Northeastern University, Khoury College of Computer Science

CS 6120 Natural Language Processing

Paper Q&A: An LLM-Based Assistant for Academic Reading

Final Report

Tongle Yao, Xuedong Pan, Taiwei Cui

yao.to, pan.xue cui.ta,

1 Executive Summary and Abstract

Our team presents a Retrieval-Augmented Generation (RAG) system adapted for academic literature. Our system allows users to fetch academic papers from Arxiv and query them in natural language. The system leverages large language models and a vector retrieval pipeline to deliver accurate, grounded answers, reducing hallucinations often found in LLMs. This solution benefits students, researchers, and engineers who seek efficient access to dense scientific knowledge. Our pipeline includes data acquisition, preprocessing, chunking, vector embedding, retrieval, and answer generation, with the final user-friendly interface. The modular design enables scalability and extensibility.

2 Motivation and Impact

With the exponential growth of academic literature, quickly extracting meaningful insights has become an overwhelming task. Our RAG system provides contextualized question-answering from academic papers. It empowers users to upload or retrieve papers and receive natural language answers based on relevant sections. This addresses challenges in comprehension, literature synthesis, and technical onboarding.

The domain of our system is intentionally scoped to computer science and machine learning literature. These fields are characterized by rapid publication cycles, evolving terminology, and dense technical content, making it difficult for readers to keep up with current developments or understand papers outside their immediate expertise. We selected this domain both for its practical impact and because it allows us to prototype and validate our system within a constrained yet relevant scope, given limited time and computational resources.

Our objective is to bridge the gap between unstructured academic content and accessible, query-driven understanding. By integrating vector-based retrieval with large language models, we aim to offer an intelligent assistant that can support literature comprehension, accelerate review, and enhance research workflows. The system improves information accessibility and reduces the time spent interpreting dense academic texts.

3 Background and Related Work

Recent advances in Retrieval-Augmented Generation (RAG) have demonstrated strong potential in addressing knowledge-intensive tasks where traditional LLMs tend to hallucinate. The RAG framework introduced by Lewis et al. (2020) [1] combines dense retrieval with generative language models to produce context-aware, accurate responses. Building upon this foundation, our system adopts a similar architecture, tailored specifically for academic literature in computer science and machine learning.

One notable work in this space is PaperQA [2], which applies RAG to scientific documents, highlighting the importance of chunking granularity, retrieval accuracy, and prompt design. We extend these ideas by designing a fully modular pipeline capable of handling arbitrary academic PDFs through structural parsing and vector-based retrieval. We integrate GROBID [3], an open-source library for parsing scholarly documents into TEI XML, which enables us to extract structured content such as titles, abstracts, and section headers. These structured elements form the basis for our semantic chunking and significantly improve retrieval relevance.

To perform semantic search, we leverage Milvus as our vector database, chosen for its scalability and speed in approximate nearest neighbor search. For embeddings, we use bge-base-en-v1.5, an English-language embedding model that produces dense vectors suitable for academic text. All embeddings are generated on pre-processed, section-tagged chunks of papers and indexed for efficient top-k retrieval.

4 Approach

4.1 Data Collection and Preprocessing

Our data is derived from public academic papers published on Arxiv.org within the domains of computer science and machine learning, specifically between November 2024 and April 2025. These papers were retrieved using the official arXiv API based on relevant categories (cs.AI and cs.GL). For each paper, we downloaded:

- Metadata (title, authors, abstract, publication date, arXiv ID, and PDF URL)
- Full-text PDFs

We processed this dataset using a modular pipeline built around open-source tools and custom scripts as described in the graph below. The pipeline includes:

- arxiv_downloader.py: Fetches metadata and PDFs
- pdf_processor.py: Extracts logical structure using GROBID, converting PDF into TEI XML

