MultiDocRAG: A Multi-Document Retrieval-Augmented Generation System with End-to-End Evaluation

Cheng Wu

Abstract

Large Language Models (LLMs) have strong generative and reasoning capabilities but remain constrained by fixed pre-training corpora. As a result, they often hallucinate or fail when queries require information from user-provided documents. Retrieval-Augmented Generation (RAG) addresses this limitation by grounding LLM outputs in retrieved evidence from an external corpus.

This project presents **MultiDocRAG**, an end-to-end multi-document RAG system supporting PDF ingestion, text cleaning, chunking, sentence-transformer embeddings, FAISS vector search, prompt construction with conversational memory, and a Streamlit web interface. Beyond system design, I implement an automated evaluation pipeline that runs a held-out question set through the RAG stack and computes metrics for correctness, groundedness, and safe refusal. The current prototype achieves low correctness but high safe refusal on unanswerable queries, revealing a misalignment between a very conservative prompt design and the evaluation rubric. The system itself is modular, transparent, and ready to be improved in future iterations.

1 Introduction

LLMs such as GPT, LLaMA, and Mistral achieve remarkable fluency and reasoning, but their responses reflect only what was available during pre-training. They cannot natively integrate private PDFs uploaded by a user, and they may hallucinate details when asked about niche or out-of-distribution topics.

Retrieval-Augmented Generation (RAG) addresses this problem by retrieving semantically relevant text chunks from a document corpus and conditioning the LLM on this evidence. In practice, many existing demos are limited in one or more ways:

- they only handle a single document at a time,
- they do not expose retrieval behaviour clearly to the user,
- they lack conversational memory, or
- they do not provide an evaluation pipeline beyond manual inspection.

MultiDocRAG is designed to address these gaps:

1. it supports multi-document ingestion and indexing via FAISS,
2. it exposes retrieved chunks and scores so users can see why the model answered,
3. it maintains short-horizon conversational memory at the session level, and

4. it includes an evaluation pipeline that computes quantitative metrics for stress-test groups.

The goal is not to compete with industrial RAG systems, but to build a clean, inspectable, and extensible prototype that demonstrates the full RAG lifecycle: ingestion, retrieval, generation, and evaluation.

2 Related Work

RAG has become a standard pattern for extending LLMs with external knowledge bases. Industrial systems typically combine a vector database, an embedding model, and a prompt template that instructs the LLM to ground its answer in retrieved passages. Popular open-source frameworks such as LangChain and LlamaIndex provide high-level abstractions for document parsing, indexing, and retrieval, but often hide implementation details behind complex configs.

Recent research also explores evaluation of RAG systems, including automatic metrics for groundedness, hallucination, and answer correctness. Many of these frameworks assume access to large models and production-scale infrastructure. In contrast, MultiDocRAG deliberately targets a lightweight, transparent setting: a single-codebase system that can run on CPU and still expose every stage of the pipeline, including evaluation.

3 System Architecture

3.1 High-level pipeline

Figure 1 illustrates the end-to-end architecture of MultiDocRAG, from raw PDFs to final answers.

