



# Albert Query - Agentic Movie Intelligence System

An intelligent conversational agent for querying movie and TV series data using Retrieval Augmented Generation (RAG), multi-tool orchestration, and semantic search.

Python 3.8+

Streamlit 1.51.0

LangChain 1.0.3

LangGraph 1.0.2

ChromaDB 1.3.4



## Table of Contents

- [Overview](#)
- [Features](#)
- [Architecture](#)
- [Installation](#)
- [Usage](#)
- [Project Structure](#)
- [Components](#)
- [Workflow](#)
- [Technologies](#)
- [Future Improvements](#)
- [Contributors](#)
- [License](#)



## Overview

**Albert Query** is an agentic AI system we developed as part of our M1 project at **Albert School** in collaboration with **Mines Paris - PSL**. The system intelligently answers questions about movies and TV series by orchestrating multiple data sources and tools through a LangGraph-based workflow.





### What Makes It Special?

Unlike traditional chatbots, Albert Query:

- **Plans before acting** - Analyzes each question to determine which tools are needed
- **Multi-source intelligence** - Combines SQL databases, vector search, external APIs, and web search
- **Semantic understanding** - Uses OpenAI embeddings to find movies by plot similarity
- **Source attribution** - Always shows where information comes from
- **Context-aware** - Maintains conversation history for follow-up questions

### Use Cases

- 🔍 **Semantic Search**: "Find me movies about space exploration with AI themes"

-  **Data Analysis:** "How many comedies were released on Netflix after 2020?"
  -  **Movie Discovery:** "Show me films similar to Inception"
  -  **Trend Analysis:** "What are the top-rated action movies from the 2010s?"
  -  **Latest Info:** "What's trending in movies this week?"
- 

## ✦ Features

### Core Capabilities

#### **Intelligent Query Planning**

- LLM-based planner analyzes questions and conversation history
- Automatically selects optimal tools (SQL, Semantic Search, OMDb, Web)
- Avoids unnecessary API calls for efficiency

#### **Multi-Database SQL Queries**

- Comprehensive catalog of 8,000+ movies/shows from Netflix, Disney+, Amazon Prime
- Structured queries with filters (year, genre, rating, type)
- Automatic database schema understanding

#### **Semantic Vector Search**

- 114MB of OpenAI embeddings (text-embedding-3-small)
- Find movies by plot descriptions, themes, or similarity
- Natural language queries in English or French

#### **OMDb API Integration**

- Enriched movie metadata (actors, awards, ratings, posters)
- IMDb links for detailed information
- Full plot summaries

#### **Web Search**

- DuckDuckGo integration for trending topics
- Latest movie news and releases
- Current events in cinema

