**A Retrieval-Augmented Question Answering System Using BERT for Stream-Specific Educational Content**

Submitted in partial fulfilment of the   
requirements of the Degree: M.Tech in Data science and engineering

By

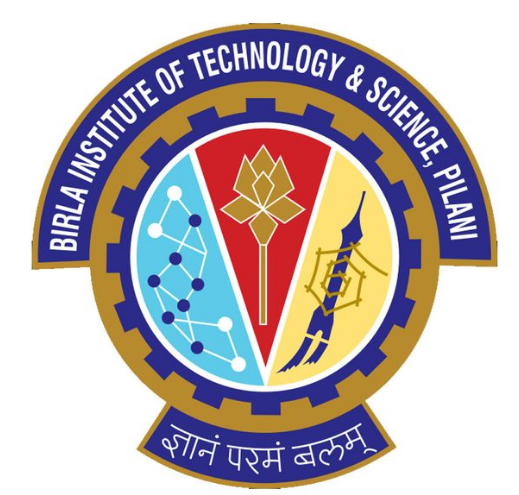
**Jaykumar Chaudhary**

**2022DC04341**

Under the supervision of

**Mr. Lalkar Eknath Chhadawelkar**

**Technical Evangelist (Cybage Software, Pune)**



BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

Pilani (Rajasthan) INDIA

JULY, 2025

**Acknowledgement**

I would like to express my heartfelt gratitude to all those who have supported and guided me throughout the journey of this dissertation.

First and foremost, I am deeply thankful to my supervisor, **Mr. Lalkar Eknath Chhadawelkar**, for his invaluable guidance, continuous encouragement, and patient supervision at every stage of this project. His insightful feedback and expertise have been crucial in shaping my work and motivating me to achieve my best.

I would also like to sincerely thank **Mrs. Radhika Raghuvanshi** for her initial guidance and for helping me refine the core idea of my dissertation. Her suggestions on the overall roadmap and her clarity of thought gave direction to my efforts from the very beginning.

My special thanks to **Jayalakshmi Natarajan ma’am**, my BITS evaluator, for her thoughtful evaluation and constructive comments, which have helped me strengthen both my research and its presentation.

Finally, I owe my deepest appreciation to my family for their unwavering support, constant encouragement, and understanding. Their faith in me has been my greatest source of motivation throughout this academic journey.

To all those who have helped me, directly or indirectly, in completing this work, I extend my sincere thanks.

**BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI**

**SECOND SEMESTER 2024-25**

**DSECLZG628T / AIMLCZG628T DISSERTATION**

**CERTIFICATE**

This is to certify that the Dissertation entitled A Retrieval-Augmented Question Answering System Using BERT for Stream-Specific Educational Content

and submitted by Mr. Jaykumar Chaudhary ID No. 2022DC04341 in partial fulfilment of the requirements of DSECLZG628T Dissertation, embodies the work done by him/her under my supervision.

**BITS ID No. 2022DC04341 Name of Student: Jaykumar Chaudhary**

**Name of Supervisor: Mr. Lalkar Eknath Chhadawelkar**

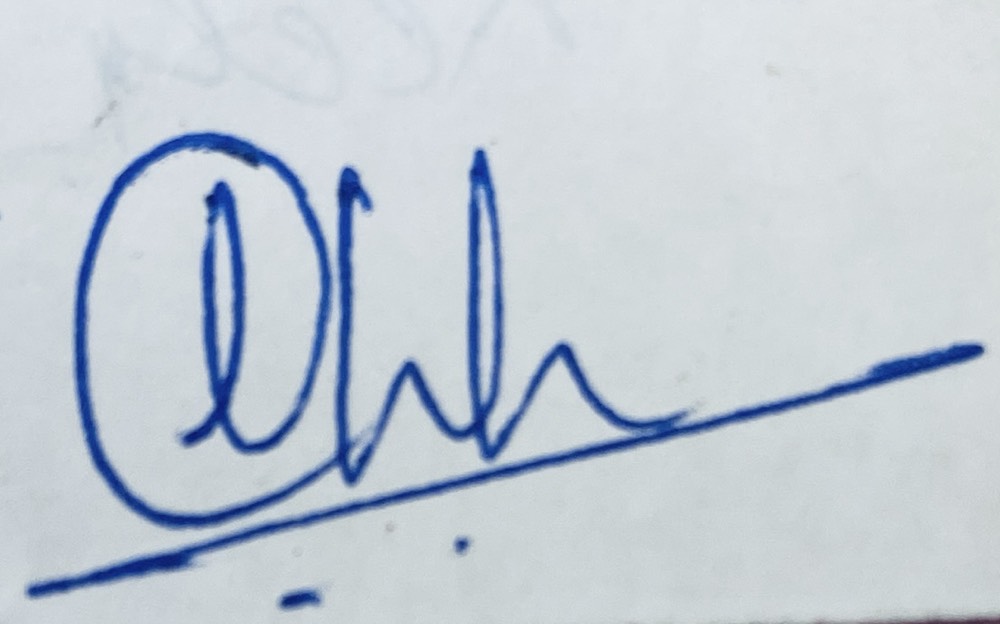
**Designation of Supervisor: Technical Evangelist**

**Qualification and Experience: M.Tech in Data Science and Engineering, 28 Years**

**Official E- mail ID of Supervisor:** [**lalkar@cybage.com**](mailto:lalkar@cybage.com)

**Topic of Dissertation: A Retrieval-Augmented Question Answering System Using BERT for Stream-Specific Educational Content**

A close-up of a text

AI-generated content may be incorrect. 

Signature of Student Signature of Supervisor  
  
Date: 4-JULY-2025 Date: 4-JULY-2025

# Abstract

In academic environments, students often struggle to quickly locate specific information across large, text-heavy educational documents such as course handouts. As elective subjects vary across semesters, navigating this material can be overwhelming, especially when trying to understand topics or explore subjects before making elective decisions. Traditional keyword-based search methods lack contextual understanding, leading to imprecise and inefficient information retrieval.

This dissertation proposes the design and development of a **Retrieval-Augmented Question Answering (QA) system** that leverages a **pretrained BERT-based model** to enable students to query academic content in natural language and receive accurate, contextually relevant answers. The system is primarily designed to process structured and semi-structured educational content — including course handouts — from the **Data Science** and **AI/ML streams**.

The proposed solution follows a **Retrieval-Augmented Generation (RAG) architecture** that separates the QA process into two components: **retrieval** and **reading**. First, a **retriever** uses **Sentence-BERT** to generate **embeddings** and identify relevant content, which is then indexed using **FAISS** for efficient **semantic search**. Next, a **reader model (BERT fine-tuned on SQuAD)** extracts the most likely answer span from the retrieved text.

Unlike conventional QA systems that require domain-specific fine-tuning or extensive labeled data, this system uses pretrained components and unsupervised document chunking. This makes it scalable, adaptable, and ideal for academic use. The system is designed to help students retrieve reliable answers to subject-related queries, assist in exam preparation, and support elective planning by giving clarity on subject depth and focus.

The expected result is a functional prototype capable of answering factual, definition-based, and conceptual questions using curriculum-aligned educational content. The system will be evaluated through QA metrics like **Exact Match** and **F1 Score**, along with qualitative feedback from users. This dissertation also discusses limitations such as the lack of support for **multi-hop reasoning** or generative answers. Nonetheless, it showcases a practical application of **NLP** in the academic domain, offering an intelligent way to support self-directed learning.

# Key Words

BERT, Question Answering, Retrieval-Augmented Generation, Semantic Search, Educational NLP, Sentence-BERT, FAISS, Extractive QA

**List of Symbols & Abbreviations**

|  |  |
| --- | --- |
| Symbol / Abbreviation | Description |
| NLP | Natural Language Processing |
| QA | Question Answering |
| RAG | Retrieval-Augmented Generation |
| LLM | Large Language Model |
| BERT | Bidirectional Encoder Representations from Transformers |
| SQuAD | Stanford Question Answering Dataset |
| FAISS | Facebook AI Similarity Search |
| ChromaDB | Chroma Vector Database |
| API | Application Programming Interface |
| PDF | Portable Document Format |
| DOCX | Microsoft Word Open XML Document Format |
| UI | User Interface |
| ID | Identification Number |

**List of Tables**

[*Table 1: Chunks After Preprocessing and Hybrid Chunking* 12](#_Toc202474155)

[*Table 2: Embedding Vectors for Text Chunks* 13](#_Toc202474156)

[*Table 3: FAISS Retrieval Results for Sample Queries* 14](#_Toc202474157)

[*Table 4: Example of Answer Sources and Types in Edge Cases* 18](#_Toc202474158)

[*Table 5: comparison of the baseline and modern RAG pipeline architectures* 22](#_Toc202474159)

[*Table 6: Example of Semantic Chunk Metadata* 23](#_Toc202474160)

[*Table 7: Example Embeddings for Semantic Chunks* 24](#_Toc202474161)

[*Table 8: Example Entries in ChromaDB Vector Store* 25](#_Toc202474162)

[*Table 9: Sample LLM answers after polishing.* 27](#_Toc202474163)

**List of Figures**

[*Figure 1: RAG Question Answering Pipeline* 7](#_Toc202474111)

[*Figure 2: Preprocessing and Hybrid Chunking Workflow* 11](#_Toc202474112)

[*Figure 3: all-MiniLM-L6-v2 Model Architecture* 12](#_Toc202474113)

[*Figure 4: Implementation Flow of Embedding and Retrieval Pipeline* 15](#_Toc202474114)

[*Figure 5: BERT-Large SQuAD QA Pipeline Architecture* 16](#_Toc202474115)

[*Figure 6: Decision Flow for Reader Module Edge Cases* 18](#_Toc202474116)

[*Figure 7: Overview of the modern modular RAG pipeline architecture* 22](#_Toc202474117)

[*Figure 8: Semantic Chunking Workflow* 23](#_Toc202474118)

[*Figure 9: Embedding Workflow for Semantic Chunks* 24](#_Toc202474119)

[*Figure 10: Document and query embeddings into ChromaDB* 25](#_Toc202474120)

