Reflection Paper

Group Members: Nicole Kwan, Yuhan Liu, Nishita Koya

I. Architecture & Design Decisions

Data: The CSV utilized for the final project reflected the file produced for the midterm project. To expand, the CSV file combined OMDb, Kaggle, and Sakila datasets, which all contained information about movie and rental information. Moreover, the PDF incorporated into the final project was titled "The determinants of box office performance in the film industry revisited." This article examines which factors help explain global box office revenue for films, which we thought would be valuable information for our "expert" system.

Ingestion & Chunking: We combined our cleaned ETL dataset (etl_cleaned_dataset.csv) with the box-office PDF to create one unified, searchable collection of text documents. We converted each movie row into a readable text block and attached lightweight metadata (title, year, genres, rating, film_id). For both the CSV and PDF pages, we used LangChain's RecursiveCharacterTextSplitter with 2,000-character chunks and 200-character overlap to approximate a 512-1024 token window while preserving context. We tagged each chunk with source information and wrote everything to ingested_documents.jsonl.

Embeddings: We took all of our ingested chunks from ingested_documents.jsonl and turned them into vector representations using the SentenceTransformers model BAAI/bge-small-en-v1.5. We first loaded every chunk's text and metadata, filtered out any empty lines, and then encoded the text in batches with normalization turned on. The resulting embedding matrix was saved to embeddings.npy, and we also saved the aligned texts.pkl and metadatas.pkl.

Vector Store: We took our saved embeddings (embeddings.npy) and built a FAISS inner-product index so we can do fast similarity search over all CSV and PDF chunks. We first loaded the embedding matrix, checked that it was 2D and float32, and then added all vectors to a faiss.IndexFlatIP, which treats our normalized embeddings as cosine-similarity vectors. We saved this index as faiss_index.bin and wrapped it in a VectorStore class that also loads the aligned texts.pkl and metadatas.pkl.

Retrieval: We wrapped our embedding model and FAISS vector store into a single Retriever class. We reused the same SentenceTransformer encoder (BAAI/bge-small-en-v1.5) to embed user queries, so the query and document vectors live in the same space. Then, we connected the encoder to our VectorStore, which loads the FAISS index alongside the aligned text chunks and metadata. The retrieve() method performs top-k cosine-similarity search ($k \geq 3$) and returns the

highest-scoring chunks with full metadata. This gives the RAG system a consistent way to pull grounded context for each question.

LLM & Prompting: We wired everything together into a RAGPipeline class that does end-to-end retrieval-augmented generation. Given a user question, we first called our Retriever to get the top-k most relevant chunks from the CSV + PDF corpus (with full metadata). We then built a grounded prompt that included a system message (Phi-3 must answer only from the provided context and cite sources) plus a numbered context block showing each chunk and its source. This prompt was passed to the local open-source model microsoft/Phi-3-mini-4k-instruct, which we ran on CPU using transformers. We sliced off just the generated continuation, trimmed extra trailing text, and returned a JSON-style result with the final answer and the exact sources used. This lets us answer both PDF questions (e.g., box office determinants) and CSV questions (e.g., highest IMDb rating).

II. Retrieval Quality & Failure Analysis

Failure Example (*Figures 1-2*): Because the query text was not put in quotes, Python interpreted the words (what, is, the, movie, inception) as variable names, not a string. This shows that retrieval quality not only depends on the vector store but also on correct query formatting. A malformed query prevents embedding and leads to an immediate system failure.

```
def main():
    #simple manual test
    r = Retriever()
    query = what is the movie inception?
    results = r.retrieve(query, k=5)

    print("\n[RETRIEVER] Sample results:")
    for i, r_i in enumerate(results, start=1):
        meta = r_i["metadata"]
        src = meta.get("source", "unknown")
        title = meta.get("title", "N/A")
        print(f"\nResult {i}:")
        print(f"  Score:   {r_i['score']:.4f}")
        print(f"  Source:  {src}")
        print(f"  Title:   {title}")
        print(f"  Preview: {r_i['text'][:200].replace('\\n', ' ')} ...")
```

Figure 1: Failure Query (retriever.py)

```
[RETRIEVER] Loading encoder: BAAI/bge-small-en-v1.5
[RETRIEVER] Loading VectorStore ...
[VS] Loading FAISS index from /content/faiss_index.bin ...
[VS] Loading texts from /content/texts.pkl ...
[VS] Loading metadatas from /content/metadatas.pkl ...
[VS] VectorStore ready with 16084 entries.
Object `inception` not found.
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-3421825357.py in <cell line: 0>()
     60
     61 if __name__ == "__main__":
---> 62     main()

/tmp/ipython-input-3421825357.py in main()
     45     r = Retriever()
     46     get_ipython().run_line_magic('pinfo', 'inception')
---> 47     results = r.retrieve(query, k=5)
     48
     49     print("\n[RETRIEVER] Sample results:")

NameError: name 'query' is not defined
```

Figure 2: Failure Returned Chunks (retriever.py)