#### **Source Attribution**

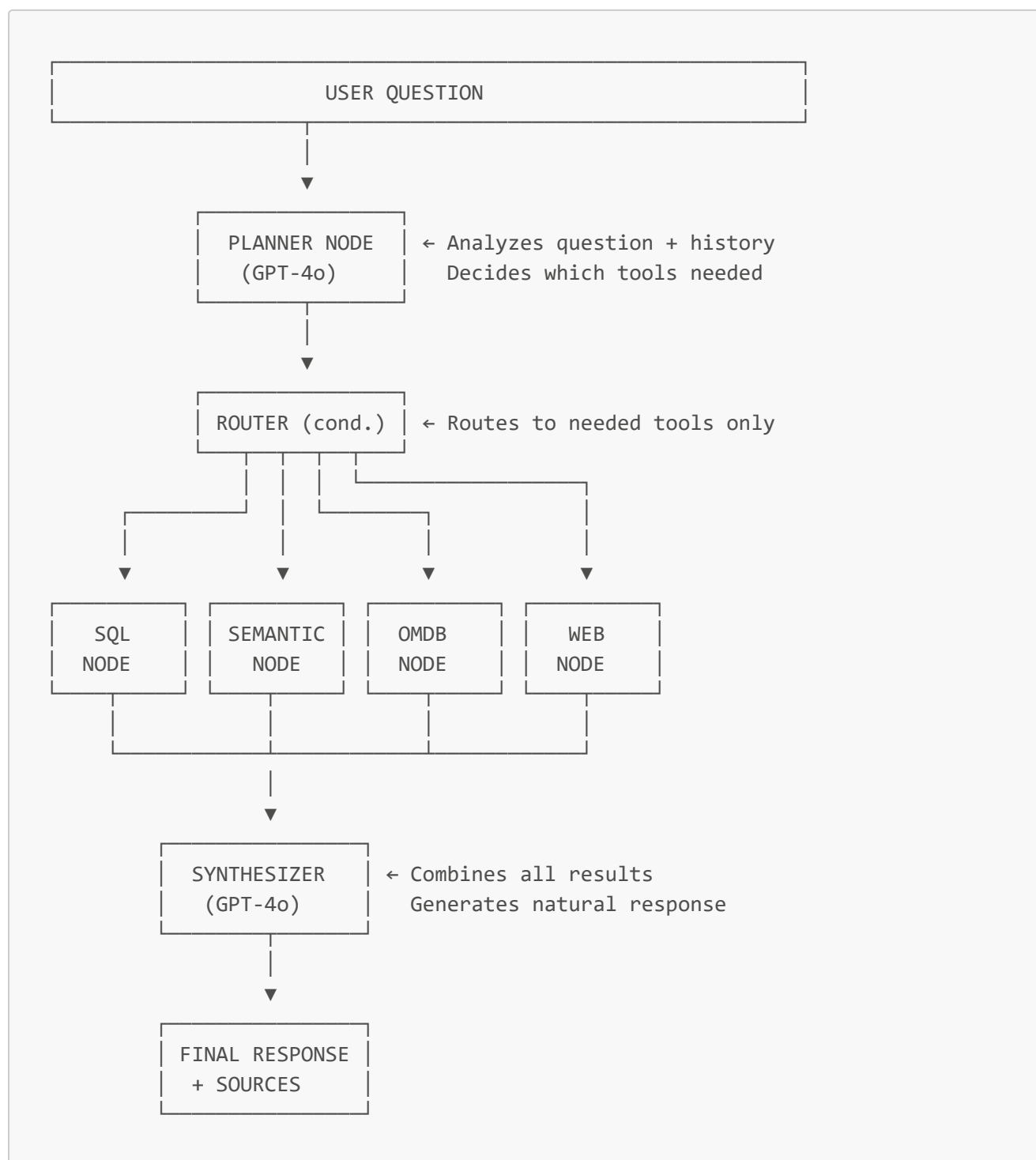
- Tracks all information sources
- Visual display of databases, APIs, and search results used
- Clickable links to IMDb and search results

#### **Conversational Memory**

- LangGraph checkpointing with MemorySaver
- Full conversation history maintained
- Context resolution ("that movie" → actual movie name)

## 📁 Architecture

Our system follows an **agentic architecture** using LangGraph to create a stateful, multi-tool workflow:



## 📁 Project Structure

```
Agentic_Systems_with_RAG_Lamy-Waerniers/
├── code/                                # Main source code
│   ├── albert_query_app.py             # Main Streamlit application (1009
lines)
```

```

|   |   | embedding_manager.py           # Vector embedding management (418
lines)
|   |   | architecture.md               # System architecture
documentation
|   |   | embedding.ipynb                # Embedding pipeline notebook
|   |   | SQLdb_creator.ipynb           # Database creation from CSVs
|   |   | testing.ipynb                 # API and integration tests
|   |   | test_semantic_search.ipynb    # Semantic search validation
|
|   | data/                             # Data storage
|   |   | csv_db/                       # Source CSV files
|   |   |   | amazon_prime_titles.csv   # Amazon Prime catalog (3.9MB)
|   |   |   | netflix_titles.csv        # Netflix catalog (3.4MB)
|   |   |   | disney_plus_titles.csv    # Disney+ catalog (385KB)
|   |   | databases/
|   |   |   | movie.db                  # Consolidated SQLite DB (32.9MB)
|   |   |   | vector_database/          # ChromaDB persistent storage
|   |   |   |   | chroma.sqlite3        # Vector DB metadata (42.7MB)
|   |   |   |   | 19c0759d-.../        # Embedding data (114MB)
|   |   |   | memory/                  # Conversation storage
|   |   |   |   | conversations/
|   |   |   |   | user_profiles/
|   |
|   | doc/                              # Documentation
|   |   | graph_schema.png              # LangGraph workflow diagram
|   |   | omdb_api_doc.json             # OMDB API reference
|   |   | OMDB_API_doc.txt
|   |
|   | .env                             # Environment configuration (git-
ignored)
|   | .gitignore                       # Git ignore rules
|   | requirements.txt                 # Python dependencies (223
packages)
|   | README.md                        # This file

```

## 🔗 Components

### 1. Main Application (`albert_query_app.py`)

The core orchestrator built with **Streamlit** and **LangGraph**.