[*Figure 11: Construction of a retriever object from the ChromaDB vector index* 26](#_Toc202474121)

[*Figure 12: LLM-Based Answer Generation and Polishing Workflow* 27](#_Toc202474122)

[*Figure 13: Modular RAG pipeline implementation* 28](#_Toc202474123)

[*Figure 14: User Interface Snapshot* 29](#_Toc202474124)

Table of Contents

[Abstract 1](#_Toc202541385)

[Key Words 1](#_Toc202541386)

[Chapter 1 Introduction 6](#_Toc202541387)

[1.1 The Challenge of Academic Information Access 6](#_Toc202541388)

[1.2 Problem Statement 6](#_Toc202541389)

[1.3 Motivation and Project Evolution 6](#_Toc202541390)

[1.4 System Overview 7](#_Toc202541391)

[1.5 Objectives of the Dissertation 7](#_Toc202541392)

[Chapter 2 Literature Survey 9](#_Toc202541393)

[2.1 Introduction 9](#_Toc202541394)

[2.2 Classic RAG and BERT-Based QA 9](#_Toc202541395)

[2.3 Advances in Chunking and Embedding 9](#_Toc202541396)

[2.4 Modern Modular RAG with LLMs 10](#_Toc202541397)

[2.5 Summary 10](#_Toc202541398)

[Chapter 3 Baseline RAG System (Traditional Approach) 11](#_Toc202541399)

[3.1 Introduction 11](#_Toc202541400)

[3.2 Document Chunking & Pre-processing 11](#_Toc202541401)

[3.2 Embedding & Retrieval 12](#_Toc202541402)

[3.2.1 Semantic Embedding with all-MiniLM-L6-v2 12](#_Toc202541403)

[3.2.3 Implementation Workflow: Embedding and Retrieval 14](#_Toc202541404)

[3.3 Reader Module 15](#_Toc202541405)

[3.3.1 Answer Extraction with BERT QA Model 16](#_Toc202541406)

[3.3.2 Handling Edge Cases: Keyword and Fall-back Heuristics 17](#_Toc202541407)

[3.4 Observations & Limitations 19](#_Toc202541408)

[Chapter 4 Modern RAG System with LLMs & Ecosystem Tools 20](#_Toc202541409)

[4.1 Motivation for Advancing the Pipeline 20](#_Toc202541410)

[4.2 New Architecture Overview 21](#_Toc202541411)

[4.2.1 System Design and Data Flow 21](#_Toc202541412)

[4.3 Enhanced Retriever Module 22](#_Toc202541413)

[4.3.1 Semantic Chunking of Academic Documents 22](#_Toc202541414)

[4.3.2 Advanced Embedding with FastEmbed (BAAI/bge-base-en-v1.5) 23](#_Toc202541415)

[4.3.3 Persistent Vector Storage with ChromaDB 24](#_Toc202541416)

[4.3.4 Retriever Object Creation from the Vector Store 25](#_Toc202541417)

[4.4 Advanced Reader/LLM Module 26](#_Toc202541418)

[4.4.1 LLM-Based Answer Generation Workflow 27](#_Toc202541419)

[4.5 Implementation Pipeline: End-to-End Workflow and Code Structure 27](#_Toc202541420)

[Chapter 5 Progress and Future Directions 30](#_Toc202541421)

[5.1 Work Completed till Mid-Semester 30](#_Toc202541422)

[5.2 Discussion and Path Forward 30](#_Toc202541423)

[Chapter 6 Bibliography / References 32](#_Toc202541424)

# Chapter 1 Introduction

## 1.1 The Challenge of Academic Information Access

In the digital age, students and educators have unprecedented access to academic resources—PDF handouts, lecture notes, assignments, and supplementary readings—often spanning hundreds or even thousands of pages. Paradoxically, this abundance can make it more difficult to pinpoint precise, contextually accurate answers to academic questions. Traditional keyword-based search tools fall short in this environment: they fail to capture semantic relationships, often misinterpret technical terminology, and cannot reliably handle mathematical or tabular content.

Recent advances in Natural Language Processing (NLP) offer transformative possibilities. **Retrieval-Augmented Generation (RAG)** has emerged as a powerful paradigm that combines semantic search with large language models (LLMs) to produce direct, context-rich responses. By retrieving relevant passages from large collections and then generating an answer grounded in those passages, RAG systems move beyond keyword matching to deliver explanations in natural, domain-appropriate language.

However, applying RAG effectively to the educational domain introduces new challenges. Academic documents are diverse in format, often containing complex structures such as multi-column layouts, equations, and embedded tables. Privacy and offline access are also critical for many institutions, creating demand for local inference options. To meet these needs, this dissertation advances a robust, modular RAG system capable of:

* Parsing and indexing multi-format academic documents (**PDF, DOCX**) with structure preservation.
* Handling complex layouts through tools like **LlamaParse**.
* Supporting both **cloud LLMs** (**Gemini, Groq**) and **local LLMs** via **Ollama** for privacy-preserving deployment.
* Integrating user feedback to iteratively improve retrieval accuracy and answer quality.

## 1.2 Problem Statement

While academic resources are more accessible than ever, extracting precise, well-contextualized answers from them remains a challenge. Existing search and Q/A tools often exhibit three key limitations:

1. **Semantic Gap** – Lack of deep understanding of academic language and structure, resulting in irrelevant or incomplete answers.
2. **Format and Structure Blindness** – Difficulty processing non-linear layouts, formulas, and tables in PDFs or Word documents.
3. **Limited Trust and Adaptability** – Minimal transparency in how answers are derived, and little capacity to adapt to user feedback.

The goal of this work is to design and implement an intelligent, context-aware question answering system that addresses these issues by combining advanced retrieval methods, modular LLM orchestration, and robust document parsing—delivering answers that are both accurate and traceable to their sources.

## 1.3 Motivation and Project Evolution

This project began as a classic, BERT-based QA pipeline employing semantic chunking, Sentence-BERT embeddings, and extractive answer selection. This early system proved that context-aware retrieval in the educational domain is feasible but also revealed opportunities for substantial improvement.

The second phase, which forms the focus of this dissertation, builds on that foundation by integrating:

* **LangChain** for modular orchestration of retrieval and generation steps.
* **ChromaDB** for scalable, persistent vector storage.
* **Multiple LLM backends**:
  + **Gemini** for high-quality, context-rich responses.
  + **Groq** for ultra-low latency inference.
  + **Ollama** for running open-weight LLMs locally, ensuring privacy and offline capability.
* **LlamaParse** for accurate parsing of complex PDFs, including multi-column layouts, formulas, and tables.
* **Multi-format ingestion** to support both PDF and DOCX academic documents.
* **User feedback integration** for adaptive retrieval improvement.
* **Document summarization** and **prompt-driven Q/A generation** to enable topic-specific learning materials.
* **RAGAS evaluation** to systematically measure answer faithfulness, context recall, and relevance.

This evolution represents a deliberate and strategic enhancement—not a pivot due to limitations—ensuring that the final system is flexible, scalable, and tailored to real academic needs.

## 1.4 System Overview

The following diagrams illustrate both the baseline and advanced architectures implemented in this project. This project implements a modern, modular Retrieval-Augmented Generation (**RAG**) application for educational question answering. The RAG methodology blends efficient document retrieval with advanced language generation, enabling the system to deliver precise, context-aware answers tailored to user queries.

The **high-level workflow** of the system is outlined in Figure 1 below:

A close-up of several colorful rectangular labels

AI-generated content may be incorrect.

*Figure 1: RAG Question Answering Pipeline*

This modular architecture enables each pipeline component to be independently developed, tuned, or upgraded. Academic documents in various formats are ingested, chunked, and embedded using advanced models, with vectors stored in ChromaDB for fast semantic retrieval. LangChain orchestrates retrieval workflows, supporting context-aware, multi-hop queries. Retrieved passages and user questions are processed by state-of-the-art LLM readers (Gemini, Groq), which generate explanatory or synthesized answers. The design also anticipates integration of local LLMs via Ollama, ensuring privacy and control for institution-specific deployments. This foundation positions the system for rapid adaptation as new models and retrieval tools emerge.

This streamlined RAG workflow underpins the flexible and robust academic QA capabilities discussed in greater depth throughout Chapters 3 and 4.

## 1.5 Objectives of the Dissertation

The overarching aim of this dissertation is to bridge the gap between static academic documents and dynamic, question-driven learning. This work focuses on designing, implementing, and evaluating a modular Retrieval-Augmented Generation (RAG) system tailored for the educational domain. The specific objectives are as follows:

**1.** **Develop a scalable, semantic QA system for multi-format academic documents**

Build a platform that can ingest and process PDF and DOCX files while preserving their structure, including multi-column layouts, tables, and formulas, ensuring accurate downstream retrieval and answering.

**2.** **Implement modular LLM orchestration for cloud and local inference**

Integrate Gemini, Groq, and Ollama as interchangeable backends, enabling deployment scenarios that balance performance, cost, and privacy needs.

**3.** **Design advanced retrieval strategies for high-fidelity context selection**

Use semantic embeddings with context expansion and metadata filtering to ensure relevant, complete, and traceable evidence is retrieved for each question.

**4.** **Ensure trustworthiness and transparency in generated answers**

Ground every answer in retrieved context, display provenance metadata, and use linguistic polishing without altering factual accuracy.

**5.** **Enable summarization and prompt-driven Q/A generation**

Support concise content summarization and use it to generate focused questions and answers for study or assessment purposes.

**6.** **Integrate user feedback for continuous improvement**

Capture binary and comment-based feedback to inform retrieval re-ranking and model tuning over time.

**7.** **Evaluate the system using RAGAS and comparative benchmarking**

Measure performance across multiple metrics—faithfulness, context recall, and answer relevance—while benchmarking LLM backends for speed, accuracy, and usability.

