La solution combine :

* **Prétraitement** des données (ville = *Paris*, **fenêtre ≤ 1 an**, nettoyage HTML),
* **Indexation vectorielle** locale via **FAISS** avec **mistral‑embed**,
* **Orchestration** des composants via **LangChain**,
* **Génération** de réponses par **Mistral Chat** (fallback medium → large → small), contrainte à **répondre uniquement depuis le contexte** récupéré.

Deux évaluations complémentaires garantissent la qualité :

* **Retrieval** (rappel) sur 20 règles : **Recall@10 = 1.000**, **Recall@20 = 1.000**.
* **Génération** (exactitude) sur 15 questions FR annotées : **ExactMatch = 15/15**, **ContainsGold = 15/15**. Ces scores sont obtenus avec **MMR** (diversification du top‑k), une **normalisation** (accents/emoji/ponctuation) et un **snapping** du résultat du LLM vers les valeurs **canoniques** (titre/lieu/CP) présentes dans le contexte récupéré.

Le POC est **reproductible de bout en bout** (README, scripts et snapshot gelé) et prêt pour une **démo live**. Le rapport détaille l’architecture, les choix techniques, les résultats et les recommandations d’industrialisation.

# 2. Contexte & objectifs

**Contexte** — Puls‑Events souhaite tester l’intégration d’un chatbot capable de **fournir des recommandations d’événements précises et contextualisées**, filtrables par localisation et période, à partir de la source OpenAgenda. **Objectif POC** — Démontrer la **faisabilité technique** d’un système RAG répondant aux exigences suivantes :

* Données **≤ 1 an** et **ville sélectionnée** (Paris dans ce POC) ;
* Réponses **factuelles** et **anachronisme‑safe** (basées uniquement sur le contexte) ;
* Pipeline **reproductible** (scripts, snapshot, index) ;
* **Tests** et **évaluations** quantitatives (retrieval et génération) ;
* Préparation à une **démo live** et **scalabilité** future.

**Ce qui est livré** : code versionné (prétraitement, index, RAG), tests unitaires, scripts d’évaluation, datasets d’évaluation (règles et QA annotées), snapshot Parquet gelé, README d’exécution.

# 3. Données & prétraitement

### 3.1 Source

* Dataset **OpenAgenda** via Opendatasoft : evenements-publics-openagenda.
* Accès API paginé, filtrage par **ville** (ex. refine.location\_city=Paris).

### 3.2 Nettoyage et normalisation

Implémenté dans src/data/preprocess\_openagenda.py :

* **Schéma stable** par normalisation des champs : uid, title, city, postal\_code, venue, website/permalink, tags, start\_utc, end\_utc, text.
* **Dates UTC** (timezone‑aware) et **filtre** : start\_utc ≥ now−365j.
* **Nettoyage HTML** via BeautifulSoup (texte brut), tags → chaîne unifiée.
* **Déduplication** sur uid.
* **Tests unitaires** (voir §7) : valident fenêtre ≤ 1 an + filtre ville + nettoyage & schéma.

### 3.3 Snapshot gelé

* Script scripts/make\_snapshot.py → data/snapshots/events\_eval.parquet.
* Intérêt : **reproductibilité** des évaluations et capacité à **geler l’état** des données.

# 4. Architecture & pipeline

### 4.1 Vue d’ensemble

OpenAgenda API → Prétraitement (≤1 an, ville, HTML strip) → Snapshot (Parquet)

→ Chunking → Embeddings (mistral-embed) → FAISS index (local)

Question utilisateur → Retriever (FAISS, MMR, k) → Contexte → Mistral Chat → Réponse

Éval retrieval : qa\_rules.csv + Snapshot → Recall@k

Éval génération : qa\_annotated.csv + Index → Exact/Contains (+ snapping) → gen\_eval\_report.csv

.env → MISTRAL\_API\_KEY

### 4.2 Indexation & recherche

Implémentée par src/index/build\_faiss.py :

* **Documents** enrichis avec des tokens normalisés (année/mois/début/fin, code postal) pour aider la **désambiguïsation temporelle** et **géographique**.
* **Chunking** : RecursiveCharacterTextSplitter(chunk\_size=800, chunk\_overlap=120).
* **Embeddings** : MistralAIEmbeddings(model="mistral-embed").
* **Vector store** : FAISS.from\_documents(...) → data/index/faiss/.

### 4.3 RAG runtime (chat CLI)

src/rag/rag\_pipeline.py :

* Chargement FAISS et retriever (k configurable, défaut = 12).
* **Prompt système** strict : *utiliser uniquement le CONTEXTE ; sinon « I don't know. »*.
* **Fallback de modèle** Mistral : medium → large → small (résilience capacité/ratelimit).
* Réponse dans la **langue de la question** (FR/EN).

# 5. Évaluation

### 5.1 Retrieval — Recall@k

* **Règles** : data/eval/qa\_rules.csv (20 lignes). Colonnes : city, date\_from, date\_to, tags\_any (OR regex, accent‑insensible) et, optionnellement, postal\_code.
* **Script** : scripts/eval\_retrieval.py ; construit un **gold set d’UID** par filtrage du **snapshot** (ville/dates/tags/ZIP), puis interroge FAISS avec la **question** naturelle FR.
* **Mesure** : **hit** si retrieved\_uids ∩ gold\_uids ≠ ∅.
* **Résultat** : **Recall@10 = 1.000**, **Recall@20 = 1.000** (20/20).

