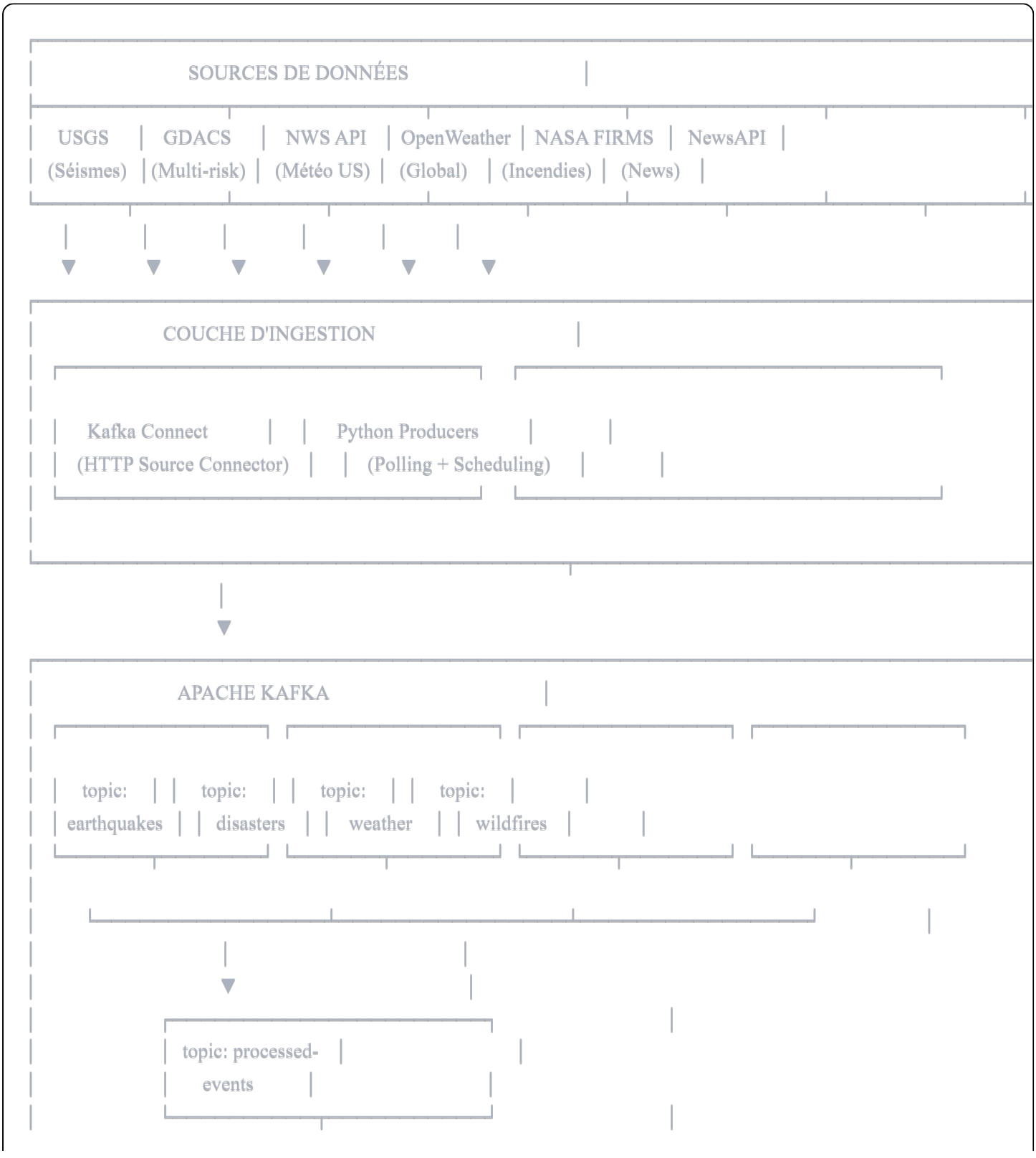
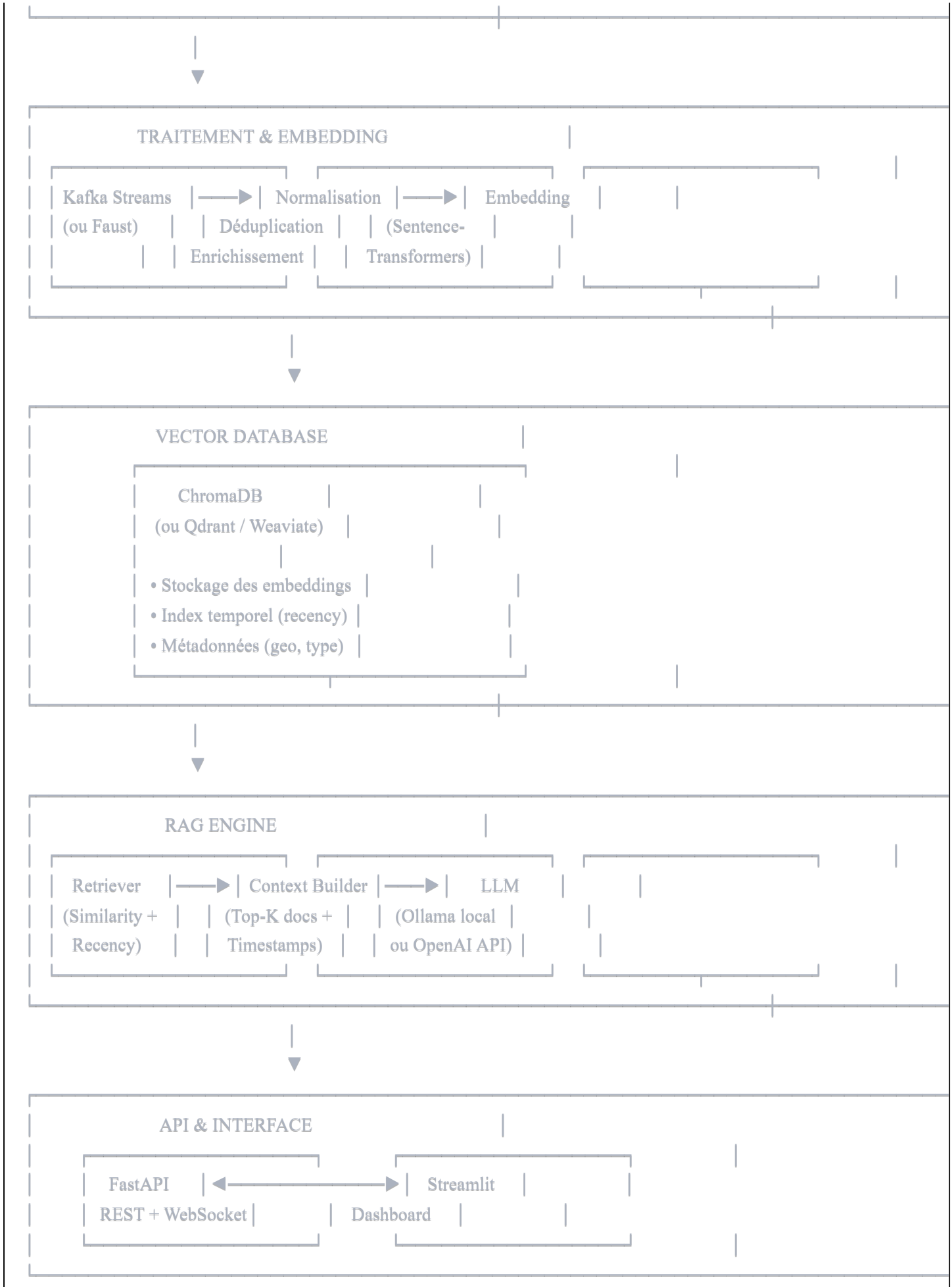
Architecture Technique - Real-time RAG avec Apache Kafka