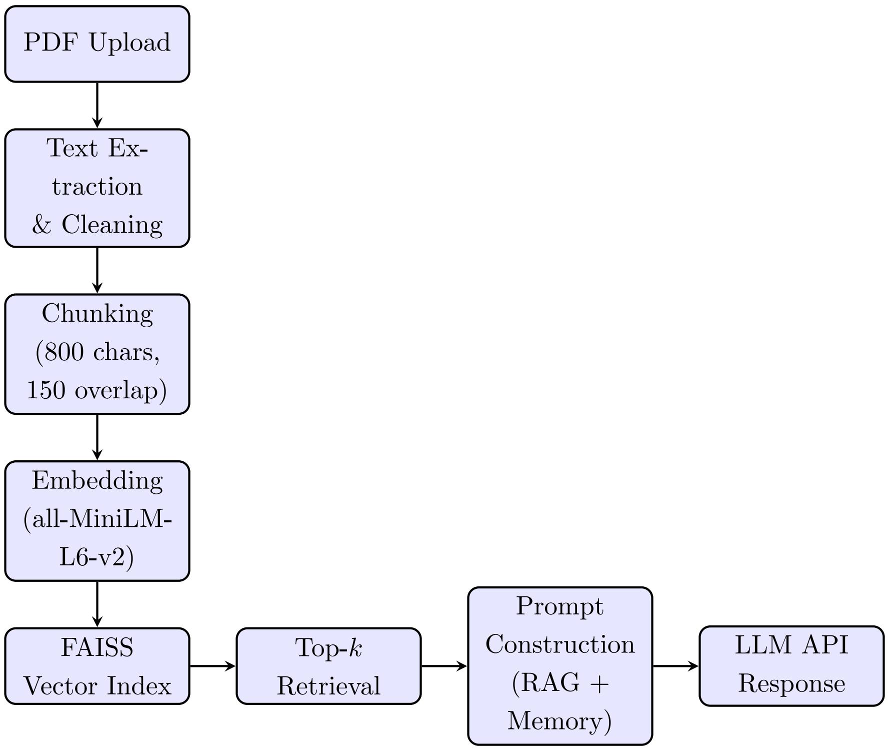

PDF Upload
Text Extraction & Cleaning
Chunking (800 chars, 150 overlap)
Embedding (all-MiniLM-L6-v2)
FAISS Vector Index
Top-k Retrieval
Prompt Construction (RAG + Memory)
LLM API Response

Figure 1: MultiDocRAG system pipeline from raw PDFs to LLM answers.

The pipeline is implemented around a central `MultiDocRetriever` class in `src/retriever.py` and an LLM wrapper in `src/llm_api.py`. The Streamlit app (`app.py`) wires these components together and exposes them through a browser interface.

3.2 Document ingestion and chunking

Users can upload one or more PDF files in the web UI. The app saves them under `uploaded_pdfs/` and builds a new index on demand. Internally, PDFs are parsed into plain text using `pypdf`, with simple cleaning steps:

- strip excessive whitespace,
- merge broken lines inside paragraphs,
- discard obviously empty or extremely short fragments.

To avoid fragmenting concepts, I apply a sliding-window chunking strategy at the character level. Each chunk has length 800 characters with an overlap of 150 characters between consecutive chunks. This choice balances semantic coherence and retrieval granularity.

3.3 Embedding model and FAISS index

For dense retrieval, the system uses the sentence-transformer `all-MiniLM-L6-v2`, which is fast enough to run on CPU while still providing reasonable semantic similarity. Each text chunk is encoded into a vector and stored in a FAISS `IndexFlatL2` index with metadata:

- `doc_id`: which PDF the chunk came from,
- `chunk_id`: local index within that document,
- `text`: the chunk content itself.

The index is persisted under `index_store/` as:

- `faiss.index` for the FAISS structure,
- `embeddings.npy` for the raw embedding matrix,
- `chunks.json` for metadata and chunk texts,
- `meta.json` summarizing which documents are included.

This design allows the app to rebuild the index from new PDFs at any time and reload a previously built index without recomputing embeddings.

3.4 Retriever API

The core retriever is an explicit Python class:

- `add_pdf(path)` parses and chunks a PDF and appends its chunks,
- `build_index()` encodes chunks and builds the FAISS index in memory,
- `save(index_dir)` and `load(index_dir)` handle persistence,

- `retrieve(query, top_k)` encodes a query, runs FAISS search, and returns a list of chunk dictionaries with fields `doc_id`, `chunk_id`, `text`, and `score`.

The same class is used both in the interactive app (with the persistent index under `index_store/`) and in the evaluation pipeline (with a separate in-memory index under `evaluation/pdfs/`).