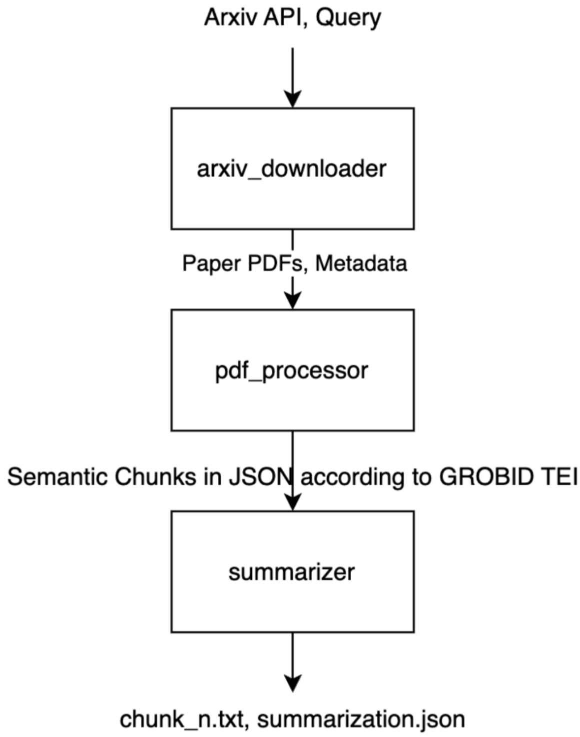


Figure 4.1: Data Collection and Preprocessing Pipeline

- summarizer.py: Summarizes long chunks or abstracts using LLM (gemini-2.0-flash-lite in our example)

Processed papers are chunked into semantically meaningful segments and stored as:

- metadata.json
- chunk_1.txt, chunk_2.txt, ... chunk_n.txt
- summarization.json

4.2 Embedding

The embedding module serves as a crucial component in our paper retrieval and question-answering system. It bridges raw document content with vector-based semantic search capabilities by transforming both paper metadata and user queries into vector representations.

The workflow begins with structured and pre-processed academic papers, which are first crawled and segmented into manageable chunks of content. Each summarization of the paper chunk is passed to an embedding model that converts chunk summarization into 768-dimensional dense vectors. These vectors are stored in a Milvus vector database along with relevant metadata to support similarity search operations. Concurrently, the original sliced content is stored in a MinIO object storage service for persistent retrieval.

When a user inputs a query via the backend interface, the system uses the same embedding model to transform the input into a query vector. This query vector is used to perform a top-k

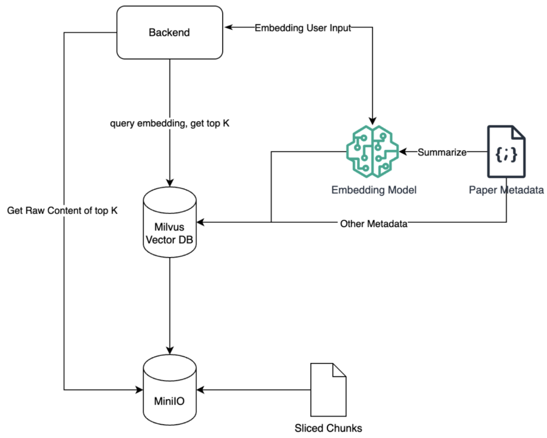


Figure 4.2: Embedding Pipeline

nearest-neighbor search against the vector database, retrieving the most semantically relevant paper chunks.

4.2.1 Embedding Model

To support effective semantic retrieval in our system, we required an embedding model that balances accuracy with computational efficiency. Our selection criteria included:

- Strong performance on English input, as the majority of our corpus consists of English academic papers
- Compatibility with resource-constrained environments, with the ability to run efficiently on CPUs without requiring GPU acceleration
- Effectiveness in prompt-based embedding, inspired by the paper PromptBERT: Improving BERT Sentence Embeddings with Prompts [5], which demonstrates that embedding quality can be significantly improved by task-specific prompts.