#### Key Components:

##### Planner Node

```

def planner_node(state: AgentState) -> dict:
    """Analyzes question and plans tool usage"""
    # Uses GPT-4o-mini to decide which tools are needed

```

```
# Returns: needs_sql, needs_semantic, needs_omdb, needs_web flags
# Also prepares specific queries for each tool
```

### Decision Logic:

- SQL: For structured data (titles, years, ratings, genres)
- Semantic: For plot-based searches and "movies like X"
- OMDB: For detailed metadata (actors, awards, posters)
- Web: For trending topics and recent news

### SQL Node

```
def sql_node(state: AgentState) -> dict:
    """Generates and executes SQL queries"""
    # Analyzes database catalog
    # Generates SQL with proper table names, filters
    # Executes query and returns results
```

### Features:

- Auto-detects correct table names
- Handles genre filtering (comma-separated values)
- Year ranges, type filtering (Movie vs TV Show)
- Result limiting and ordering

### Semantic Search Node

```
def semantic_search_node(state: AgentState) -> dict:
    """Performs vector similarity search"""
    # Translates query to English if needed
    # Optimizes query for embeddings
    # Searches ChromaDB with OpenAI embeddings
    # Returns top-k similar movies
```

### Important Notes:

- Movie descriptions are in **English** - queries are auto-translated
- Similarity scores may appear low (40-60%) but rankings are accurate
- Uses cosine similarity on text-embedding-3-small vectors

### OMDB Node

```
def omdb_node(state: AgentState) -> dict:
    """Fetches enriched movie data from OMDB API"""
```

```
# Extracts movie title from context
# Queries OMDB API
# Returns: plot, actors, awards, IMDb rating, poster URL
```

## Web Search Node

```
def web_node(state: AgentState) -> dict:
    """Performs web search via DuckDuckGo"""
    # Executes search query
    # Returns recent results with URLs
```

## Synthesizer Node

```
def synthesizer_node(state: AgentState) -> dict:
    """Combines all results into natural response"""
    # Takes SQL, semantic, OMDB, web results
    # Generates coherent natural language answer
    # Includes source attribution
```

## 2. Embedding Manager ([embedding\\_manager.py](#))

Handles creation and management of vector embeddings for semantic search.

### Key Functions:

```
def get_or_create_collection(chroma_path,
                             collection_name="movie_descriptions"):
    """Gets or creates ChromaDB collection with OpenAI embeddings"""

def embed_movies_if_not_exists(collection, movies, batch_size=100):
    """Embeds movies only if not already in collection"""
    # Checks existing IDs to avoid duplicates
    # Batches for efficiency
    # Tracks progress with tqdm

def query_movies(collection, query_text, n_results=5, where_filter=None):
    """Queries movies by semantic similarity"""
    # Embeds query with same model
    # Searches vector DB
    # Returns ranked results with similarity scores
```

### Embedding Structure:

```
{
  "id": "net0042",           # Unique ID (prefix + index)
  "document": "A sci-fi thriller...", # Movie description (embedded)
  "metadata": {
    "title": "The Matrix",
    "database": "movie.db",
    "table": "netflix_titles"
  }
}
```

### 3. Database Schema

#### SQL Tables:

Each table (netflix\_titles, amazon\_prime\_titles, disney\_plus\_titles) contains:

Column	Type	Description
show_id	TEXT	Unique identifier (e.g., "s1")
type	TEXT	"Movie" or "TV Show"
title	TEXT	Movie/show title
director	TEXT	Director name(s)
cast	TEXT	Comma-separated actors
country	TEXT	Production country
date_added	TEXT	Date added to platform
release_year	INTEGER	Year of release
rating	TEXT	Age rating (PG, R, etc.)
duration	TEXT	Runtime or seasons
listed_in	TEXT	Comma-separated genres
description	TEXT	Plot summary