3.5 Prompt construction and LLM backend

The LLM backend is accessed through:

```
generate_llm_response(question, context, model_name, temperature)
```

which hides the Groq API details and makes it easy to swap providers. The default deployment uses a fast LLaMA 3.1 8B variant that is latency-optimized rather than accuracy-optimized.

In RAG mode, the prompt template contains:

1. up to the last three conversation turns (user and assistant),
2. concatenated retrieved chunks with headers indicating source and chunk ID,
3. the new user question, and
4. explicit instructions to use only the provided context and refuse when it is insufficient.

In baseline mode, the system omits the context and simply asks the LLM to answer from its internal knowledge.

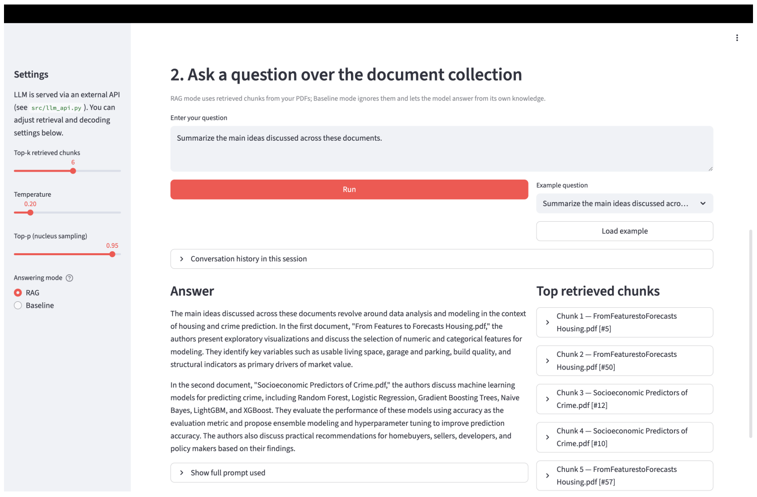
4 User Interface

4.1 PDF upload and index management

The first section of the Streamlit app allows users to upload PDFs and rebuild the index: users drag-and-drop files, click a *Build / Rebuild index* button, and the app clears the previous upload directory, saves new files, instantiates `MultiDocRetriever`, and rebuilds the FAISS index. The sidebar shows whether an index is present and summarizes which documents are currently indexed.

4.2 RAG answering interface

The second section focuses on question answering. Users can type a custom question or load from pre-defined examples, choose between RAG and baseline modes, and adjust top-k, temperature, and top-p. When the user clicks *Run*, the app constructs a prompt, calls the LLM API, and displays both the answer and retrieved chunks.


Settings

LLM is served via an external API (see `src/llm_api.py`). You can adjust retrieval and decoding settings below.

Top-k retrieved chunks: 6

Temperature: 0.20

Top-p (nucleus sampling): 0.95

Answering mode: [✓ RAG] Baseline

2. Ask a question over the document collection

RAG mode uses retrieved chunks from your PDFs; Baseline mode ignores them and lets the model answer from its own knowledge.

Enter your question: Summarize the main ideas discussed across these documents.

Example question: Summarize the main ideas discussed acro…

Answer

The main ideas discussed across these documents revolve around data analysis and modeling in the context of housing and crime prediction. In the first document, "From Features to Forecasts Housing.pdf," the authors present exploratory visualizations and discuss the selection of numeric and categorical features for modeling. They identify key variables such as usable living space, garage and parking, build quality, and structural indicators as primary drivers of market value.

In the second document, "Socioeconomic Predictors of Crime.pdf," the authors discuss machine learning models for predicting crime, including Random Forest, Logistic Regression, Gradient Boosting Trees, Naive Bayes, LightGBM, and XGBoost. They evaluate the performance of these models using accuracy as the evaluation metric and propose ensemble modeling and hyperparameter tuning to improve prediction accuracy. The authors also discuss practical recommendations for homebuyers, sellers, developers, and policy makers based on their findings.