We ultimately selected bge-base-en-v1.5 as our primary embedding model. This model supports English retrieval tasks and prompt-based encoding. For instance, during query embedding, we prepend the user input with the prompt: "Represent this sentence for searching relevant passages:" This helps the model generate context-aware embeddings that better align with document representations in the vector database.

The bge-base-en-v1.5 produces 768-dimensional embeddings and offers strong performance on benchmarks such as MTEB (Massive Text Embedding Benchmark), while remaining lightweight enough for efficient CPU inference. This makes it well-suited for real-world applications where computing resources may be constrained.

4.2.2 Vector Database

To enable efficient similarity-based retrieval of academic content, our system integrates the Milvus vector database. Given our current data volume and architectural simplicity requirements, we deployed Milvus in standalone mode, which provides a lightweight yet robust solution.

Each chunk of processed academic content is embedded into a 768-dimensional vector using the bge-base-en-v1.5 model. The choice of 768 dimensions is driven by both the model's output specification and our summarization step, which ensures each text chunk under 512 tokens.

For indexing and retrieval, we configure Milvus to use the IVF_FLAT index type in combination with cosine similarity as the distance metric. Cosine similarity emphasizes the angular distance between vectors, making it more aligned with semantic proximity. In contrast, alternatives like L2 distance are better suited for modalities like images, while inner product (IP) is more appropriate for non-normalized vectors.

For database operations, we use the official Python SDK pymilvus, which offers seamless integration with our Python-based backend. Additionally, we leverage Attu, the official Milvus visualization tool, for managing collections as it provides a nice interface.

4.2.3 Storage

To support both persistent storage and system interoperability, our project adopts MinIO as the object storage backend. MinIO is a S3-compatible object storage system that offers excellent portability and scalability. Its compatibility with the Amazon S3 API enables seamless integration with popular tools and libraries. This ensures that our system remains decoupled from os-specific file management and can easily transition to a cloud-native setup if needed.

MinIO serves two primary roles within our architecture:

- Storage of Original Paper Chunks: While the embeddings are computed based on summarized segments of the papers, we retain the original text slices for reference and display purposes. These original slices are stored in MinIO, and the corresponding object keys are saved alongside the vector representations in the Milvus database.
- Persistent Storage Backend for Milvus: Milvus relies on object storage for durability, particularly in standalone mode. Although Milvus conducts most of its search operations in memory to ensure low-latency vector retrieval, the underlying data and index snapshots are persisted to MinIO.

4.3 Backend & User Interface

The backend of our system orchestrates the entire retrieval-augmented generation (RAG) workflow, bridging the user interface, embedding engine, vector database, and large language model (LLM).

The workflow begins when a user submits a question through Streamlit. This input is first routed to a Rephrase Agent, which is responsible for rewriting the user's question to make it more concise.

The rephrased query is then sent to the embedding service through API to retrieve the most relevant paper chunks. These chunks, along with the rephrased query and prior chat history, are passed to the Query Agent. The Query Agent assembles this information into a well-structured prompt, which is then sent to the LLM for final answer generation.

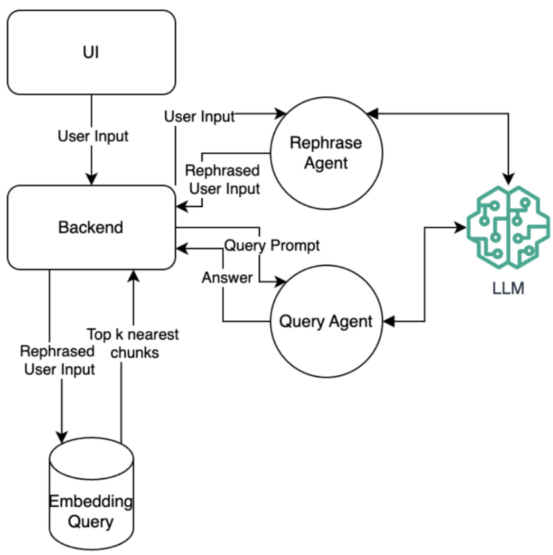


Figure 4.3: Backend & UI Architecture

4.3.1 LLM

To support natural language generation and complete the final step of the RAG pipeline, we utilize Ollama, a lightweight and flexible LLM framework that allows us to easily run and experiment with various open-source language models in a local environment.