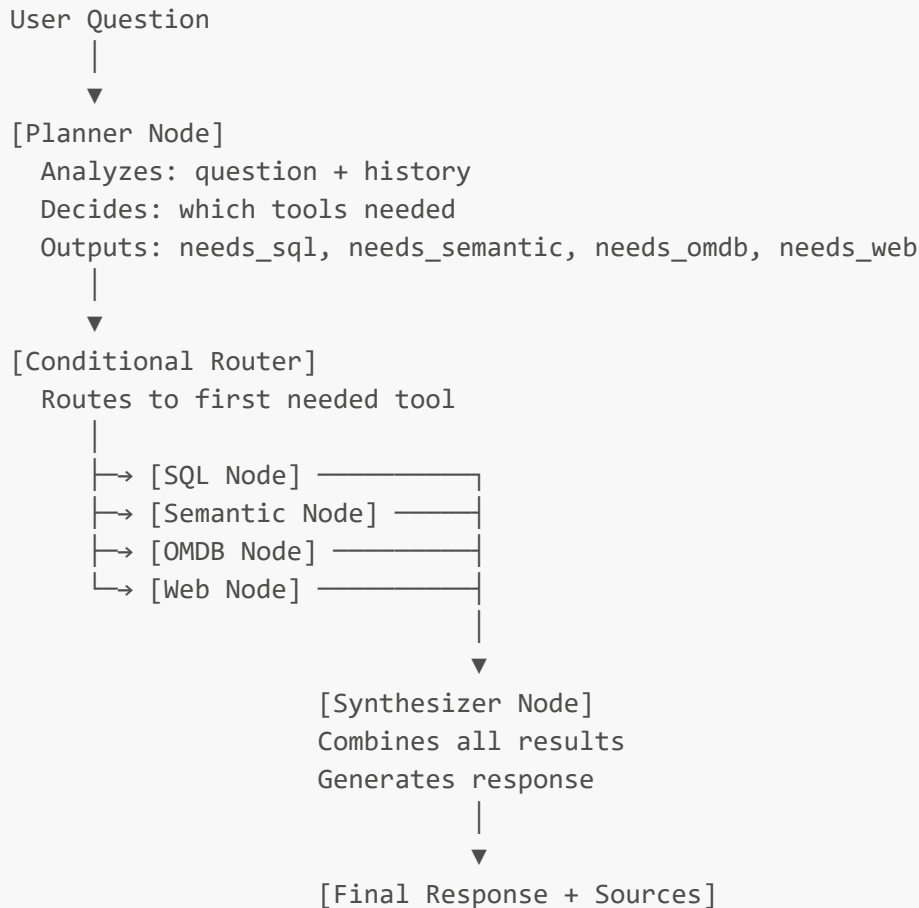
**Total Records:** ~8,800 movies/shows

### 4. Vector Database

#### ChromaDB Collection:

- Name: `movie_descriptions`
- Embedding Model: OpenAI text-embedding-3-small (1536 dimensions)
- Total Vectors: ~8,800 movie descriptions
- Storage: Persistent on disk (~156MB total)
- Metadata: title, database, table for filtering

## Workflow



## Future Improvements

We've identified several areas for optimization and enhancement. Here's our roadmap:

### 1. Catalog Caching System

#### Problem:

- Database catalog is rebuilt on every app startup
- Slow initialization (~2-5 seconds)
- Redundant SQL queries for schema introspection

#### Solution:

```
# Implement caching with invalidation detection
def get_or_build_catalog(db_path: str, cache_path: str) -> dict:
    """
    Cache database catalog as JSON
    - Compare file modification times to detect changes
    - Load from cache if DB unchanged
    - Rebuild only when necessary
    """
```



### Expected Impact:

- ⚡ 10-50x faster startup time
- 📁 Reduced SQL queries
- 🔄 Auto-invalidation on schema changes

### Implementation:

- Save catalog to `data/databases/catalog_cache.json`
- Include DB file mtime and size for change detection
- Add force-rebuild option for manual invalidation