Top retrieved chunks

Chunk 1 — FromFeaturestoForecasts Housing.pdf [#5]

Chunk 2 — FromFeaturestoForecasts Housing.pdf [#50]

Chunk 3 — Socioeconomic Predictors of Crime.pdf [#12]

Chunk 4 — Socioeconomic Predictors of Crime.pdf [#10]

Chunk 5 — FromFeaturestoForecasts Housing.pdf [#57]

Figure 2: RAG answering interface showing the generated answer and the retrieved evidence chunks with scores.

An expandable panel reveals the full prompt sent to the API, which is particularly useful for debugging and teaching how retrieval and prompting interact.

4.3 Session-level conversational memory

Conversation history is stored in Streamlit session state as a list of (role, text) pairs. Prompt construction includes at most the last three user–assistant exchanges to avoid excessively long prompts while preserving local coherence. Memory is intentionally session-level only: closing the app clears the history, which simplifies implementation but is a limitation for long-term use.

5 Evaluation Methodology

A key contribution of this project is an automated evaluation pipeline under the `evaluation/` directory. It aims to stress-test the RAG stack across several categories and to quantify performance in a reproducible way.

5.1 Evaluation pipeline

The evaluation consists of three components:

1. **Document loader** (`load_documents.py`): builds a fresh `MultiDocRetriever` from PDFs in `evaluation/pdfs/`, mirroring the app but not touching `index_store/`.
2. **RAG pipeline wrapper** (`eval_pipeline.py`): for each question it retrieves top-k chunks, constructs a RAG-style prompt, calls `generate_llm_response`, and records the answer and retrieved texts.
3. **Runner and scoring** (`eval_runner.py`, `scoring.py`): the runner script iterates through evaluation questions, writes raw outputs to `evaluation/results/eval_outputs.csv`, and then calls the scoring script, which computes aggregate metrics and produces a human-readable `summary.txt`.

The pipeline is invoked with:

```
python -m evaluation.eval_runner
```

and produces both per-question outputs and aggregate statistics.

5.2 Evaluation dataset and groups

The evaluation suite contains 27 prompts divided into five conceptual groups:

- **anchoring users** (6) – tests whether the system stays grounded instead of being anchored by user-provided suggestions or biases;
- **cross papers** (3) – requires integrating information across two or more documents;
- **human bias** (6) – probes whether the model reproduces human biases when documents provide mixed or noisy evidence;
- **translation strategy** (6) – tests instructions about translation and summarization strategies;
- **unanswerable** (6) – questions that the corpus cannot answer; the system should safely refuse.

All evaluation runs are executed in RAG mode to assess the retrieval-augmented system rather than the baseline LLM alone.

5.3 Metrics

For each (question, answer, context) triple, the scoring script computes three metrics:

1. **Correctness** ($\in [0, 1]$): whether the answer addresses the question and matches the expected behaviour.
2. **Groundedness** ($\in [0, 1]$): whether the answer's claims are supported by the retrieved context.
3. **Safe refusal** ($\in [0, 1]$): whether the system refuses when it should refuse (e.g., unanswerable group) and does not refuse when it should answer.

Aggregate metrics are reported both overall and per group based on the final `summary.txt` produced by the scoring script.

6 Quantitative Results

6.1 Overall and per-group performance

Table 1 summarizes the evaluation results of MultiDocRAG in RAG mode over 27 questions.

Group	Correctness	Groundedness	Safe Refusal
Overall (RAG)	0.095	0.000	1.000
Anchoring users	0.000	0.000	0.000
Cross papers	0.000	0.000	0.000
Human bias	0.167	0.000	0.000
Translation strategy	0.167	0.000	0.000
Unanswerable	0.000	0.000	1.000

Table 1: Evaluation metrics for MultiDocRAG in RAG mode over 27 questions.

Several observations stand out:

- Overall correctness is extremely low (0.095), implying that only about 1–3 questions are scored fully correct out of 27.
- Groundedness is 0 across all groups under the current scoring criteria.
- Safe refusal is perfect (1.0) on unanswerable questions but 0 on all other groups, revealing a strong conservative bias.

