Reflection: CinemaSleuths RAG Pipeline

A. Architecture & Design Decisions

The goal of this project was to design and implement a domain-specific Retrieval-Augmented Generation (RAG) system that could accurately answer questions using only the curated data. Our chosen domain (movie box office performance and related metadata) was well suited for a RAG architecture, as it combines structured numerical data, such as worldwide gross, domestic vs. foreign revenue, etc. with descriptive attributes such as genre, ratings, and production details. This allowed us to test both factual retrieval and natural-language generation in a constrained setting.

At a high level, the system follows a standard RAG pipeline: ingestion, chunking, embedding, retrieval, and generation. The ingestion stage loads a cleaned CSV file produced during our previous ETL assignment called etl_cleaned_dataset.csv. Each row of the dataset represents a movie and is converted into a natural-language document containing box office figures, audience reception, and production metadata. This transformation was an intentional design choice, as it allows structure tabular data to be indexed and retrieved using semantic similarity search.

For chunking, we used a RecursiveCharacterTextSplitter with a chunk size of approximately 1000 characters and an overlap of 200 characters. These values were chosen to balance context completeness with retrieval precision. Movie records are relatively compact, but including overlap helps preserve key information such as gross revenue figures that might otherwise be split across chunk boundaries.

For embeddings, we selected BAAI/bge-small-en-v1.5, an open-source embedding model that performs well on semantic retrieval tasks while remaining lightweight enough to run on a 16GB CPU-only virtual machine. This choice aligned with the project constraints and ensured that embedding generation was feasible without GPU acceleration. The embeddings are stored locally and indexed using FAISS, which provides fast similarity search and integrates cleanly with our custom pipeline.

The generation component uses Qwen1.5-4b-Chat, a local open-source language model that can run comfortably on CPU within the provided memory constraints. We intentionally avoided external APIs to ensure reproducibility and compliance within the assignment's emphasis on local models. Prompting was designed to strictly enforce grounding, instructing the model to answer only using retrieved context and to avoid hallucinating information not present in the data.

B. Retrieval Quality & Failure Analysis

Overall, retrieval quality was strong and consistent. For factual queries such as "Which movie had the highest worldwide gross?" or "What genres are associated with this film", the system reliably

retrieved the most relevant movie records from the FAISS index. Source snippets included in API responses demonstrated that the system was grounding its answers in the correct parts of the dataset..

However, during testing we observed an important limitation related to numerical reasoning. In one representative example, the system retrieved multiple movies along with their worldwide gross values, including the correct highest-grossing film. Despite having the correct information in context, the language model sometimes selected the wrong movie when asked to identify the maximum value. This failure was not due to retrieval errors but rather to the language model's limited ability to reliably compare numerical values when running locally on CPU. This issue highlights a well-known limitation of small language models: while they excel at language understanding and summarization, they are not always reliable at arithmetic or deterministic comparisons. Importantly, the system did not hallucinate incorrect data; instead, it misinterpreted correctly retrieved information. This distinction is critical because it demonstrates that the RAG pipeline itself was functioning as intended.

In a production setting, this limitation would ideally be addressed by routing certain deterministic queries (ex. "highest, " "lowest," "average") through programmatic logic rather than relying entirely on generative reasoning. For the purposes of this assignment, we documented this behavior as a known limitation and reflected on it as an engineering tradeoff rather than attempting to verify a solution.

C. API & Engineering Challenges

One of the primary engineering challenges was managing performance on a CPU-only environment. Both embedding generation and language model inference are computationally expensive operations, especially when performed repeatedly. Initially, restarting the Flask server caused the entire pipeline, including embedding computation and FAISS index construction, to be rebuilt form scratch, resulting in long startup times.

To address this, we implemented a simple but effective caching mechanism. Generated embeddings, the FAISS index, and processed document chunks are serialized and stored locally after the first run. On subsequent startups, the pipeline loads these cached artifacts instead of recomputing them. This reduced startup time from several minutes to just a few seconds and made the system far more practical to test and demonstrate.

Another challenge involved dependency management for local model loading. Using Hugging Face's device_map="auto" functionality required the additional accelerated library, which was not initially installed. This resulted in runtime errors that were resolved by explicitly installing the required dependency and restarting the server. This experience reinforced the importance of carefully reading error traces and understanding the runtime requirements of modern ML libraries.

Finally, managing latency for first-time inference required tuning generation parameters. By limiting the maximum number of generated tokens, we significantly reduced response time while still producing useful, concise answers. This tradeoff between completeness and responsiveness is an important consideration for real-world RAG systems.

D. Team Collaboration & Process

Our team consisted of three members: Richard, Krishna, and Nadia. We collaborated throughout the project with clear division and responsibility while maintaining shared oversight of the system's overall correctness and integration. Richard led the technical implementation, including designing and building the RAG pipeline, integrating the ETL dataset, implementing chunking, embeddings, FAISS indexing, caching optimizations, and developing and testing the Flask API on a 16GB CPU virtual machine. Krishna focused on the data understanding and retrieval quality, helping validate that key movie attributes were correctly represented in the ingested documents, testing retrieval relevance across different query types, and providing feedback on chunking and prompt design. Nadia contributed to the documentation, cleaning of the huge dataset from our last ETL project, evaluation, and reflection, assisting with repository organization, README clarity, and analyzing system behavior during testing (like identifying limitations in numerical reasoning for inclusion in the failure analysis). As a team, we used Git for version control and followed an interactive development process, testing each pipeline component independently before integration and documenting observed limitations as part of our final reflection.

E. Conclusion

This project successfully demonstrated a complete, end-to-end RAG system that integrates ETL, vector search, and local language model inference. While the system is not without limitations, particularly in numerical reasoning, it fulfills the core objectives of the assignment and reflects realistic engineering tradeoffs encountered in modern data systems. Most importantly, it highlights how careful architecture, grounding, and evaluation can produce a reliable and interpretable conversational AI system built entirely on local resources.