# Chapter 2 Literature Survey

## 2.1 Introduction

The landscape of question answering (QA) systems has evolved rapidly with advancements in natural language processing and deep learning. Traditional information retrieval systems—relying on keyword matching, bag-of-words models, or statistical ranking—could rarely capture the rich context and deep semantics inherent in complex academic materials. As educational resources have grown in scale and complexity, students increasingly need tools that can understand context, reason over information, and generate precise, trustworthy answers. Retrieval-Augmented Generation (RAG) addresses this need by combining the power of modern information retrieval with the generation capabilities of large language models (LLMs). This dual approach enables systems to first identify the most relevant parts of vast document collections and then generate high-quality, contextually aware answers that are both accurate and explainable—qualities essential for educational applications where trust and traceability matter (Liu et al., 2023; Bo Ni et al., 2025; Shah et al., 2024).

## 2.2 Classic RAG and BERT-Based QA

Classic RAG pipelines represent a significant leap forward from earlier QA architectures. At their core, these systems operate in two tightly integrated stages: retrieval and reading. In the retrieval stage, a user’s question is converted into a dense vector representation, often using models such as Sentence-BERT. This vector is then used to search a vector database—commonly implemented with scalable tools like FAISS—to identify and rank the most semantically relevant document chunks (Liu et al., 2023; Wenqi Fan et al., 2024). The use of dense retrieval allows the system to capture subtle relationships, such as synonyms or rephrased concepts, that keyword search would miss.

Once the top-ranked chunks are retrieved, the reader stage utilizes a powerful pre-trained or fine-tuned BERT model to extract the specific answer span. This model can process entire passages, weighing both the immediate question and the broader context, to find the most likely answer. This two-step process—first narrowing the search space and then drilling down to the answer—delivers both efficiency and accuracy, making it highly effective for academic QA where precision is critical and answers are often embedded in dense, technical text (Wenqi Fan et al., 2024; Liu et al., 2023).

## 2.3 Advances in Chunking and Embedding

Chunking and embedding are foundational steps in any RAG system, but their importance is magnified in the educational domain. Academic materials are typically structured around complex, multi-part explanations, hierarchical headings, bulleted lists, and embedded formulas. Basic approaches that divide text by arbitrary length or page breaks risk fragmenting crucial context, leading to poor retrieval performance. Recent research and practical experiments advocate for **semantic chunking**: grouping together related explanations, definitions, and associated mathematical content so that each chunk represents a coherent knowledge unit (Franklin Lee & Tengfei Ma, 2025; Shah et al., 2024).

These carefully constructed chunks are then embedded into a high-dimensional vector space using advanced models like FastEmbed or domain-adapted Sentence-BERT. Embeddings capture both local and global context, enabling the system to match queries with the most appropriate content, even if it is phrased differently or spread across multiple sections. Effective chunking and embedding ensure that answers are not only relevant but also pedagogically sound—providing students with both factual content and the necessary context for deeper understanding (M. Shah et al., 2024; Bo Ni et al., 2025).

## 2.4 Modern Modular RAG with LLMs

The newest generation of RAG systems capitalizes on the capabilities of large language models and the flexibility of modular orchestration frameworks. Unlike earlier extractive QA pipelines, these architectures can perform complex reasoning and synthesis—combining information from multiple sources and presenting it in clear, structured, and natural language. Modular tools such as **LangChain** allow developers to seamlessly connect document loaders, semantic chunkers, dense retrievers, and a variety of LLMs (including Gemini, Groq, and locally hosted Ollama) within a single pipeline (Sonal Prabhune & Donald J. Berndt, 2024; Petko Georgiev et al., 2024; Rohan Anil et al., 2023).

Scalable vector databases like **ChromaDB** ensure that even large, multi-document educational datasets can be efficiently indexed and queried. LLMs are not limited to extracting text—they can summarize, paraphrase, and even explain answers, which is particularly valuable in education where explanations often matter as much as facts. The modularity of these systems supports easy experimentation, integration of new models, and adaptation to different academic subjects or user requirements. Studies show that such modular RAG systems outperform traditional pipelines in both answer quality and user engagement, while also making it easier to maintain and upgrade the QA system over time (Sonal Prabhune & Donald J. Berndt, 2024; Franklin Lee & Tengfei Ma, 2025).

## 2.5 Summary

Overall, the evolution from classic BERT-based QA to advanced, LLM-driven RAG systems enables much richer, more flexible question answering in academic settings. Our project follows this trajectory, combining the best practices identified in recent literature with innovations in modularity and retrieval (Liu et al., 2023; Shah et al., 2024).

# Chapter 3 Baseline RAG System (Traditional Approach)

## 3.1 Introduction

This chapter presents the baseline question answering pipeline developed as the foundation of this project. This approach leverages established retrieval-augmented generation (RAG) techniques, focusing on traditional methods for preprocessing, chunking, semantic retrieval, and answer extraction. By systematically transforming raw academic materials into structured, searchable units and employing proven models for both retrieval and reading, the baseline system establishes the core workflow upon which later advancements are built. The results and observations from this pipeline not only demonstrate the feasibility of retrieval-based QA in educational contexts but also highlight key limitations that motivate the development of a more advanced, LLM-driven solution in subsequent chapters.

## 3.2 Document Chunking & Pre-processing

A robust question answering system must first convert raw academic materials—whether in PDF or DOCX format—into structured, meaningful segments for downstream analysis. This process, known as pre-processing and chunking, is crucial for ensuring the system can accurately locate and extract relevant answers.

A black background with white squares

AI-generated content may be incorrect.

*Figure 2: Preprocessing and Hybrid Chunking Workflow*

This diagram illustrates the end-to-end workflow from input document (PDF or DOCX) through paragraph extraction and intelligent chunking, resulting in a collection of context-rich text segments ready for retrieval.

In this workflow, input documents are first loaded using appropriate tools (such as python-docx for DOCX files or a PDF parsing library for scanned materials). The system extracts non-empty paragraphs to ensure that only content-bearing text is processed. Each paragraph is then evaluated: unusually long paragraphs are split into sentences and regrouped using a hybrid strategy, while shorter paragraphs are merged to achieve optimal chunk size. This approach leverages both the document’s inherent structure and linguistic boundaries, resulting in chunks that balance context with focus.

The importance of this workflow lies in its direct impact on retrieval effectiveness. If chunks are too large, they may contain irrelevant or diluted information; if too small, they can lose essential context. By dynamically adjusting chunk size and boundaries, the system produces segments that are well-matched to the variety of queries posed by students.

One clear advantage of supporting both PDF and DOCX formats is the flexibility it offers in academic settings, where course materials can appear in a range of digital forms. The use of proven NLP libraries such as NLTK for sentence tokenization further enhances the quality of chunking, ensuring linguistic coherence within each segment.

A practical outcome of this preprocessing and chunking is shown below:

|  |  |  |
| --- | --- | --- |
| Chunk # | Word Count | Chunk Preview |
| 1 | 96 | Entropy is a measure of uncertainty in machine learning… |
| 2 | 109 | Regularization helps to prevent overfitting by introducing a penalty term… |
| 3 | 80 | The bias-variance tradeoff describes how model complexity affects prediction error… |

*Table 1: Chunks After Preprocessing and Hybrid Chunking*

Through this approach, the system generates a curated list of searchable, informative text segments—laying a strong foundation for subsequent semantic embedding, retrieval, and answer extraction. This thoughtful chunking is essential for both the efficiency and accuracy of the overall QA pipeline, directly influencing the quality of answers provided to students.

## 3.2 Embedding & Retrieval

Accurate retrieval in question answering systems hinges on two fundamental processes: the conversion of text into meaningful vector representations, and the rapid search of these vectors to find relevant content. This section explains both aspects as implemented in the baseline pipeline.

### 3.2.1 Semantic Embedding with all-MiniLM-L6-v2

At the core of semantic search lies the **all-MiniLM-L6-v2 model**—a compact yet powerful transformer architecture. This model consists of six transformer encoder layers and roughly 22.7 million parameters. It is specifically optimized to generate fixed-length, 384-dimensional vector embeddings for sentences and short paragraphs, efficiently capturing their semantic content.



*Figure 3: all-MiniLM-L6-v2 Model Architecture*

The model is widely recognized for its ability to encode complex meaning into a small vector, making it ideal for large-scale semantic search. The resulting embeddings are robust to variations in wording and can be compared using simple distance metrics.

When a text chunk is processed by all-MiniLM-L6-v2, the output is a dense vector that can be directly compared with embeddings of other chunks or queries. The first few components of such vectors for sample chunks are shown in Table 2:

|  |  |  |
| --- | --- | --- |
| Chunk # | Chunk Preview | Embedding (First 8 Dims) |
| 1 | “Entropy is a measure…” | |  | | --- | |  |  |  | | --- | | [0.022, -0.108, 0.176, …] | |
| 2 | “Regularization helps…” | |  | | --- | |  |  |  | | --- | | [0.073, 0.029, 0.109, …] | |

*Table 2: Embedding Vectors for Text Chunks*

**3.2.2 Efficient Retrieval with FAISS**

A central challenge in large-scale question answering is finding the most relevant text chunks for any given query—quickly and accurately—among thousands of candidates. This is where **FAISS (Facebook AI Similarity Search)** becomes essential. Developed by Meta AI, FAISS is an open-source library purpose-built for fast similarity search and clustering of dense vectors, making it especially suitable for natural language processing and semantic search applications.

FAISS excels at handling high-dimensional embeddings, such as those produced by neural language models, and can scale to collections containing millions of vectors. It offers a range of indexing strategies, from brute-force (exact) search to highly efficient approximate methods, enabling the system designer to balance accuracy and latency as needed. For academic QA, FAISS is typically used with normalized embeddings and inner product (cosine similarity) search, ensuring that semantically similar text chunks are identified regardless of their original scale or distribution.

Several factors make FAISS the preferred choice for this system:

* **Speed and Scalability:** FAISS supports real-time search, delivering sub-second responses even as the academic corpus expands.
* **Flexibility:** It provides multiple search algorithms suitable for both small and very large datasets.
* **Ease of Integration:** FAISS is compatible with major machine learning environments and is easy to deploy in Python-based pipelines.
* **Reliability:** As a widely adopted open-source project, FAISS is robust and production-ready.