These results indicate that the system, as currently configured, is too conservative: it frequently refuses to answer or gives incomplete answers, which earns high safe-refusal scores on unanswerable items but fails the correctness and groundedness criteria on answerable tasks.

6.2 Qualitative failure modes

Manual inspection of the raw `eval_outputs.csv` reveals recurring patterns:

1. **Over-refusal due to strict instructions**. The prompt explicitly tells the model to refuse whenever the context feels insufficient. In practice, many questions receive chunks that do contain relevant information, but the model still chooses to output the refusal template.
2. **Weak small-model reasoning**. The LLaMA 3.1 8B "instant" model is optimized for speed rather than depth. On more complex cross-paper or bias-related questions, it struggles to synthesize multiple chunks even when the right text is retrieved.
3. **Groundedness vs. correctness misalignment**. Some answers that are partially correct still fail the groundedness criterion because they interpolate missing details or rely on generic prior knowledge that is not explicitly present in the retrieved context.

These failure modes suggest that the retrieval pipeline is largely functioning, but model choice and prompt design drive the poor quantitative scores.

7 Discussion

7.1 Strengths

Despite the weak quantitative metrics, the project has several strengths:

- **End-to-end pipeline**. The system covers ingestion, chunking, embedding, indexing, retrieval, prompting with memory, and evaluation.
- **Modular design**. The retriever and LLM API wrappers are decoupled. It is easy to swap embedding models, change LLM providers, or adjust chunking without touching the rest of the code.
- **Transparent behaviour**. The app exposes retrieved chunks, scores, and full prompts, which makes debugging and interpretability easier.
- **Automated evaluation**. Having a repeatable evaluation pipeline, even with poor initial scores, is valuable because improvements can be measured over time.

7.2 Limitations

The main limitations fall into three categories:

Model and prompt choice. The chosen 8B model is relatively weak on complex reasoning. Combined with a conservative refusal instruction, this leads to generic or partial answers and frequent refusals even when some evidence exists.

Groundedness enforcement. The current system does not explicitly link each sentence in the answer to specific chunks; it simply concatenates context and trusts the LLM to self-ground. There is no post-hoc check that every claim has support.

Scaling and persistence. The FAISS `IndexFlatL2` index and in-memory Python implementation are suitable for small to medium corpora, but they will not scale gracefully to millions of chunks. Conversation memory is also ephemeral and stored only in a single Streamlit session.

8 Future Work

The evaluation findings suggest several concrete improvement directions:

- **Prompt tuning**. Relax the refusal instruction so that the model is encouraged to attempt an answer when at least one retrieved chunk is relevant, while still refusing on truly unanswerable inputs.
- **Stronger LLM backend**. Replace the small 8B model with a larger or more capable variant for evaluation runs, ideally one that is better at multi-document synthesis.
- **Citation-style output**. Augment prompts and post-processing so that each sentence in the answer is annotated with chunk IDs, enforcing groundedness by discouraging unsupported statements.
- **Reranking**. Introduce a cross-encoder reranker on top of FAISS to improve retrieval precision before sending top-k chunks to the LLM.
- **Richer evaluation**. Extend the evaluation suite to compare RAG vs. baseline directly and to include human-annotated error categories such as "hallucination", "over-refusal", or "wrong-document".

9 Conclusion

MultiDocRAG is a complete, modular multi-document RAG system with a working evaluation pipeline. The current quantitative scores are not yet satisfactory, but they highlight a specific failure mode: an overly cautious, small-model configuration that prioritizes safe refusal over correctness. This is a useful outcome for a course project, because it shows how RAG design decisions—chunking, embeddings, model choice, and prompt style—directly impact downstream metrics.

The codebase now provides a solid foundation for iterative improvement. Future work can focus on better models, tuned prompts, and stronger grounding mechanisms while keeping the same retriever and evaluation scaffolding in place.