🎯 Vue d'Ensemble

Ce document décrit l'architecture complète du système RAG (Retrieval Augmented Generation) en temps réel pour la surveillance des catastrophes mondiales.

📐 Architecture Globale




TRAITEMENT & EMBEDDING
Kafka Streams
(ou Faust)
Normalisation
Déduplication
Enrichissement
Embedding
(Sentence-
Transformers)
VECTOR DATABASE
ChromaDB
(ou Qdrant / Weaviate)
• Stockage des embeddings
• Index temporel (recency)
• Métadonnées (geo, type)
RAG ENGINE
Retriever
(Similarity +
Recency)
Context Builder
(Top-K docs +
Timestamps)
LLM
(Ollama local
ou OpenAI API)
API & INTERFACE
FastAPI
REST + WebSocket
Streamlit
Dashboard

🧩 Composants Détaillés

1. Sources de Données (Data Sources)

Source	Endpoint	Fréquence Polling	Format
USGS	earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_hour.geojson	1 min	GeoJSON
GDACS	gdacs.org/xml/rss.xml	5 min	RSS/XML
NWS	api.weather.gov/alerts/active	2 min	GeoJSON
OpenWeatherMap	api.openweathermap.org/data/3.0/onecall	10 min	JSON
NASA FIRMS	firms.modaps.eosdis.nasa.gov/api/...	15 min	CSV/JSON
NewsAPI	newsapi.org/v2/everything	5 min	JSON

2. Apache Kafka

Topics Structure

```
disaster-rag-cluster/
├── raw-earthquakes        # Données brutes USGS
├── raw-disasters          # Données brutes GDACS
├── raw-weather-alerts     # Alertes NWS + OpenWeather
├── raw-wildfires          # Données NASA FIRMS
├── raw-news               # Articles NewsAPI
├── processed-events       # Événements normalisés
├── embedded-events        # Événements avec embeddings
└── dlq-events             # Dead Letter Queue (erreurs)
```

Configuration Kafka

yaml

```yaml
# docker-compose.yml (extrait)
kafka:
  num.partitions: 3
  default.replication.factor: 1
  retention.ms: 604800000  # 7 jours
  retention.bytes: 1073741824  # 1 GB
```

3. Schéma de Données Unifié

json


```
{
 "event_id": "usgs_nc75101291",
 "event_type": "earthquake",
 "source": "USGS",
 "timestamp": "2024-12-11T14:32:00Z",
 "ingested_at": "2024-12-11T14:32:05Z",
 "location": {
  "latitude": 36.5123,
  "longitude": -121.8765,
  "place": "10km SW of Hollister, CA",
  "country": "US"
 },
 "severity": {
  "level": "moderate",
  "value": 4.2,
  "unit": "magnitude"
 },
 "content": {
  "title": "M 4.2 - 10km SW of Hollister, CA",
  "description": "Earthquake of magnitude 4.2 detected...",
  "raw_text": "..."
 },
 "metadata": {
  "alert_level": "green",
  "tsunami_warning": false,
  "felt_reports": 156,
  "url": "https://earthquake.usgs.gov/..."
 }
}
```


4. Vector Database (ChromaDB)

Pourquoi ChromaDB ?

- ✅ Open-source et gratuit
- ✅ Simple à installer (pip install)
- ✅ Supporte les métadonnées pour filtrage
- ✅ Persistance locale
- ✅ Idéal pour projets académiques

Collection Schema

```python
collection = chroma_client.create_collection(
    name="disaster_events",
    metadata={"hnsw:space": "cosine"}
)

# Structure d'un document
{
    "id": "usgs_nc75101291",
    "embedding": [0.123, -0.456, ...],  # 384 dimensions
    "document": "M 4.2 earthquake detected 10km SW of Hollister...",
    "metadata": {
        "event_type": "earthquake",
        "source": "USGS",
        "timestamp": 1702305120,  # Unix timestamp
        "latitude": 36.5123,
        "longitude": -121.8765,
        "severity_level": "moderate",
        "severity_value": 4.2
    }
}
```