## 2. 🧠 Persistent Long-Term Memory

### Current State:

- Memory stored in LangGraph's MemorySaver (in-memory only)
- Lost on application restart
- No cross-session learning

### Proposed Architecture:

```
# SQLite-based conversation storage
conversations/
├── user_123/
│   ├── session_20250116_001.json    # Conversation history
│   ├── session_20250116_002.json
│   └── preferences.json             # Learned preferences
├── user_456/
└── ...

# Conversation schema
{
  "session_id": "20250116_001",
  "user_id": "user_123",
  "timestamp": "2025-01-16T10:30:00Z",
  "messages": [...],
  "topics": ["action movies", "2020s cinema"],
  "preferences_learned": {
    "favorite_genres": ["action", "sci-fi"],
    "preferred_platforms": ["netflix"]
  }
}
```

### Features to Add:

- 📁 Persist conversations to disk (JSON or SQLite)
- 👤 User-specific history and preferences

- 🔍 Semantic search over past conversations
- 📊 Analytics on user interests
- 🎯 Personalized recommendations based on history

### Technical Implementation:


- Replace MemorySaver with custom SQLiteCheckpoint
- Add user authentication (see #3)
- Implement conversation summarization for long histories
- Privacy controls (GDPR compliance)

## 3. 🗑️ User Management & API Key Interface

### Current Limitation:


- Single shared API keys in `.env`
- No multi-user support
- API costs not attributable to users


### Proposed UI:



 Settings

---

User ID: [user\_123                      ]

 OpenAI API Key:  
 [sk-proj-\*\*\*\*\*] [Save]

 OMDb API Key:  
 [8871ab87                      ] [Save]

 Token Usage This Session: 1,234  
 Estimated Cost: \$0.05

[Clear History] [Export Data]

### Implementation:

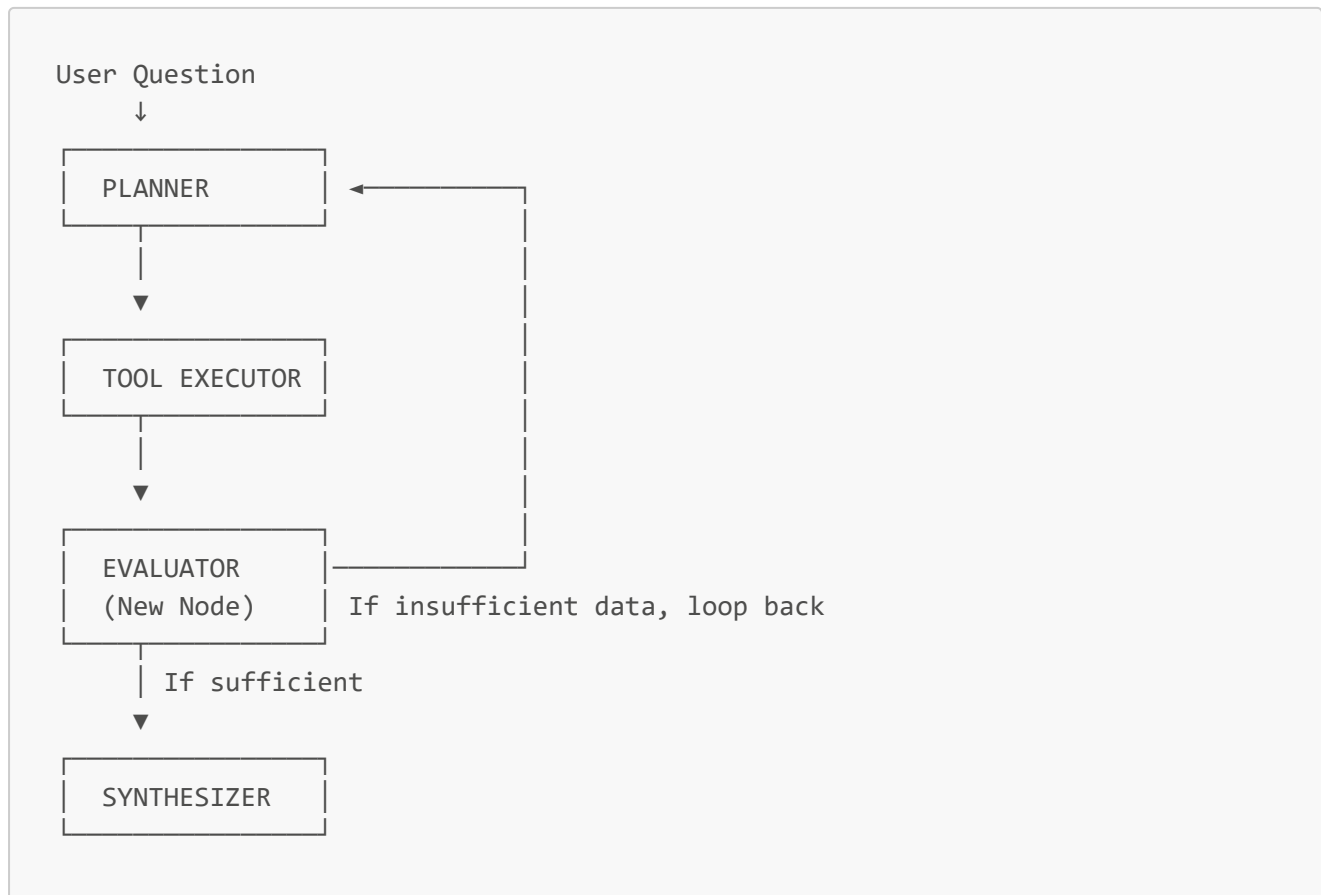
- Streamlit sidebar with settings panel
- Encrypted key storage per user (keyring library)
- Session-based authentication
- Token tracking and cost estimation
- Rate limiting per user