After testing multiple candidate models, we selected LLaMA3-6B as our production model. LLaMA3-6B strikes a practical balance between performance and resource consumption. It possesses sufficient language generation capabilities for our academic question-answering task while maintaining a relatively small parameter size that enables low latency on an instance with a single T4 GPU.

A critical factor in choosing LLaMA3-6B was its lack of access to the latest academic content, as it is a model trained on a pre-2023 dataset and does not have internet access. Ensuring the LLM itself lacks direct knowledge of newly published papers, we can better assess the effectiveness of our RAG pipeline in injecting relevant, the latest knowledge into the response process.

4.3.2 Prompt

Our system employs two specialized agents—Rephrase Agent and Query Agent—to construct effective prompts for both retrieval and answer generation.

The Rephrase Agent reformulates user input by combining it with recent conversation history through a prompt template. Its goal is to eliminate ambiguity and pronouns, producing a concise and self-contained question. This step is necessary because our embedding model has a context limit of 512 tokens, making it impractical to include full conversation history in retrieval. The rephrased query improves both embedding precision and retrieval relevance.

The Query Agent is responsible for answer generation. It constructs a prompt that includes the rephrased question, relevant paper chunks retrieved from the vector database, and optionally, recent dialogue context. This prompt is then passed to the LLM. If the LLM uses content from the provided chunks, the final answer includes inline citations pointing to the source papers.

This prompt strategy ensures accurate retrieval, maintains context relevance, and increases answer transparency.

4.3.3 User Interface

The user interface of our system is built using Streamlit, a lightweight web framework designed for rapid development of data-driven applications. Streamlit provides a clean and interactive chat interface that closely resembles standard dialogue systems, making it intuitive for users to ask questions and view model responses.

Its built-in components and layout system allowed us to quickly deploy a functional prototype without extensive front-end development. This streamlined the integration between the user-facing interface and the backend pipeline, enabling seamless communication with the rephrase, retrieval, and answer generation modules.

5 Data Sources and Data Analysis

Data Sources and Data Analysis Our dataset comprises 6132 academic papers sourced from arXiv.org, filtered by relevant categories such as cs.AI and cs.GL and published between November 2024 and April 2025. These papers collectively yielded 135,614 semantically segmented text chunks, with each paper contributing approximately 22 chunks on average. Of these chunks, 41,615 chunks (30%) were automatically summarized using an LLM model (gemini-2.0-flash-lite), selected based on a length threshold (300 words) to be able to store as the query key in the vector database.

The resulting data is structured in a consistent, modular format:

- metadata.json: Contains paper-level information such as title, authors, abstract, publication date, and arXiv ID.
- chunk_n.txt: Stores individual logical chunks extracted from the full-text PDF.
- summarization.json: Maps each chunk to its corresponding LLM-generated summary.

An example chunk_n.txt may contain technical content like:

"Recurrent neural networks, long short-term memory..."

While the corresponding summarization.json provides a concise summary for retrieval tasks:


```
[
    {
        "chunk_file": "chunk_1.txt",
        "summaries": [
            {
                "prompt": "1",
                "content": "Recurrent neural networks, long short-term memory...
```

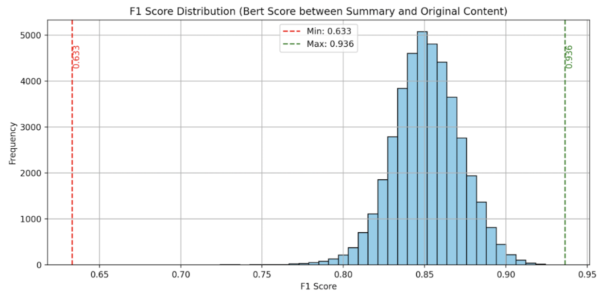


Figure 6.1: BERTScore over All Summarized Chunks


```
                    }
                ]
        }
]
```