In this pipeline, all chunk embeddings are added to a FAISS index configured for inner product search. When a user question arrives, it is embedded using the same model and submitted to FAISS, which instantly returns the top-k most semantically similar chunks from the entire collection.

The effectiveness of FAISS retrieval is illustrated by the sample queries and their best-matched chunks in Table 3:

|  |  |  |
| --- | --- | --- |
| Query | Top Chunk (Preview) | Similarity Score |
| What is entropy in ML? | Entropy is a measure of uncertainty… | 0.87 |
| How do we prevent overfitting? | Regularization helps prevent… | 0.83 |

*Table 3: FAISS Retrieval Results for Sample Queries*

By leveraging FAISS, the retrieval module ensures that user queries are rapidly and accurately matched with the most relevant academic content, providing a robust foundation for high-quality answer extraction downstream.

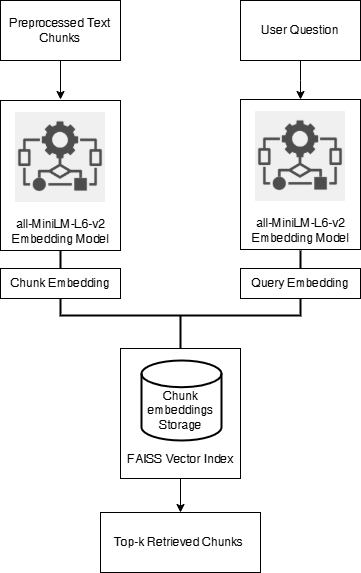
### 3.2.3 Implementation Workflow: Embedding and Retrieval

The embedding and retrieval components of the baseline question answering system are designed to operate in a streamlined, reproducible workflow. This architecture ensures that academic materials are consistently processed and indexed, while queries are efficiently matched to relevant content—regardless of scale or document format.

At the implementation level, the workflow begins with the collection of pre-processed text chunks obtained from various academic documents. Each chunk is systematically fed into the all-MiniLM-L6-v2 embedding model, resulting in a dense vector representation that encapsulates its semantic meaning. These vectors are normalized and added to a FAISS index, forming the searchable database.

When a user submits a question, it undergoes the same embedding process, producing a query vector in the same high-dimensional space. This query embedding is then used to search the FAISS index, which rapidly returns the top-k most semantically similar document chunks. These retrieved chunks are passed to the downstream answer extraction stage.

This modular pipeline not only ensures high performance and scalability, but also supports easy extension and debugging—each stage is logically separated and can be independently evaluated.



*Figure 4: Implementation Flow of Embedding and Retrieval Pipeline*

This diagram depicts the system’s operational steps, from input documents through embedding, indexing, and retrieval, with clear indication of data flow between each stage.

## 3.3 Reader Module

The final step in the baseline pipeline is answer extraction—identifying and presenting the most relevant, concise response to a user’s question. This is accomplished through a two-stage approach: first, a state-of-the-art BERT-based reader is applied to the top retrieved text chunks, and second, the system employs smart keyword-based heuristics and fall-back strategies to maximize answer reliability, even when the direct extractive model is uncertain.

### 3.3.1 Answer Extraction with BERT QA Model

The reader module in this system is built upon the **bert-large-uncased-whole-word-masking-finetuned-squad** model, one of the most robust and widely adopted architectures for extractive question answering. BERT (Bidirectional Encoder Representations from Transformers) is a deep transformer-based language model designed to understand context in natural language by processing words in both directions—left-to-right and right-to-left—within a sentence.

The **"bert-large-uncased-whole-word-masking-finetuned-squad"** variant is a large-scale model featuring 24 transformer encoder layers and over 340 million parameters. It is pre-trained on a vast English corpus and further fine-tuned on the SQuAD (Stanford Question Answering Dataset), specializing it for reading comprehension and QA tasks. The “whole word masking” technique during training enables BERT to develop a more nuanced sense of word boundaries and contextual relationships, improving answer extraction accuracy.

In this pipeline, when a user submits a question, it is paired with the top retrieved chunk from the document collection. The BERT QA model receives both as input and processes them through its deep stack of transformer layers, leveraging multi-head self-attention mechanisms to model complex contextual dependencies. The model then outputs the most probable start and end positions for the answer span within the context, along with a confidence score for its prediction.

A screen shot of a cell phone

AI-generated content may be incorrect.

*Figure 5: BERT-Large SQuAD QA Pipeline Architecture*

The pipeline’s workflow can be summarized as:

* **Input:** User question and top-ranked context chunk.
* **Model:** “bert-large-uncased-whole-word-masking-finetuned-squad” processes through multiple self-attention layers.
* **Output:** Predicted answer span within the context, with an associated confidence score.

This approach ensures that the system can accurately extract answers directly from text, particularly when the answer is explicitly stated in the context and closely matches the phrasing of the question. The result is a high-precision, explainable answer that can be traced to a specific document segment.

### 3.3.2 Handling Edge Cases: Keyword and Fall-back Heuristics

While transformer-based models like BERT excel at extracting answer spans when the context is explicit and well-matched, real-world academic handouts can present cases where answers are implicit, phrased differently, or distributed across multiple sentences. To ensure robustness, the pipeline incorporates a two-stage backup mechanism: **global keyword matching** and a final **fall-back heuristic**.

In the first backup stage, the system employs a keyword overlap strategy. When the BERT QA model’s output does not meet minimum standards for answer length or confidence, the system scans the Top-K retrieved chunks for sentences sharing key non-stop word terms with the user’s question. By tallying the intersection of content words between the question and each candidate sentence, the method surfaces sentences that, while perhaps not extracted by the neural model, are likely to contain directly relevant information.

If this keyword matching still fails to yield a suitable answer, the pipeline defaults to a straightforward fall-back: presenting the first sentence of the top-ranked chunk. This ensures the user always receives a response grounded in the most relevant context available, even if the answer is only partially matched.

This layered approach—moving from neural extraction, to keyword scoring, to context-based fall-back—ensures answer delivery is both robust and explainable. It also supports transparency for users and evaluators, as each answer can be traced to the logic or source that produced it.

A screenshot of a black background

AI-generated content may be incorrect.

*Figure 6: Decision Flow for Reader Module Edge Cases*

To illustrate the decision process and variety of answer sources, Table 4 shows example queries with the answer produced, the method used, and any confidence or keyword overlap scores as available.

|  |  |  |  |
| --- | --- | --- | --- |
| Query | Answer | Method Used | Score/Type |
| What is overfitting? | “Overfitting occurs when a model…” | BERT QA | 0.93 |
| What are optimization methods? | “Gradient descent is a commonly…” | Keyword Match | Overlap: 3 words |
| Explain variance in ML. | “Variance is the amount by which…” | Fallback | n/a |

*Table 4: Example of Answer Sources and Types in Edge Cases*

This multi-tiered reader module design ensures high answer reliability, maximizing the chance of providing a relevant, complete response for a broad spectrum of academic questions.

## 3.4 Observations & Limitations

Evaluation of the baseline retrieval-augmented question answering pipeline highlights several practical strengths and key limitations in handling complex academic documents.

**Key Observations:**

* The combination of **hybrid chunking** (merging and splitting paragraphs/sentences by length), dense embeddings (all-MiniLM-L6-v2), and FAISS-based retrieval enables the system to surface relevant passages, even when query phrasing differs from the source.
* The BERT-large SQuAD reader shows strong accuracy for fact-based questions where answers are clearly stated within the top retrieved chunk.
* Keyword-based and fall-back heuristics provide robustness, ensuring the system delivers some answer even when the neural model’s confidence is low.

**Limitations and Challenges:**

* **Context Fragmentation:** Only one chunk is used for answer extraction, so questions requiring information across multiple sections or paragraphs are not well-supported.
* **Extractive-only Answers:** The pipeline can only extract literal spans; it cannot synthesize or rephrase for explanatory or multi-part queries.
* **Chunking Strategy Sensitivity:** The effectiveness of retrieval and answer extraction depends heavily on how chunks are formed; improper chunk size or grouping may lead to lost context or irrelevant content.
* **Implicit and Scattered Answers:** When answers are implied or distributed across sentences, the system may fail to retrieve a satisfactory result.
* **Resource Scalability:** As the academic corpus grows, embedding and search operations become more demanding, even with FAISS optimizations.
* **Model Limitations:** Answer quality can be limited by the generalization and vocabulary coverage of the pre-trained models, particularly in specialized subject areas.

# Chapter 4 Modern RAG System with LLMs & Ecosystem Tools

Recent advancements in large language models (LLMs) and the open-source NLP ecosystem have greatly enhanced the potential of academic question-answering systems. This chapter presents a fully implemented **modular Retrieval-Augmented Generation (RAG) pipeline**, deployed with an integrated **React frontend** and **Python FastAPI backend**, and supporting multiple LLM backends including **Gemini**, **Groq**, and the locally deployable **Ollama** for on-premises inference.

The system incorporates **advanced document ingestion and parsing with LlamaParse**, **context-aware semantic chunking**, **high-performance embeddings with persistent ChromaDB storage**, and a **feedback-aware retrieval mechanism** that continuously improves answer quality. It has been designed for high scalability, low-latency responses, and transparent provenance, ensuring that every generated answer is both contextually relevant and traceable to its source. This chapter details each stage of the pipeline, connecting user-facing interactions with backend processing logic, and illustrating how the system delivers accurate, flexible, and explainable answers for academic use.

## 4.1 Motivation for Advancing the Pipeline

While the traditional retrieval-augmented QA pipeline provides a strong foundation for academic search, its limitations become increasingly apparent as the complexity and scale of educational content grow. Three primary challenges drive the motivation for a modernized, LLM-centric architecture:

**1. Scalability and Flexibility:**

The classic pipeline relies on static chunking and embedding workflows, which become cumbersome as the volume and diversity of course materials increase. Adapting to new document formats, custom chunking logic, or the addition of new data sources often requires manual intervention or pipeline reconfiguration.

**2. Limitations of Extractive QA:**

BERT-based models, though powerful for extractive tasks, are inherently limited to returning verbatim spans from retrieved chunks. They struggle with open-ended, generative, or multi-hop questions, and cannot synthesize information from across multiple passages.