## 4. 🔄 Workflow Enhancement: Planner Loop

### Current Issue:

- Linear workflow: Planner → Tools → Synthesizer → End
- No feedback loop if initial plan was insufficient
- Cannot self-correct or ask for more tools

### Proposed Architecture:







### Evaluator Node Logic:

```

def evaluator_node(state: AgentState) -> dict:
    """
    Checks if gathered data is sufficient to answer question
    Returns: continue (to synthesizer) or replan (back to planner)
    """
    results_quality = assess_data_completeness(state)

    if results_quality["sufficient"]:
        return {"next": "synthesize"}
    else:
        # Request additional tools
        return {
            "next": "planner",
            "feedback": "Need more data: " + results_quality["missing"]
        }
  
```

### Benefits:

-  Self-correcting behavior
-  More complete answers
-  Adaptive tool selection
-  Avoids premature responses

## 5. Embedding Quality Improvements

### Current Limitations:

- Similarity scores often low (<50%)
- Movie descriptions are single sentences only
- No chunking strategy
- Basic embedding model (text-embedding-3-small)

### Improvement Strategies:




#### A. Enhance Description Quality (Priority #1)

Currently, we only embed the plot description field from databases (single sentence).

#### Solution:

Use APIs to enrich our database with way longer movie descriptions.

Expected Impact:

-  More contextual embeddings
-  Better similarity scores (10-20% improvement)
-  Genre/cast matching in semantic search

## 6. Structured Output Enforcement

### Problem:

- JSON parsing errors possible in synthesizer
- Inconsistent response formats
- Difficult to extract structured data

### Solution - Pydantic Models:

```
from pydantic import BaseModel, Field
from typing import List, Optional

class MovieResult(BaseModel):
    title: str
    year: int
    rating: Optional[str]
    genres: List[str]
    description: str
    source: str # "sql", "semantic", "omdb"

class QueryResponse(BaseModel):
```

```

    answer: str = Field(..., description="Natural language answer")
    movies: List[MovieResult] = Field(default=[], description="Structured movie
data")
    sources: List[str] = Field(..., description="Sources used")
    confidence: float = Field(..., ge=0, le=1, description="Confidence score")
    follow_up_suggestions: List[str] = Field(default=[], description="Suggested
follow-up questions")

# Use in synthesizer
structured_llm = llm.with_structured_output(QueryResponse)
response = structured_llm.invoke(synthesis_prompt)

```

#### Benefits:

- ☒ Guaranteed valid JSON
- ☒ Type safety
- ☒ Easier frontend integration
- ☒ Better error messages