5. Embedding Model

Modèle	Dimensions	Taille	Performance
all-MiniLM-L6-v2 (recommandé)	384	80 MB	Rapide, bon équilibre
all-mpnet-base-v2	768	420 MB	Plus précis, plus lent
paraphrase-multilingual-MiniLM-L12-v2	384	470 MB	Multilingue

6. LLM Options

Option	Coût	Latence	Recommandation
Ollama (Llama 3.2)	Gratuit	~2-5s	✅ Projet académique
Ollama (Mistral 7B)	Gratuit	~2-3s	✅ Bon compromis
OpenAI GPT-4o-mini	~$0.15/1M tokens	~1s	Si budget disponible
Groq (Llama 3)	Gratuit (limite)	~0.5s	Alternative cloud

🔄 Flux de Données

```
1. INGESTION (toutes les X minutes)
   API Source → Python Producer → Kafka Topic (raw-*)

2. PROCESSING (stream continu)
   Kafka Topic (raw-*) → Kafka Streams → Normalisation → Topic (processed-events)

3. EMBEDDING (stream continu)
   Topic (processed-events) → Embedding Model → ChromaDB

4. QUERY (à la demande)
   User Query → Embed Query → ChromaDB Search → Top-K Results → LLM → Response
```

🛠️ Stack Technologique

Infrastructure

Composant	Technologie	Version
Message Broker	Apache Kafka	3.6+
Stream Processing	Kafka Streams / Faust	-
Container	Docker + Docker Compose	24+

Backend

Composant	Technologie	Version
Langage	Python	3.10+
API Framework	FastAPI	0.104+
Task Scheduler	APScheduler	3.10+
Kafka Client	kafka-python / confluent-kafka	-

AI/ML

Composant	Technologie	Version
Embeddings	sentence-transformers	2.2+
Vector DB	ChromaDB	0.4+
LLM Runtime	Ollama	0.1+
RAG Framework	LangChain (optionnel)	0.1+

Frontend

Composant	Technologie	Version
Dashboard	Streamlit	1.28+
Visualisation	Plotly / Folium	-

📁 Structure du Projet

```
disaster-rag-system/
│
├── docker-compose.yml          # Infrastructure (Kafka, Zookeeper, etc.)
├── .env                        # Variables d'environnement (API keys)
├── requirements.txt            # Dépendances Python
│
├── src/
│   ├── __init__.py
│   │
│   ├── ingestion/              # Producteurs Kafka
│   │   ├── __init__.py
│   │   ├── base_producer.py    # Classe abstraite
│   │   ├── usgs_producer.py    # Séismes
│   │   ├── gdacs_producer.py   # Multi-risques
│   │   ├── nws_producer.py     # Météo US
│   │   ├── owm_producer.py     # OpenWeatherMap
│   │   ├── firms_producer.py   # Incendies
│   │   └── news_producer.py    # Actualités
│   │
│   ├── processing/             # Traitement des streams
```

```
│   │   ├── __init__.py
│   │   ├── normalizer.py       # Normalisation du schéma
│   │   ├── deduplicator.py     # Déduplication
│   │   ├── enricher.py         # Enrichissement géo
│   │   └── stream_processor.py # Kafka Streams / Faust
│   │
│   ├── embedding/              # Pipeline d'embedding
│   │   ├── __init__.py
│   │   ├── embedder.py         # Sentence Transformers
│   │   ├── chunker.py          # Text chunking
│   │   └── vector_store.py     # ChromaDB client
│   │
│   ├── rag/                    # Moteur RAG
│   │   ├── __init__.py
│   │   ├── retriever.py        # Recherche vectorielle
│   │   ├── reranker.py         # Re-ranking par recency
│   │   ├── context_builder.py  # Construction du contexte
│   │   └── generator.py        # LLM generation
│   │
│   ├── api/                    # API REST
│   │   ├── __init__.py
│   │   ├── main.py             # FastAPI app
│   │   ├── routes/
│   │   │   ├── query.py        # Endpoint RAG
│   │   │   ├── events.py       # Liste des événements
│   │   │   └── health.py       # Health checks
│   │   └── websocket.py        # Real-time updates
│   │
│   └── utils/                  # Utilitaires
│       ├── __init__.py
│       ├── config.py           # Configuration
│       ├── logger.py           # Logging
│       └── schemas.py          # Pydantic models
│
├── dashboard/                  # Interface Streamlit
│   ├── app.py
│   ├── pages/
│   │   ├── 1_📊_Overview.py
│   │   ├── 2_🗺️_Map.py
│   │   └── 3_💬_Chat.py
│   └── components/
│
├── tests/                      # Tests
│   ├── test_ingestion.py
│   ├── test_processing.py
│   └── test_rag.py
│
```

```
├── scripts/                # Scripts utilitaires
│   ├── setup_kafka.sh
│   ├── start_producers.py
│   └── init_vectordb.py
│
└── docs/                   # Documentation
    ├── architecture.md
    └── api.md
```