The full dataset, including metadata, raw text chunks, and summaries, is available for download at: https://drive.google.com/file/d/1ZOOySvXF4yVDVrB6C6Rj5GbNcxNOeKKV/view?usp=sharing

6 Evaluation and Results

6.1 Data Preprocessing Evaluation

To address the quality of LLM-generated summaries during preprocessing, we computed F1 of BERTScore [4] between each generated summary and its corresponding original content. The histogram below shows the distribution of F1 scores of BERTScore across all summarized chunks.

The majority of scores lie between 0.8 and 0.9, indicating that most summaries maintain a very similar semantic meaning with the original content. The minimum F1 score observed was 0.633, suggesting a few challenging or noisy chunks with summaries, but only a very limited number, while the maximum reached 0.936, representing near-perfect retention of original meaning.

6.2 RAG System Output Result

In the comparison, we asked the same question "Tell me what is KeepKV with references" to different LLMs. Our system, powered by our RAG system, consistently delivers accurate and concrete answers by grounding its responses in verified source documents. In contrast, when the same question is posed to models like LLaMA 3 without RAG integration or GPT-3.5, the outputs often include hallucinated information, fabricated references, and non-existent URLs. This disparity is clearly demonstrated in the RAG System Output Result section of our paper, where our RAG-enhanced model provides verifiable and source-backed responses, while the baseline models frequently generate misleading or entirely false citations.

7 Conclusion

In summary, our system presents an effective and modular approach to Retrieval-Augmented Generation for academic literature, focused on the computer science and machine learning domains. By combining structured document parsing, semantic chunking, vector-based retrieval,

