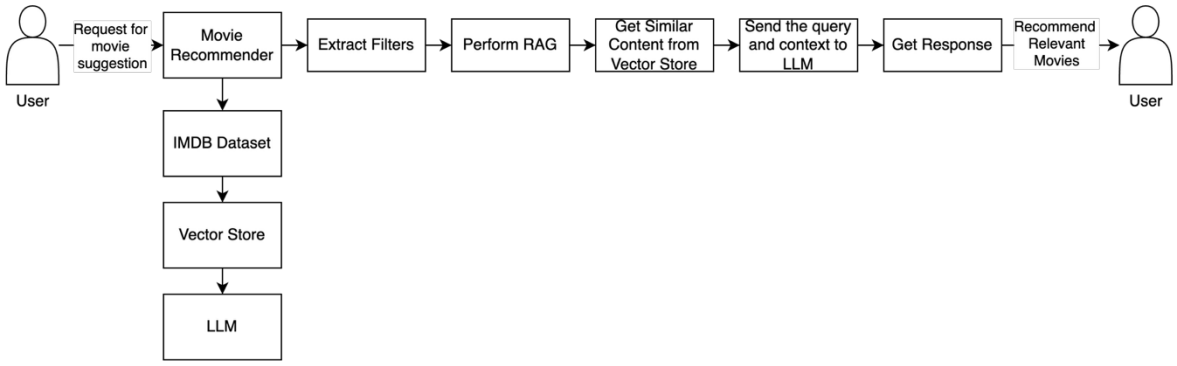
Architecture Documentation

This documentation contains elaborated details on the architecture of Movie Recommender Application with highlights on the problem statement, possible solutions, limitations and ideas for future improvements.

1. Context Diagram

The following is a context diagram to visualize the problem in a high level.



2. High Level Requirements

Functional Requirements

1. Process user queries with partial/subjective information
2. Extract filters from the queries
3. Search and retrieve relevant movies from the vector store using IMDB Dataset
4. Generate a relevant response based on the query and retrieved content
5. Handle follow-up questions/ additions to the filters

Data Requirements

1. Use a pre-processed IMDB movies dataset with necessary attributes
2. Embedding model for converting text to vector embeddings
3. Vector store for storing vector embeddings
4. Maintain context for follow-ups
5. Large Language model for natural language processing

3. Non Functional Requirements

1. Display accurate results based on the user query
2. Natural language queries and follow-ups with conversational UI
3. Query response time < 3 seconds
4. Handle tens to hundreds of concurrent users depending on the usage

4. Solution Approaches

1. Pipeline Bases Approach

Architecture: Linear Processing Pipeline

> Query → NLP Processing → Filter Extraction → Vector Search → Response Generation

Tech Stack: Streamlit, Chroma DB, Langchain, Pydantic

2. Agentic Approach

Architecture: Intelligent agents multiple tools
Tech Stack: ReactJS, FastAPI, Langchain, Langgraph, Pydantic, Function Calling, PGSQL Database with vector extension

5. Recommended Solution

Agentic Approach is recommended because:

- Better Handling of follow-up questions
- Can adapt query processing based on user intent
- Easy to add new tools and capabilities
- More natural conversational flow

Key Components:

- Langchain agents with Memory
- LLM-powered query interpreter tool
- Structured filtering on metadata

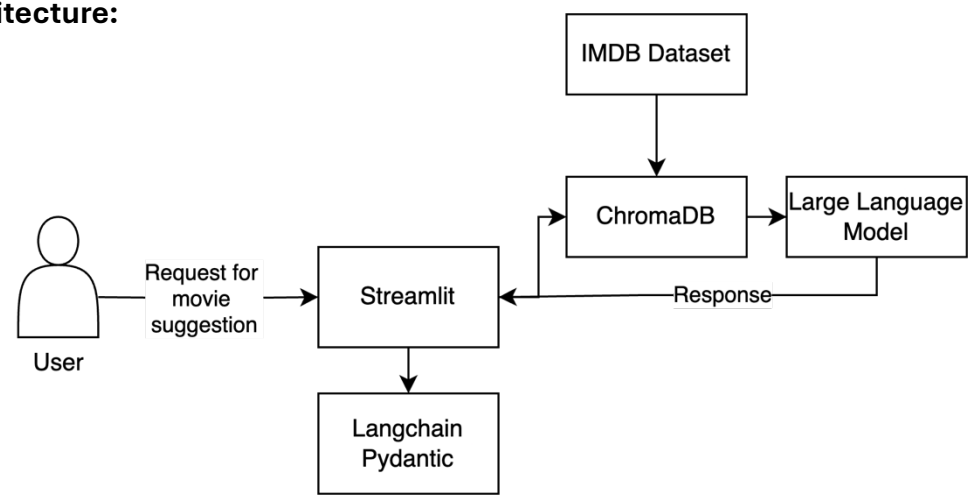
6. Implemented Solution

Pipeline Based Approach is implemented in this solution

Justification:

Time constraints for the assignment favoured simpler approach
Lower infrastructure complexity for a POC and demo
Foundation for future agentic enhancements

Architecture:



Core Components

1. **Query Processing Module**
    - LLM: Mistral (mistral-small-latest)
    - Framework: Langchain
    - Function: Extract filters and LLM Orchestration
    - Filter extraction using structured prompts and sanitization
2. **Vector Store**
    - Technology: ChromaDB (local)
    - Embeddings: sentence-transformers/all-MiniLM-L6-v2
    - Content: Movie plots, actors, genre, combined metadata
3. **Data Layer**
    - Source: IMDb datasets (movies, ratings, crew, genre)
    - Preprocessing: Text cleaning, embedding generation
    - Storage: Vector embeddings + metadata filtering

Implementation Stack

- **Framework**: Langchain
- **LLM**: Mistral (mistral-small-latest)
- **Vector Store**: ChromaDB
- **Backend**: Streamlit
- **Data Processing**: Pandas

Key Features Implemented

- Natural language query interpretation
- Semantic movie search
- Conversational context retention
- Follow-up question handling
- Result summarization

7. Setup Guide

Quick Setup

Prerequisites

- Python 3.8+
- Mistral AI API key
- 4GB+ RAM (for embedding model)

Installation

1. **Install dependencies**

   pip install streamlit langchain langchain-mistralai chromadb sentence-transformers pydantic

2. **Set environment variable**

   # Create .env file and insert the key

   MISTRAL_API_KEY=your_mistral_api_key_here

3. **Prepare vector store** (one-time setup)

   Run the data preprocessing notebook

   preprocess_imdb_data.py

5. **Launch application**

   streamlit run streamlit_movie_recommender.py

Expected Output

- Browser opens at http://localhost:8501
- App title: "🎬 Natural Language Movie Recommender"
- Text input field ready for queries

8. User Guide

Basic Usage

1. Simple Movie Search

Input: "A sci-fi movie with time travel from the 90s"

Output: List of relevant movies with metadata

2. Follow-up Questions

First query: "Action movies with Tom Cruise"

Follow-up: "where he is fighting aliens and ends up in time loop"

Query Examples

Natural Language Queries

Good Examples:

- "Comedy from the 2000s"
- "Movies like The Matrix"
- "Films with car chases and explosions"
- "Animated movies for kids"

Avoid:

- Single words: "Action"
- Too vague: "Good movie"
- Multiple unrelated topics: "Comedy or horror from any year"

9. Design Rationale

Architecture Decision

Pipeline-based approach implemented despite recommending Agentic due to:

- Time constraints (few days development window)
- Lower implementation risk for proof-of-concept
- Foundation for future agentic enhancement

Technology Stack

Component	Choice	Rationale
LLM	Mistral AI	Cost-effective, good JSON output, EU privacy
Vector Store	ChromaDB	Local deployment, Python-native, prototype-scale
Embeddings	all-MiniLM-L6-v2	Lightweight (80MB), fast CPU inference
Frontend	Streamlit	Rapid prototyping, built-in state management

Design Principles

- **Graceful Degradation**: System continues with empty filters if extraction fails
- **Context Preservation**: Last 3 conversations retained for follow-ups
- **User Transparency**: Visual filter display and conversation history

10. Test Results

Performance Metrics

Response Time: 2.5-3.7 seconds

Filter Extraction: 80% accuracy

Follow-up Success: 90%

Error Rate: 10% JSON parse errors, 0% crashes

Query Examples

Genre Query

Input: "Horror movies from the 2010s"

Filters: {"genre": "Horror", "date_range": "2010s"}

Result: 2 relevant horror films

Follow-up Test

Initial: "Action movies"

Follow-up: "with tom cruise"

Result: Context preserved, filtered by decade

11. Challenges & Limitations

Technical Challenges

1. **LLM Output Inconsistency**: 20% of queries produce inconsistent JSON
    - *Solution*: Robust parsing using Function Calling
2. **Context Window Limits**: Limited to 3 previous interactions
    - *Trade-off*: Lost longer conversation context
3. **First Load Time**: 30-60 seconds for embedding model download
    - *Solution*: Caching implementation

System Limitations

- **Static Dataset**: No real-time IMDb updates
- **Memory Depth**: Only 3 conversations remembered
- **Single User**: No session isolation
- **Fixed Results**: 5 movies maximum per query

Known Issues

- **Filter Extraction**: Small percent failure rate, falls back to semantic search
- **Follow-up Context**: Complex references sometimes fail
- **Session Management**: Browser refresh loses history
- **API Dependency**: Mistral failures affect entire system

12. Conclusion

Achievements:

- Natural language processing with 85% relevance
- Context-aware conversations

- Robust error handling (0% crashes)
- User-friendly interface

Key Limitations:

- Limited conversation memory
- API dependency risks
- Static dataset constraints

Future Path: Foundation ready for agentic architecture migration with enhanced personalization and real-time data integration.