🚀 Démarrage Rapide

1. Prérequis

```bash
# Docker et Docker Compose
docker --version
docker-compose --version

# Python 3.10+
python --version

# Ollama (pour LLM local)
ollama --version
```

2. Installation

```bash
```

```
# Cloner le projet
git clone https://github.com/your-repo/disaster-rag-system.git
cd disaster-rag-system

# Créer l'environnement virtuel
python -m venv venv
source venv/bin/activate  # Linux/Mac
# ou: venv\Scripts\activate  # Windows

# Installer les dépendances
pip install -r requirements.txt

# Copier et configurer les variables d'environnement
cp .env.example .env
# Éditer .env avec vos clés API
```

3. Démarrer l'infrastructure

bash

```
# Lancer Kafka + Zookeeper + ChromaDB
docker-compose up -d

# Vérifier que tout fonctionne
docker-compose ps

# Télécharger le modèle LLM
ollama pull llama3.2
```

4. Lancer le système

bash

```
# Terminal 1: Producteurs (ingestion)
python scripts/start_producers.py

# Terminal 2: Stream processor
python -m src.processing.stream_processor

# Terminal 3: API
uvicorn src.api.main:app --reload

# Terminal 4: Dashboard
streamlit run dashboard/app.py
```

📊 Métriques & Monitoring

KPIs à Suivre

- **Ingestion Rate**: Événements/minute par source
- **Processing Latency**: Temps entre ingestion et embedding
- **Query Latency**: Temps de réponse RAG (P50, P95, P99)
- **Vector DB Size**: Nombre de documents indexés
- **Retrieval Quality**: Pertinence des résultats (à évaluer manuellement)

Outils Recommandés

- **Kafka UI**: Interface pour monitorer Kafka (Kafdrop, Kafka-UI)
- **Prometheus + Grafana**: Métriques système
- **Logging**: Python logging → fichiers JSON

🔐 Considérations de Sécurité

1. **API Keys**: Stockées dans `.env`, jamais dans le code
2. **Rate Limiting**: Respecter les limites des APIs sources
3. **Data Retention**: Politique de suppression après 7 jours
4. **Accès**: Authentification sur l'API (JWT optionnel pour le projet)

📈 Évolutions Possibles

1. **Multi-langue**: Ajouter des sources en français, espagnol
2. **Alertes Push**: Notifications WebSocket/Email pour événements critiques
3. **Fine-tuning**: Adapter le modèle d'embedding aux données disaster
4. **Kubernetes**: Déploiement scalable en production
5. **Evaluation**: Benchmark RAGAS pour mesurer la qualité RAG