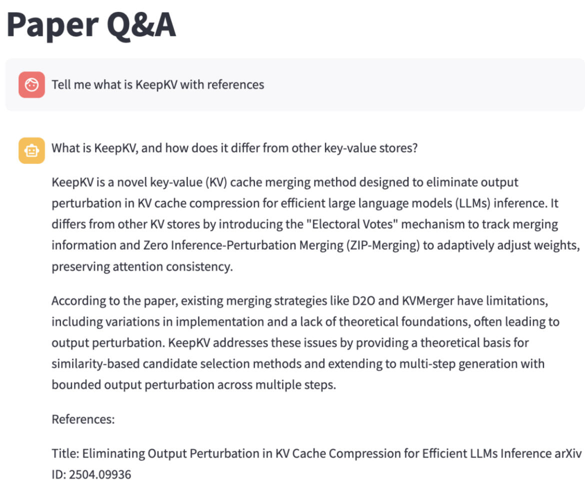


Figure 6.2: Result from Paper Q&A

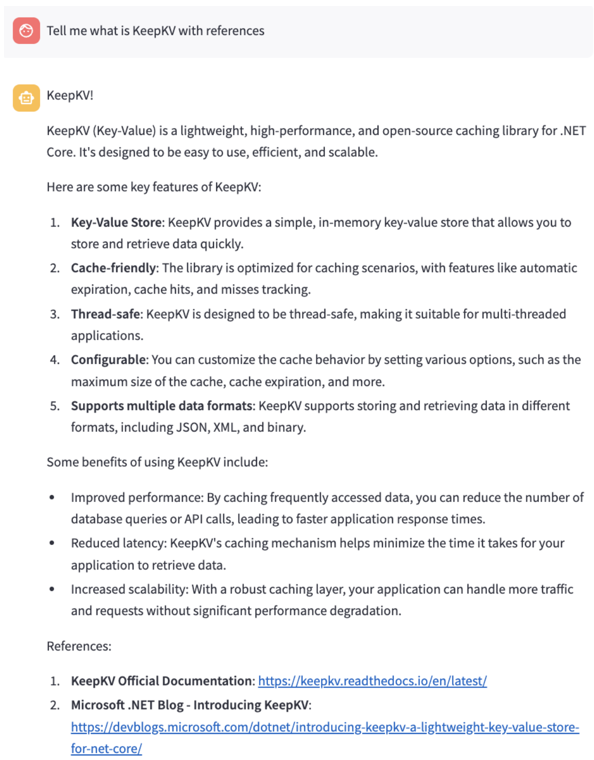
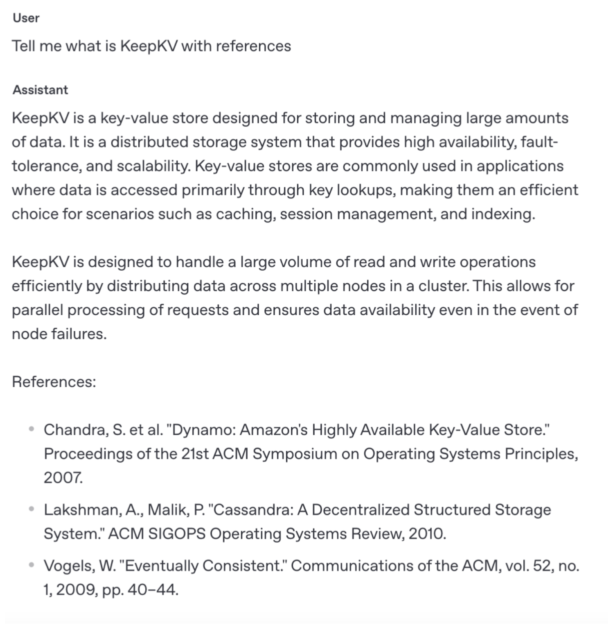


Figure 6.3: Result from LLaMA 3 without RAG



User

Tell me what is KeepKV with references

Assistant

KeepKV is a key-value store designed for storing and managing large amounts of data. It is a distributed storage system that provides high availability, fault-tolerance, and scalability. Key-value stores are commonly used in applications where data is accessed primarily through key lookups, making them an efficient choice for scenarios such as caching, session management, and indexing.

KeepKV is designed to handle a large volume of read and write operations efficiently by distributing data across multiple nodes in a cluster. This allows for parallel processing of requests and ensures data availability even in the event of node failures.

References:

- Chandra, S. et al. "Dynamo: Amazon's Highly Available Key-Value Store." Proceedings of the 21st ACM Symposium on Operating Systems Principles, 2007.
- Lakshman, A., Malik, P. "Cassandra: A Decentralized Structured Storage System." ACM SIGOPS Operating Systems Review, 2010.
- Vogels, W. "Eventually Consistent." Communications of the ACM, vol. 52, no. 1, 2009, pp. 40–44.

Figure 6.4: Result from GPT 3.5

and prompt-driven LLM responses, we offer users a powerful tool for navigating dense research papers through natural language queries. This solution improves access to technical content, reduces the effort required for literature review, and provides a foundation for future expansion into other academic fields and languages.

8 References

[1] Lewis, Patrick, et al. "Retrieval-augmented generation for knowledge-intensive nlp tasks." Advances in neural information processing systems 33 (2020): 9459-9474.

[2] Lála, Jakub, et al. "Paperqa: Retrieval-augmented generative agent for scientific research." arXiv preprint arXiv:2312.07559 (2023).

[3] P. Lopez, "GROBID: Combining Automatic Bibliographic Data Recognition and Term Extraction for Scholarship Publications," Lecture Notes in Computer Science, pp. 473–474, Jan. 2009, doi: https://doi.org/10.1007/978-3-642-04346-8_62.

[4] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "BERTScore: Evaluating Text Generation with BERT," arXiv:1904.09675 [cs], Feb. 2020, Available: https://arxiv.org/abs/1904.09675.

[5] Jiang, Ting, et al. "Promptbert: Improving bert sentence embeddings with prompts." arXiv preprint arXiv:2201.04337 (2022).