## 7. 🎯 Token Optimization

#### Current Issues:

- Prompts are verbose (500-800 tokens each)
- Full database catalog sent to planner (1000+ tokens)
- Conversation history grows unbounded

#### Optimization Strategies:

##### A. Prompt Compression

##### B. Catalog Summarization

##### C. Conversation Summarization

##### D. Lazy Loading

#### Expected Savings:

- 📊 60-70% reduction in prompt tokens
- 💰 ~\$0.02 → \$0.005 per query
- ⚡ Faster LLM responses

## 8. 🧠 UI/UX Enhancements

#### Proposed Features:

- 📄 **Results Table View** - Toggle between chat and table display
- 🎬 **Movie Cards** - Rich display with posters, ratings, cast
- 📊 **Query Statistics** - Show token usage, cost, response time

- 🌑 **Dark Mode** - Theme switching
- 📄 **Export Results** - Download conversations as JSON/CSV
- 🗣️ **Voice Input** - Speech-to-text for queries
- 🌐 **Multi-language Support** - Full i18n for French/English
- 📱 **Mobile Responsiveness** - Optimize for mobile devices
- ⌨️ **Keyboard Shortcuts** - Power user features

## 9. 🛠️ **Testing & Quality Assurance**

**Current Gap:** No automated tests

**Coverage Goals:**

- Unit tests for each node
- Integration tests for workflow
- Performance benchmarks
- Regression tests for common queries

## 10. 🛡️ **Security & Privacy**

**Enhancements Needed:**

- 🛡️ API key encryption at rest
- 🚫 Input sanitization (SQL injection prevention)
- 🧑‍💻 PII detection and redaction in conversations
- 📋 Audit logging for all queries
- 🛑 Rate limiting and abuse prevention
- 🌐 HTTPS enforcement in production
- 🗙 Content filtering for inappropriate queries

## 11. 🚀 **Performance & Scalability**





**Optimization Opportunities:**

- ⚡ **Async Tool Execution** - Run SQL, Semantic, OMDB in parallel
- 💾 **Result Caching** - Cache common queries (Redis)
- 🔍 **Vector Index Optimization** - Use HNSW parameters tuning
- 📊 **Database Indexing** - Add indexes on common query columns
- 🔄 **Connection Pooling** - Reuse DB connections
- ☁️ **Deployment** - Docker + cloud hosting (AWS/GCP)
- 📦 **CDN Integration** - Cache static assets

## 12. 📊 **Analytics & Monitoring**

**Tracking Metrics:**

- 📈 Query latency by tool type
- 💰 Cost per query (token usage)
- 🎯 Tool selection accuracy (planner effectiveness)

-  User engagement metrics
-  Error rates and types
-  Most common queries and topics
-  Semantic search quality metrics

## Contributors

This project was developed as part of our Master's degree at **Albert School** in collaboration with **Mines Paris - PSL**.

### Team:

- Vincent Lamy & Alexandre Waerniers - Architecture, LangGraph workflow, semantic search

### Institution:

- Albert School (Paris, France)
- Mines Paris - PSL

---

## License

This project is licensed under the MIT License - see the LICENSE file for details.

---

## Acknowledgments



We would like to thank:

- **OpenAI** for providing the GPT-4 and embedding APIs
- **LangChain** team for the excellent orchestration framework
- **ChromaDB** for the vector database
- **Streamlit** for the intuitive UI framework
- **OMDb API** for movie metadata
- Our professors and advisors at Albert School and Mines Paris

---

## Contact

For questions, suggestions, or collaboration:

-  GitHub: [Vincent-20-100](#)
-  Institution: Albert School / Mines Paris - PSL

---

## Academic Context

This project demonstrates:

- ☒ Agentic AI architectures with LangGraph
- ☒ Multi-tool orchestration and planning

- ☒ RAG (Retrieval Augmented Generation) patterns
- ☒ Vector databases and semantic search
- ☒ Production-ready ML system design
- ☒ Full-stack AI application development

**Course:** Agentic Systems with RAG

**Year:** 2025 (M1)

**Institution:** Albert School x Mines Paris - PSL

---

★ If you found this project useful, please consider giving it a star! ★