**3. Advances in LLMs and Tooling:**

The emergence of highly capable large language models—such as Gemini, Groq, and locally-deployable models via Ollama—enables true generative QA, where answers can be synthesized, summarized, or reworded as needed. At the same time, ecosystem tools like LangChain and ChromaDB offer modular, extensible building blocks for robust retrieval, prompt chaining, and integration with multiple LLMs, making the system easier to scale and maintain.

These factors collectively motivate the transition to a **modular, LLM-driven RAG architecture**—one designed to overcome the limitations of the traditional baseline, adapt to diverse academic domains, and deliver context-aware, high-quality answers at scale.

## 4.2 New Architecture Overview

The modern Retrieval-Augmented Generation (RAG) pipeline is composed of tightly integrated, modular components designed to efficiently process academic documents and answer natural-language queries with both precision and traceability. The system brings together cutting-edge open-source tooling, robust vector storage, and multi-LLM orchestration to deliver a responsive and context-aware question-answering experience.

The key components include:

* **Document Ingestion Module:**

Allows users to upload academic handouts in .pdf or .docx format. The uploaded documents are tagged with subject metadata and passed through a preprocessing pipeline for parsing and segmentation.

* **Parsing Engine (Standard + LlamaParse):**

Handles both conventional single-column documents and structurally complex ones, such as multi-column layouts, tables, and mathematical expressions. LlamaParse is selectively applied based on file structure to retain fidelity in academic semantics.

* **Semantic Chunking & Embedding (via BGE + FastEmbed):**

The parsed content is segmented into context-aware chunks, preserving logical boundaries like section headers and formula explanations. Each chunk is embedded using the BAAI/bge-base-en-v1.5 model via FastEmbed to produce dense, semantically rich vectors.

* **Vector Store (ChromaDB):**

All chunk embeddings are stored in a persistent ChromaDB instance, allowing high-throughput semantic search over large and growing document sets. The store supports metadata tagging (subject, page number, etc.) for fine-grained retrieval control.

* **User Interaction Layer (Query Input + LLM Selector):**

Users submit their question through the UI, selecting the academic subject and choosing their preferred LLM backend (Gemini, Groq, or Ollama). This selection dynamically configures the answer generation pipeline.

* **Semantic Retriever (LangChain Orchestrated):**

The user query is embedded and compared against the stored vectors in ChromaDB to fetch the top-k most relevant chunks. Context expansion ensures adjacent content is included to improve grounding.

* **LLM Answer Generator (Gemini / Groq / Ollama):**

The retrieved chunks and question are compiled into a prompt and passed to the selected LLM backend. Gemini and Groq offer cloud-hosted generative capabilities, while Ollama enables local inference—ideal for privacy-conscious deployments.

* **Answer Polishing & Provenance (spaCy):**

The raw LLM output is refined for readability and conciseness using spaCy’s NLP tools. Metadata such as document name and page number are attached for transparency.

* **UI Display + Feedback Capture:**

The final answer, along with expandable supporting context and citations, is presented in the frontend. Users may provide binary feedback (Helpful / Not Helpful) and optional comments, which are stored for analysis and pipeline improvement.

### 4.2.1 System Design and Data Flow

The enhanced system begins with a user submitting a natural-language query through the web interface, selecting the **subject domain** and optionally choosing the **preferred LLM backend** (**Gemini**, **Groq**, or **Ollama**). The query is **embedded** and processed by the **semantic retrieval module**, which searches a **persistent ChromaDB vector store** populated with **embeddings** generated from ingested academic documents (**PDF/DOCX**). Retrieval incorporates **context expansion** and **metadata filtering** to ensure that relevant surrounding content—such as adjacent paragraphs, equations, or diagrams—is also included.

The most relevant document chunks are then passed into the **LLM orchestration layer**, where the selected backend generates an answer grounded in the retrieved context. The answer undergoes post-processing with **spaCy** for **linguistic polishing** and is annotated with **provenance metadata**, enabling users to trace it back to the original source material. Finally, the **polished answer**, supporting excerpts, and citations are presented in the UI, where the user can provide **binary feedback and optional comments**. This feedback is logged for evaluation and can be leveraged to **improve retrieval ranking** and **overall answer quality** in future interactions.

Please refer to Figure 7 below for a high-level overview of the modern modular RAG pipeline architecture, illustrating the flow of data and modular integration of each component.

A group of colorful rectangular labels

AI-generated content may be incorrect.

*Figure 7: Overview of the modern modular RAG pipeline architecture*

Table 5 below summarizes the key differences between the baseline and modern RAG architectures. The modern system offers improved scalability, flexibility, and answer quality through modular orchestration, persistent vector storage, and generative LLM capabilities, while also being future-ready for local LLM integration.

|  |  |  |
| --- | --- | --- |
| Feature | Baseline System | Modern RAG System |
| Orchestration | Hardcoded pipeline | LangChain modular flows |
| Vector Store | FAISS (in-memory) | ChromaDB (persistent) |
| Reader Model | BERT-large (extractive) | Gemini/Groq (LLM, generative), Ollama (Local) |
| Document Formats | DOCX, PDF (manual) | Multi-format, extensible |
| Chunking Strategy | Hybrid size-based | Pluggable/context-aware |
| Retrieval Flexibility | Top-k only | Multi-hop, hybrid |
| Scalability | Limited | High |
| Answer Style | Extractive | Generative/explanatory |
| Integration Ease | Low (manual code) | High (configurable) |

*Table 5: comparison of the baseline and modern RAG pipeline architectures*

With this modular architecture in place, each stage of the pipeline—from document upload and parsing to semantic chunking, retrieval, LLM-based answer generation, and feedback capture—operates in a cohesive, extensible flow.

The subsequent sections explore each of these components in greater detail, highlighting their design, implementation, and role within the end-to-end academic question-answering system.

## 4.3 Modular Retrieval Layer: Upload to Semantic Retrieval

The retrieval backbone of the academic question-answering system is designed to transform raw educational documents into semantically searchable units—delivering context-rich content to downstream language models. This layer is responsible for everything from handling multi-format document uploads to generating dense vector embeddings and retrieving top-k relevant content chunks with high precision.

What distinguishes this system from conventional document search workflows is the integration of several specialized modules, including **LlamaParse** for structurally complex files, **semantic chunking** strategies tailored to academic writing, **FastEmbed-powered BGE embeddings**, and **ChromaDB**, a persistent high-performance vector store. These components work together in a tightly orchestrated pipeline that supports subject-specific indexing, advanced metadata tagging, and query-time filtering to maintain both scalability and academic relevance.

In the following subsections, we break down the entire retrieval pipeline into clearly defined stages—starting with how documents are uploaded and parsed, and concluding with how semantically relevant context is retrieved in response to a user query. Each component will be examined in terms of its theory, implementation role, and contributions to answer accuracy.

### 4.3.1 Document Upload & Subject Mapping

The document upload interface marks the entry point of the academic question-answering system, where users provide their subject-specific learning materials. The interface is designed to be intuitive and flexible, supporting documents in both **PDF** and **DOCX** formats—allowing a wide range of content including lecture notes, handouts, and technical reports.

Each upload begins with the user selecting a **subject domain** (e.g., *Machine Learning*, *Natural Language Processing*) and optionally supplying a brief description. These details are important not just for user organization, but also for how the backend stores and later retrieves the document.

Once a file is selected and submitted, the **frontend sends a structured request** to the backend upload API. This request contains:

* The selected subject
* The uploaded document (as FormData)
* A boolean flag indicating whether to use **LlamaParse**
* An optional description

On the backend, the file is saved inside a central directory named /uploads/. To ensure separation by subject, each file is **automatically prefixed with the subject name**, forming paths such as:

*/uploads/NLP\_NLP\_Sem2\_Tables.docx*

*/uploads/MachineLearning\_ML\_Handout1.pdf*

This filename convention guarantees **subject-level isolation** during future retrieval and indexing, while also making it easy to trace the origin of any document.

Users also have the option to enable **advanced parsing** for structurally complex files. If the **LlamaParse toggle** is selected, this preference is recorded during upload and will later influence how the file is parsed when ingestion is triggered.

A close-up of a computer screen

AI-generated content may be incorrect.

This flow captures the document ingestion entry point: users upload a file tagged by subject, optionally enabling LlamaParse. Once stored with a subject-prefixed name, the file becomes ready for ingestion through a separate backend trigger.

The **user interface** plays a crucial role in guiding this process. A clean upload panel allows the user to pick a subject, select files, and toggle advanced parsing. Upon successful upload, the UI triggers a backend ingestion pipeline and confirms upload status with appropriate notifications.