### 5.2 Génération — Exact / Contains

* **Jeu annoté** : data/eval/qa\_annotated.csv (15 questions **FR**). Réponses gold **unitaires** (multi‑gold possible via A||B).
* **Script** : scripts/eval\_generation.py (robuste et scalable) :
  + **Retrieval** : **MMR** (diversification), k=30, fetch\_k=60, lambda=0.5.
  + **Instruction** : *répondre par une* ***seule*** *courte* ***expression*** *:* ***titre****,* ***lieu*** *ou* ***code postal****.*
  + **Snapping** : post‑traitement mappant la réponse LLM vers la **valeur canonique** (titre/lieu/ZIP) la plus proche **présente dans le contexte** (fuzzy matching).
  + **Normalisation** : accents/emoji/ponctuation (Unidecode), **postal** : comparaison 5 chiffres.
  + **Fallback modèles** : medium → large → small + **retries**.
* **Résultat** : **ExactMatch = 15/15**, **ContainsGold = 15/15**. Fichier de sortie : data/eval/gen\_eval\_report.csv.

# 6. Résultats & démonstration

* Le chatbot répond de façon **factuelle** en citant titres, lieux, dates ou liens **uniquement si présents** dans le contexte.
* Questions d’exemple (démo CLI) :
  1. *« Des idées pour enfants dans le 75001 en septembre 2025 ? »*
  2. *« Un événement sur le suffrage des femmes à Paris en septembre 2024 ? »*
  3. *« Quel est le code postal de “Spectacles contés : la Cour des Contes …” ? »*
* Comportement attendu : réponse **brève** strictement depuis le contexte ; **« I don't know. »** si hors fenêtre (≤ 1 an) ou hors périmètre (ville).

# 7. Tests unitaires

tests/test\_vector\_time\_city.py couvre :

* **Fenêtre** ≤ 1 an et **filtre ville** (mock API) ;
* **Schéma & nettoyage** (colonnes/HTML/tags en string) ;
* **FAISS round‑trip** (save → load → retrieve) avec **fake embeddings** (offline), garantissant le fonctionnement de base sans dépendances externes.

# 8. Limites & risques

* **Fenêtre temporelle bornée** (≤ 1 an) : questions hors période → *« I don't know. »*
* **Qualité des données** (titres décorés, champs manquants) : atténuée par normalisation et snapping.
* **Quotas/latence LLM** : atténués par **fallback** et prompts **sobres**.
* **Index local** (FAISS) : très rapide mais **non distribué** ; à surveiller si volumétrie multi‑villes/long terme.

# 9. Recommandations (passage à l’échelle)

* **Observabilité** : journalisation des requêtes, **Recall@k**, latence, taux « I don't know. », dérive des données.
* **Indexation** : tester HNSW/IVF, ou service vectoriel managé (scaling, sauvegarde, monitoring).
* **Reranking** : cross‑encoder léger pour reclasser le top‑k si besoin de précision top‑3.
* **Anti‑hallucination** : réponses **sourcées** (liens), score de confiance, garde‑fous sur dates/lieux.
* **Automatisation** : ETL incrémental et **tests continus** (qualité données et perfs retrieval/génération).

# 10. Reproductibilité & commandes clés

**Artifacts** : snapshot Parquet gelé, index FAISS rebuildable, jeux/rapports d’évaluation. **Secrets** : .env.example (jamais de clé en clair).

Extraits (voir README pour la liste complète) :

*# Snapshot*

python scripts/make\_snapshot.py --city Paris --max-records 4000 --lookback-days 365 --out data/snapshots/events\_eval.parquet

*# Index*

python -m src.index.build\_faiss --city Paris --max-records 9000 --lookback-days 365 \

--chunk-size 800 --chunk-overlap 120 --index-out data/index/faiss

*# Éval retrieval*

python scripts/eval\_retrieval.py --qa data/eval/qa\_rules.csv \

--snapshot data/snapshots/events\_eval.parquet --index data/index/faiss --klist 10,20

*# Éval génération*

python scripts/eval\_generation.py --qa data/eval/qa\_annotated.csv \

--index data/index/faiss --k 30 --out data/eval/gen\_eval\_report.csv

*# Chatbot*

python -m src.rag.rag\_pipeline --index data/index/faiss --k 12 --chat-size medium

# 11. Annexes

### 11.1 Arborescence projet (résumé)

project\_root/

├── .env.example

├── requirements.txt

├── README.md

├── data/

│ ├── eval/ (qa\_rules.csv, qa\_annotated.csv, gen\_eval\_report.csv)

│ ├── index/faiss/ (index.faiss, index.pkl)

│ └── snapshots/events\_eval.parquet

├── scripts/ (make\_snapshot.py, eval\_retrieval.py, eval\_generation.py)

├── src/

│ ├── data/preprocess\_openagenda.py

│ ├── index/build\_faiss.py

│ └── rag/rag\_pipeline.py

└── tests/test\_vector\_time\_city.py

### 11.2 Paramètres principaux

* **Ville** : Paris (configurable)
* **Fenêtre** : 365 jours
* **Chunking** : 800 / 120
* **Retriever (éval génération)** : MMR, k=30, fetch\_k=60, lambda=0.5
* **Retriever (chat)** : k=12
* **LLM** : Mistral (fallback medium → large → small)
* **Embeddings** : mistral-embed (cohérent build/load)