Success Example (*Figures 3-4*): The system correctly identified the relevant movie records when asked which film had the highest IMDb rating. The returned answer also specifically included the movie title and its associated rating information, which was taken from the dataset. The system, additionally, was able to provide sources from the CSV dataset in terms of what relevant information it utilized to answer the user's question.

```
question2 = "Which movie in the dataset has the highest IMDb rating?"
result2 = rag.answer(question2, k=5)

print("\n================= QUESTION 2 =================")
print("Q:", question2)
print("\n=== ANSWER ===")
print(result2["answer"])
print("\n=== SOURCES USED ===")
for i, src in enumerate(result2["sources"], start=1):
    meta = src.get("metadata", {})
    title = meta.get("title", "N/A")
    origin = meta.get("source", "unknown")
    page = meta.get("page", None)
    print(f"[Source {i}] title={title} | source={origin} | page={page}")
```

Figure 3: Success Query (rag.py)

```
[RAG] Retrieving top-5 sources for: 'Which movie in the dataset has the highest IMDb rating?'
[RETRIEVER] Embedding query: 'Which movie in the dataset has the highest IMDb rating?'
[RETRIEVER] Searching top-5 in VectorStore ...
[RAG] Generating answer with Phi-3...

================= QUESTION 2 =================
Q: Which movie in the dataset has the highest IMDb rating?

=== ANSWER ===
The movie with the highest IMDb rating in the dataset is "Super Size Me" with a rating of 7.2 [Source 2].

=== SOURCES USED ===
[Source 1] title=the best movie | source=csv | page=None
[Source 2] title=super size me | source=csv | page=None
[Source 3] title=view from the top | source=csv | page=None
[Source 4] title=view from the top | source=csv | page=None
[Source 5] title=i can only imagine | source=csv | page=None
```

Figure 4: Success Returned Chunks (rag.py)

III. API & Engineering Challenges

For the API and engineering side, we ran into a few practical challenges. For starters, loading both the BGE encoder and the Phi-3-mini-4k-instruct model on CPU took quite some time, so our first request was dominated by model initialization. To keep runtime reasonable, we loaded the RAGPipeline once at the top of app.py (as a global object) instead of re-initializing it per request. We also capped the max_new_tokens in rag.answer() so responses stayed fast enough for interactive use. Furthermore, we added basic error handling around the Flask endpoint so that malformed requests didn't crash the server. In /api/ask, we explicitly checked for a missing "question" field and returned a 400 JSON error instead of throwing a Python exception. At the pipeline level, earlier files (like ingest.py and embeddings.py) already caused clear FileNotFoundErrors if the CSV, JSONL, or embedding artifacts were missing, which helped explain failures such as "no FAISS index found" instead of silently returning empty results. We also had to think about environment issues. For example, the vector store artifacts (texts.pkl, metadatas.pkl, faiss_index.bin) were generated with Python 3.12, so we noted in the README that the professor should run everything with Python 3.12 to avoid compatibility errors.

The hardest bug we hit was getting the API to actually accept requests in different environments. In Colab, the Flask server blocked the notebook cell, so our first requests.post(...) calls failed with ConnectionRefusedError because the server wasn't actually running when we hit it from another cell. On local macOS, we initially tried binding to an external IP, which produced "Can't assign requested address," and then ran into "ModuleNotFoundError: No module named 'rag'" because the folder structure didn't match our imports. We fixed this by (1) always running app.py from the api/ directory so Python can see rag_pipeline/ on the path, (2) using host="0.0.0.0", port=8000 for the server, and (3) testing the endpoint locally with curl once the server log showed it was listening. These details ended up taking more time than the main RAG logic, but they were necessary to make the system reachable via a real /api/ask endpoint.

Furthermore, one of the biggest practical issues we ran into was latency from the LLM. When we wired the API to our RAG pipeline, the server would often hang at "[RAG] Generating answer with Phi-3..." for a long time, and sometimes it felt like the request never finished. This likely happened because Phi-3-mini is still a relatively large model and we were running it on CPU. Additionally, our prompts included several retrieved chunks. We attempted to address this issue by keeping a single global RAGPipeline instance alive in app.py (rather than reloading the model per request) and by capping max_new_tokens.

IV. Team Collaboration & Process

We built most of the RAG pipeline in Google Colab, sharing notebooks, testing outputs, and committing stable versions to GitHub. Nicole drafted the initial pipeline structure, while Yuhan and Nishita repeatedly tested each stage and identified issues with chunking. As the project grew, Nicole handled API integration and the optional web UI, and both she and Nishita worked on the reflection paper.

Our Git workflow was intentionally simple because most of our development happened inside shared Google Colab notebooks. We wrote and tested nearly all components of the RAG pipeline directly in Colab, where we could run code quickly and debug together. Whenever a script or output reached a stable working state, we exported the file and uploaded it to GitHub. Instead of using branches or complex pull-request structures, we relied on frequent Colab testing, direct communication, and straightforward GitHub updates. This allowed all members to collaborate without dealing with heavy Git processes.

One lesson we learned was how important it is to maintain a clean and consistent directory structure. This is especially because minor path inconsistencies caused import errors like “ModuleNotFoundError: No module named 'rag',” which took time to untangle. Another takeaway was that binary artifacts (such as .pkl and .bin files) do not behave well in version control. Instead, storing them in a dedicated folder (outputs/) and documenting how to regenerate them simplified collaboration.