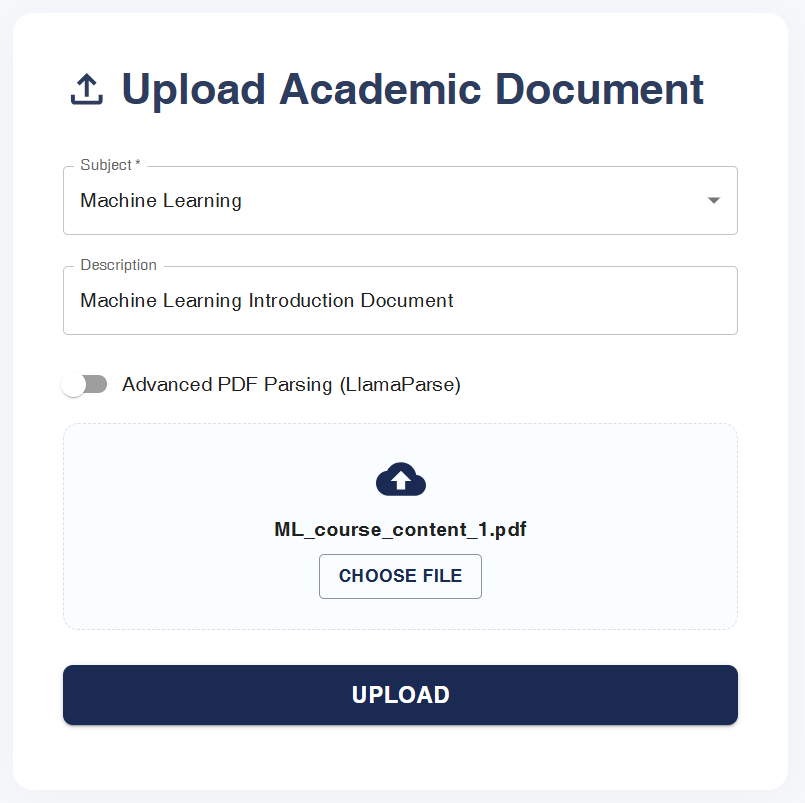


Figure X: Document Upload Interface with Subject Selection and LlamaParse Toggle

Internally, the uploaded documents are indexed with associated metadata such as file name, parser used, and storage location. This information supports both traceability and chunk-level provenance in later stages. An example of this internal structure is outlined below:

|  |  |  |
| --- | --- | --- |
| Subject | File Name | Stored Path |
| Machine Learning | ML\_course\_content\_1.pdf | /uploads/Machine Learning/ ML\_course\_content\_1.pdf |
| Natural Language Processing | NLP\_Sem2\_Tables.docx | /uploads/NLP/NLP\_Sem2\_Tables.docx |

### 4.3.2 Document Ingestion & Parsing

Once a document is successfully uploaded, the next step in the pipeline is ingestion—a process that transforms the raw file into a format suitable for retrieval-based question answering. This stage focuses on handling relatively clean academic documents (e.g., lecture notes, handouts) that do not require layout-aware parsing. The system extracts meaningful text from each page, associates it with metadata, and prepares it for downstream processing such as semantic chunking and embedding.

**Triggering the Ingestion Process**

The ingestion process is initiated by the user via the web interface by clicking the **“Process”** or **“Ingest”** button shown after a successful file upload. This action sends a structured POST request to the backend ingestion API.

The request body includes three key fields:

* **subject**: The subject domain selected by the user during upload (e.g., "Machine Learning")
* **filename**: The name of the uploaded file (already prefixed with subject to ensure separation)
* **use\_llamaparse**: A boolean flag indicating whether advanced parsing should be used (set to false for standard parsing)

Upon receiving the request, the system first validates that the file exists in the /uploads/ directory. If valid, the system proceeds with standard parsing logic depending on the file type—**PDF** or **DOCX**.

**File-Type-Specific Parsing Logic**

To support multiple document formats, the system dynamically selects the appropriate parser:

* **PDFs** are processed using **PyPDFLoader**, which extracts text from each page while preserving academic formatting where possible.
* **DOCX** files are handled using **Docx2txtLoader**, which flattens Word document content into a plain-text format suitable for further processing.

These loaders abstract away file-specific quirks and return a clean, page-wise representation of the content.

Only pages that contain a meaningful amount of text are retained for the next stages. Blank pages, metadata-only pages, or pages with unreadable formatting are skipped to ensure that downstream modules operate on quality data.

A screenshot of a computer program

AI-generated content may be incorrect.

**Figure X: Flowchart for Standard Document Ingestion and Parsing**

This diagram illustrates the backend flow triggered by the “Ingest” button on the UI. Based on file type, the system uses either PyPDFLoader or Docx2txtLoader for standard parsing and then prepares the text for semantic chunking. **A screenshot of a computer

AI-generated content may be incorrect.**

The web interface makes this entire flow seamless for the user. After selecting a file and uploading it, a dedicated button allows them to begin processing the document. A backend notification confirms successful ingestion and transitions the system to the chunking phase.

Example Metadata Table After Parsing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Page No. | File Name | Subject | Parsing Method | Used in Chunking? |
| 1 | ML\_course\_content\_1.pdf | Machine Learning | PyPDFLoader | Yes |
| 2 | ML\_course\_content\_1.pdf | Machine Learning | PyPDFLoader | No |
| 3 | NLP\_Sem2\_Tables.docx | NLP | Docx2txtLoader | Yes |

Each parsed page is enriched with subject metadata and page number, enabling traceability and clean separation across domains. The system is now prepared to perform **semantic chunking**, where these pages are further divided into idea-preserving text units for intelligent indexing.

### 4.3.2.1 Advanced Parsing with LlamaParse

In conventional academic question-answering systems, PDF parsing is often handled through lightweight tools like PyPDFLoader, which operate on plain text extraction. However, educational documents frequently include complex layouts—multi-column text, embedded tables, mathematical equations, structured headers, and figures—which often break standard parsers. To address this, our system integrates **LlamaParse**, an LLM-powered document parser built for robust, structure-preserving extraction.

**What is LlamaParse?**

**LlamaParse** is a state-of-the-art document-parsing engine offered by **LlamaIndex**. It is designed to convert rich-format documents like PDFs, DOCX, PPTX, EPUB, and others into semantically structured Markdown or plain text. Its capabilities are powered by a generative LLM backend, allowing it to follow custom parsing instructions and preserve complex layouts such as:

* Tabular data
* Diagrams and figures
* Hierarchical sectioning (headings, subsections)
* Mathematical notations
* Page-level metadata

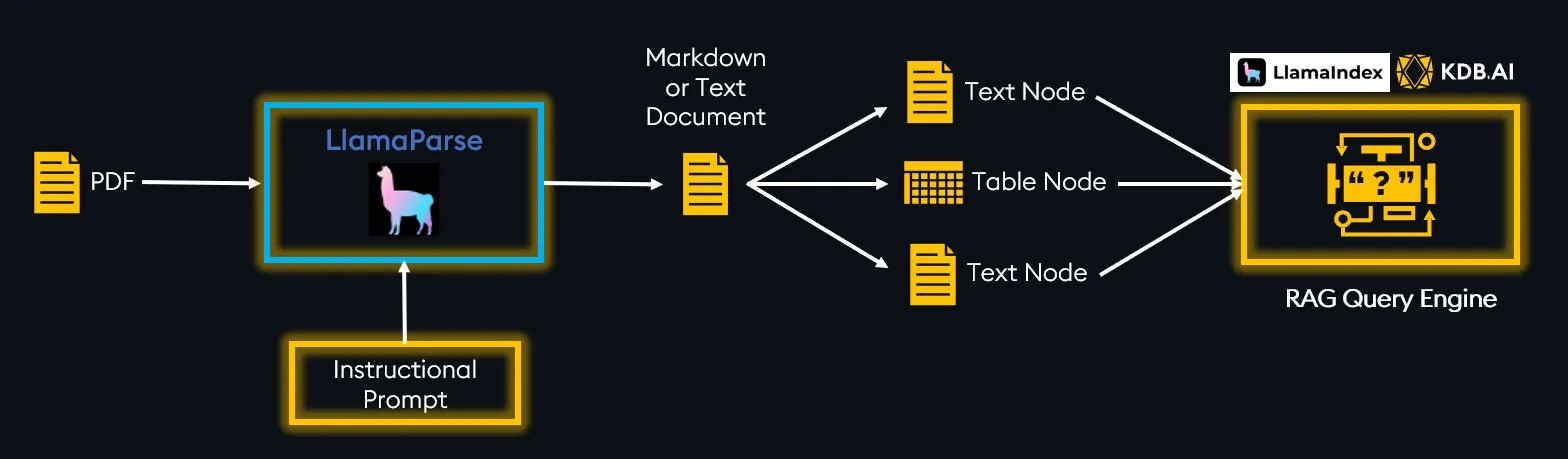


Image X: Ingest Complex Documents with LlamaParse

The core functionality of LlamaParse is to enable the creation of retrieval systems over these complex documents like PDFs. LlamaParse does this by extracting data from these documents and transforming them into easily ingestible formats such as markdown or text. Once the data is transformed, it can be embedded and loaded into your RAG pipeline.

**Overview of LlamaParse Features:**

* **Supported file types**: PDF, .pptx, .docx, .rtf, .pages, .epub, etc…
* **Transformed output type**: Markdown, text
* **Extraction Capabilities**: Text, tables, images, graphs, comic books, mathematics equations
* **Customized Parsing Instructions:**Since LlamaParse is LLM enabled, you can pass it instructions just as if you were prompting an LLM. You could use this prompt to describe the document therefore adding more context for the LLM to use while parsing, indicate how you want the output to look, or ask the LLM to do preprocessing during parsing like sentiment analysis, language translation, summarization, etc…
* **JSON Mode:**Outputs the complete structure of the document, extracts images with size and location metadata, extracts tables in JSON format for easy analysis. This is perfect for custom RAG applications where document structure and metadata are used to maximize informational value of documents and for citing where in a document retrieved nodes originate.

**The Markdown Advantage**

There are some unique advantages to LlamaParse transforming a PDF into markdown format. Markdown specifies the inherent structure of the document by identifying structural elements like titles, headers, subsections, tables, and images. This may seem trivial, but since markdown identifies these elements, we can easily split a document into smaller chunks based on structure using specialized parsers from LlamaIndex like the [**MarkdownElementNodeParser()**](https://docs.llamaindex.ai/en/stable/api_reference/node_parsers/markdown_element/). The result of representing a PDF file in markdown format is it enables us to extract each element of the PDF and ingest them into the RAG pipeline.

**Key Advantage**:

Unlike standard extractors, LlamaParse uses the Markdown format to retain document hierarchy, which significantly improves downstream chunking, semantic search, and citation accuracy in RAG pipelines.

**How We Use It in Our Pipeline**

In our architecture, when a user uploads a document and enables the **"Advanced PDF Parsing (LlamaParse)"** toggle, the backend pipeline triggers LlamaParse for that specific file. The logic is configured via a Boolean flag (***use\_llamaparse***) sent to the ingestion API.

Here is how the process unfolds:

1. **Document Parsing Instruction**:

The parser is initialized with a domain-specific instructional prompt—e.g., explaining the nature of the uploaded document (like “ML notes” or “research paper”), so it can align extraction with expected content semantics.

1. **Structured Output**:

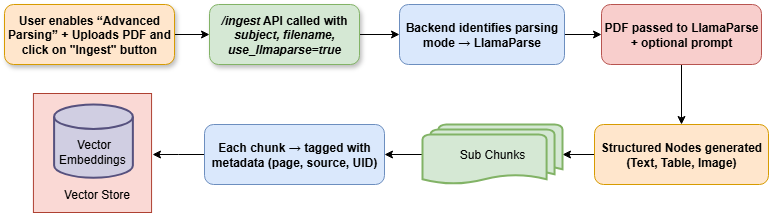
LlamaParse converts the PDF into structured nodes (e.g., text, table, image), each enriched with metadata such as page number, chunk type, and position in the hierarchy.

1. **Chunking Strategy**:

Using LangChain’s **RecursiveCharacterTextSplitter**, we break down each parsed node into smaller semantic sub-chunks. This improves granularity in downstream retrieval.

1. **Metadata Enrichment**:  
   Each chunk is tagged with fields like:
   * *source\_pdf*
   * *parsed\_by* (marked as "*llamaparse*")
   * *parent\_chunk\_type* (e.g., Table, Text)
   * *chunk\_uid* (hashed ID for traceability)  
     This metadata later powers **feedback-based re-ranking**, document traceability, and UI-level provenance display.

Below is a simplified block-style flow representation showing what happens under the hood when advanced parsing is enabled:



**Figure X** illustrates the enhanced ingestion pipeline when **advanced LlamaParse-based parsing** is enabled. Upon upload, the system routes the document through a parsing mode switch, invoking LlamaParse to extract structured nodes such as text, tables, and images. These nodes are then semantically chunked, enriched with metadata (including page number, document source, and chunk UID), and finally stored as vector embeddings in the persistent vector store (ChromaDB). This structured approach ensures high-fidelity semantic indexing and transparent traceability of all downstream answers.

**Sample Instructional Prompt (used in our code)**

“The provided document is a quarterly report filed by Uber Technologies, Inc. with the SEC...

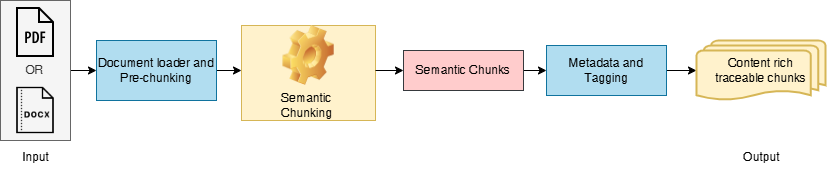
It contains many tables. Try to be precise while answering the questions.”  
  
This prompt allows LlamaParse to better interpret the domain, preserve semantic context, and assist in aligning the document layout with our RAG pipeline’s goals.

**Sample Screenshot of successfully parsed PDF available on llmaIndex Cloud:**

****

### 4.3.3 Document Upload & Subject Mapping

**Semantic chunking** is a transformative step that goes beyond basic length-based splitting or hybrid chunking. Rather than merely dividing text by word count or paragraph boundaries, semantic chunking aims to segment documents into coherent, meaning-rich passages that align with natural topic shifts and content boundaries. This ensures that each chunk is not only manageable in size but also maximally relevant for retrieval and answer generation.



*Figure 8: Semantic Chunking Workflow*

In practice, the pipeline leverages **LangChain’s SemanticChunker**, which utilizes a pre-trained embedding model to identify content boundaries based on semantic coherence rather than superficial markers. Each document is initially pre-chunked—often at the page or paragraph level—to simplify processing. The SemanticChunker then analyses these segments, using statistical or percentile-based thresholds on embedding similarity to determine where one idea ends and another begins.

The goal of semantic chunking is to maintain the structural and logical integrity of academic documents, preserving grouping between **headings and their corresponding explanations**, **formulas with descriptions**, and **tables with associated analysis**. This human-aligned segmentation ensures that each chunk makes sense independently while being optimized for retrieval.

This approach produces chunks that more closely match the way humans naturally navigate and reference academic material, leading to higher retrieval precision. Every chunk created is enriched with essential metadata to support **traceability, explainability, and feedback-aware retrieval**. These include:

|  |  |
| --- | --- |
| Field | Description |
| source\_pdf | The original file name |
| page | Page number (for both PDF and DOCX) |
| subject | Subject domain selected by user |
| chunk\_uid | |  | | --- | |  |  |  | | --- | | Unique hash of chunk content | |

A sample output from the semantic chunking stage is illustrated below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Chunk\_uid | Source\_pdf | Chunk Text Preview | | Page |
| Machine\_Learning\_ML\_course\_content\_1.pdf\_3\_d41ab0c54d | Machine\_Learning\_ML\_course\_content\_1.pdf | 1.5 Challenges in Machine Learning Despite its power, deploying… | 2 | |
| Machine\_Learning\_ML\_course\_content\_1.pdf\_3\_3f1169e00e | Machine\_Learning\_ML\_course\_content\_1.pdf | Recall (Sensitivity): Out of all actual positives, how many were correctly predicted… | 7 | |
| Machine\_Learning\_ML\_course\_content\_2.pdf\_1\_53c38d0b0f | Machine\_Learning\_ML\_course\_content\_2.pdf | to the observed data, provided there are enough relevant examples in memory… | 1 | |

*Table 6: Example of Semantic Chunk Metadata*

### 4.3.4 Advanced Embedding with FastEmbed (BAAI/bge-base-en-v1.5)

A pivotal step in the modern retriever pipeline is converting each semantic chunk into a vector that accurately represents its meaning for semantic search. This process, known as **embedding**, is essential for allowing the system to retrieve content based on conceptual similarity, rather than just matching keywords.

In this work, the **FastEmbed** framework, utilizing the **BAAI/bge-base-en-v1.5** model, is employed for this purpose. This state-of-the-art model produces 768-dimensional embeddings, designed to capture the core semantics of sentences and paragraphs—making it highly effective for academic QA over diverse content.

Before delving deeper, consider the flow illustrated in Figure 9 below:

A screen shot of a computer

AI-generated content may be incorrect.

*Figure 9: Embedding Workflow for Semantic Chunks*

The process begins with the semantic chunks produced in the previous stage. Each chunk enters the embedding model, depicted in the center box. The FastEmbed (BAAI/bge-base-en-v1.5) model applies deep language understanding to encode the full meaning of the text—not just the words used—into a 768-dimensional vector. This is shown in the diagram by the downward arrow from "Semantic Chunks" into the "Embedding Model" box. The resulting output, a dense vector, is directed into the "Vector Embeddings" box, representing a collection of these embeddings for all chunks in the academic corpus.

This visual flow captures the essence of the embedding phase: it bridges raw, meaning-rich text with a mathematical space optimized for rapid, context-sensitive retrieval. By mapping all document chunks in this way, the system can efficiently compare user queries to the most semantically relevant passages, regardless of surface phrasing or document structure.

A sample of the generated embeddings is shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| Chunk\_uid | Source\_pdf | Chunk Preview | Embedding (First 8 Dims) |
| Machine\_Learning\_ML\_course\_content\_1.pdf\_3\_d41ab0c54d | Machine\_Learning\_ML\_course\_content\_1.pdf | 1.5 Challenges in Machine Learning Despite its power, deploying… | 0.071, -0.021, 0.122, 0.043, -0.055, 0.108, 0.023, -0.037 |
| Machine\_Learning\_ML\_course\_content\_1.pdf\_3\_3f1169e00e | Machine\_Learning\_ML\_course\_content\_1.pdf | Recall (Sensitivity): Out of all actual positives, how many were correctly predicted… | 0.091, -0.014, 0.113, 0.037, -0.051, 0.104, 0.021, -0.029 |

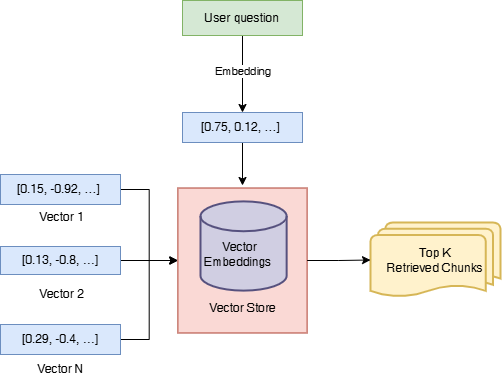
*Table 7: Example Embeddings for Semantic Chunks*

### 4.3.5 Persistent Vector Storage with ChromaDB

Once each semantic chunk is transformed into a dense vector using the FastEmbed framework, the system must store, index, and retrieve these embeddings efficiently. For this, the system employs **ChromaDB**, a lightweight yet production-ready vector database optimized for **fast semantic search** and **persistent storage**.

ChromaDB is designed to handle large-scale academic corpora, supporting rapid insertion, querying, and deletion of high-dimensional vectors. Its persistence feature allows the vector index to be saved and reloaded from disk, ensuring that the system can handle evolving document sets without the need to recompute all embeddings at each startup. This makes the pipeline suitable for real-world, production-scale deployments in educational settings.

Figure 10 below depicts the flow of data and vectors into ChromaDB:



*Figure 10: Document and query embeddings into ChromaDB*

In this workflow, all chunk embeddings are stored in ChromaDB (center box). When a user submits a question, it is passed through the same embedding model and the resulting query vector is sent to ChromaDB. The vector store rapidly computes similarity scores and returns the top-k most similar document chunks for downstream LLM-based answer generation. This design not only speeds up retrieval, but also ensures persistence—vectors and metadata can be stored across sessions, and the system can scale to tens of thousands of documents.

During ingestion, the system uses **Chroma.from\_documents()** to store all vectorized chunks. At retrieval time, **Chroma()** is reloaded using the same directory path. This ensures session persistence and allows incremental document ingestion without loss of prior index data.

Table 8 below provides an example of the indexed data for a few chunks:

|  |  |  |  |
| --- | --- | --- | --- |
| Chunk ID | Source Document | Metadata (Page) | Vector Store Path |
| 001 | lecture1.pdf | 4 | /outputs/chroma\_machine\_learning. |
| 002 | lecture2.pdf | 7 | /outputs/chroma\_ machine\_learning |

*Table 8: Example Entries in ChromaDB Vector Store*

### 4.3.6 Retriever Object Creation from the Vector Store

The culmination of the enhanced retriever module is the creation of a standardized retriever object, which acts as the interface for semantic search over the indexed document chunks. Once all semantic chunks have been embedded and stored in ChromaDB, this object is initialized to efficiently surface the most relevant passages in response to search queries or for downstream answer generation.

The underlying logic connects the persistent vector store containing the indexed semantic chunks to the retriever component responsible for executing top-k similarity searches. This modular design ensures flexibility and maintainability, making it easy to adjust retrieval parameters or integrate additional functionality as the pipeline evolves.

Once all semantic chunks have been embedded and stored within the ChromaDB vector store, the system must prepare a mechanism to retrieve the most relevant chunks during inference. This is achieved by constructing a **retriever object**, a key abstraction provided by LangChain, which acts as the interface between user queries and stored vector data.

This design decouples retrieval logic from vector storage, allowing flexible parameter tuning such as:

* Number of chunks (top\_k) to retrieve
* Embedding model variations
* Integration with feedback re-ranking

This process is illustrated below:

A black and white rectangle with black text

AI-generated content may be incorrect.

*Figure 11: Construction of a retriever object from the ChromaDB vector index*

By encapsulating the retrieval logic in this way, the modern RAG system enables robust, context-aware selection of document passages assuring that the most relevant academic content is efficiently delivered to the LLM-based answer generation module.

This encapsulation ensures that when a user poses a question, the system efficiently identifies and forwards only the most semantically aligned passages—forming the bridge between storage and reasoning.

# Chapter 5 Reader Module and User Interaction Layer

The **Reader Module** is the final stage in the pipeline where user questions meet meaningful answers. It connects retrieved context chunks to large language models (LLMs), performs structured prompt construction, invokes LLMs for response generation, and then presents the answers to the end user in a clean, interactive interface.

This module is not just about generating responses — it is about delivering **grounded**, **context-aware**, and **traceable** answers, leveraging powerful LLM backends and clean UX workflows. The system supports multiple backends, including:

* **Google Gemini (gemini-1.5-flash-latest)** via Gemini API
* **Llama 3 (llama3-8b-8192)** hosted on **Groq** for lightning-fast inference
* **Ollama-compatible local LLMs**

Alongside answer generation, the Reader module also ensures:

* Injection of **retrieved context** into the prompt
* Display of **retrieval metadata** (source document, page, rank)
* **Feedback collection** from users for future ranking improvements

This chapter explains the complete reader pipeline — from query submission to answer presentation — including API orchestration, prompt formatting, LLM execution, and result visualization.

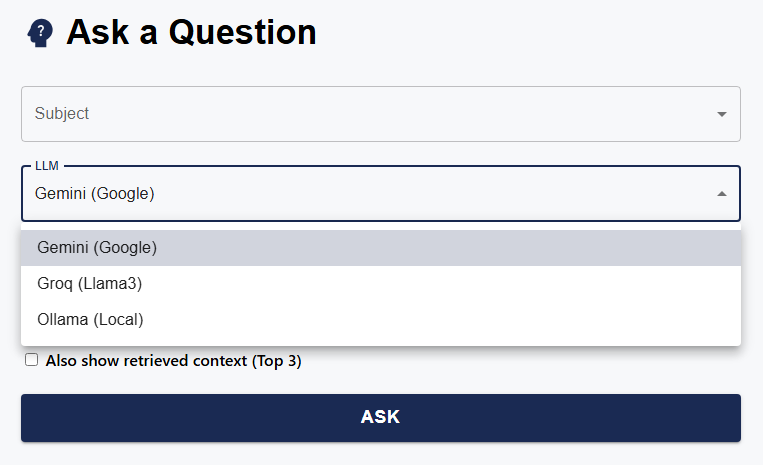
## 5.1 Multi-LLM Answer Generation and Orchestration

The Reader module is responsible for transforming retrieved chunks into final, polished answers using large language models (LLMs). A key enhancement in this work is the support for **multiple LLM backends**, offering deployment flexibility, performance tuning, and privacy control.

Rather than coupling the system to a single model, the architecture allows seamless switching between **Google Gemini**, **Llama 3 via Groq**, and **locally hosted LLMs via Ollama**. This design empowers users and developers to optimize for latency, cost, and deployment environment — whether cloud-based or offline.

**UI Interaction**

On the question-answering interface, users can choose their preferred LLM via a dropdown menu. Once a model is selected, the system routes the query through the corresponding backend and returns the generated response along with relevant context.



### Overview of Each LLM

1. **Google Gemini (gemini-1.5-flash-latest)**

Gemini is Google’s cutting-edge transformer-based LLM known for its **robust context handling** and **scalable prompt processing**. The variant integrated here — **Gemini 1.5 Flash** — is optimized for speed and cost, making it suitable for real-time academic question answering.

* Supports **large context windows** (~1M tokens max, ~32k typical)
* Handles **structured academic prompts** effectively
* Hosted via Google’s official Generative AI API

Gemini excels at answering factual questions by extracting key ideas from the prompt and delivering concise yet informative responses.

1. **LLaMA 3 (8B, via Groq)**

LLaMA 3 is a state-of-the-art open-weight LLM by Meta, and it’s hosted on Groq’s **ultra-low-latency inference platform**. This integration offers powerful academic performance with near-instantaneous output.

* Model: llama3-8b-8192 (8B parameters, 8192-token context)
* Hosted via **Groq API**, enabling lightning-fast generation
* Produces well-grounded, structured answers
* Ideal for **chat-like academic experiences**

Groq provides a practical middle-ground between open-source flexibility and production-grade performance.

1. **Ollama (Local Inference)**

Ollama enables self-hosted LLM execution on a developer’s machine or private server. This work supports any LLM served via Ollama, with defaults like llama2 or llama3.

* Ensures **full privacy**: no data leaves the local system
* Supports **offline usage**, ideal for institutional or restricted environments
* Configurable with different model sizes and memory footprints
* Typically used with GPU acceleration for performance

Ollama integration allows this system to run **independently of external APIs**, giving end users full control over data, deployment, and customization.

### 5.1.1 API Endpoint: /ask-question – Unified Interface for Answer Generation

The /ask-question endpoint acts as the **central API** for the question-answering workflow. Once the user submits a question through the frontend, this route orchestrates the retrieval and LLM generation process based on the selected **subject domain** and **LLM backend** (Gemini, Groq, or Ollama).

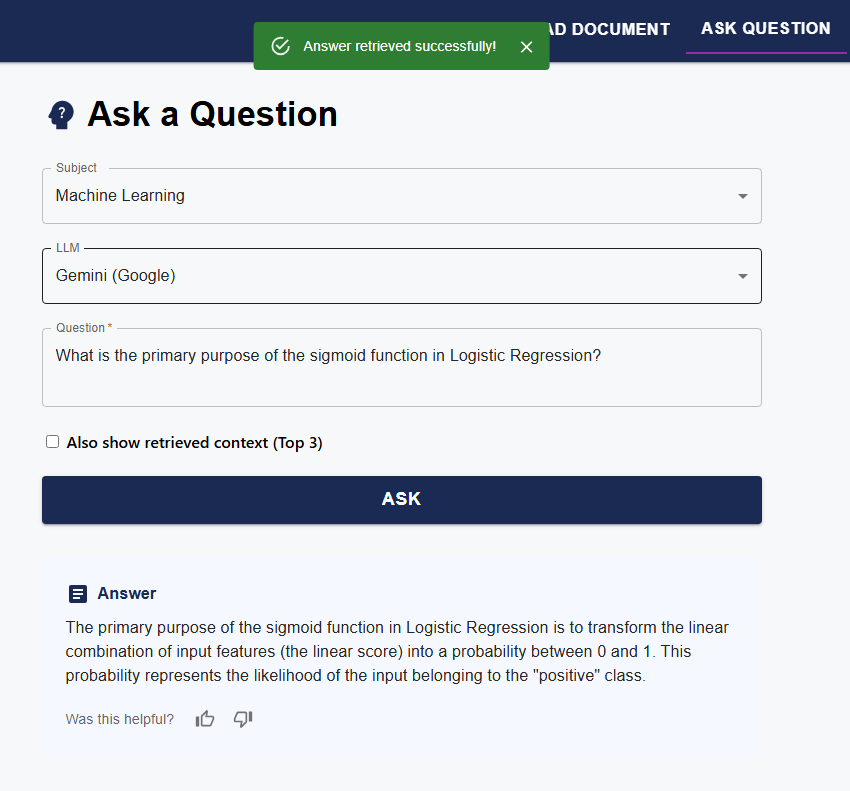
Upon receiving the POST request, the backend constructs a **retrieval-augmented chain**, sends the **user query + retrieved chunks** to the selected LLM, and formats the final response. Additionally, when the optional flag add\_context is enabled, the endpoint also includes a list of the top-5 retrieved passages, each with source metadata (filename, page, etc.) for provenance tracking and display on the UI.

|  |  |  |
| --- | --- | --- |
| **Parameter / Field** | Type | Description |
| subject | string | Subject selected by user (used to find the right ChromaDB folder) |
| question | string | 7 |
| llm\_choice | string | Backend LLM to use: "gemini", "groq", or "ollama" |
| add\_context (optional) | boolean | Whether to return supporting context passages |
| Response: `answer` | string | Final answer generated by the selected LLM |
| Response: `qa\_session\_id | UUID | Unique identifier for feedback and tracking |
| Response: `context`(optional) | list[dict] | Top retrieved passages with metadata: `source\_pdf`, `page`, `preview`, `rank` |

**LLM Answer Response (Structured View) when add\_context is ’false’**

|  |  |
| --- | --- |
| **Field** | Description/Value |
| Answer | "The primary purpose of the sigmoid function in Logistic Regression is to transform the linear combination of input features (the linear score) into a probability between 0 and 1. This probability represents the likelihood of the input belonging to the 'positive' class." |
| QA Session ID | |  | | --- | | f6705050-bef6-42c3-825f-04b204cc91b2 |  |  | | --- | |  | |

The below interface demonstrates the primary QA interaction screen where a user selects the subject domain, chooses a preferred LLM backend (Gemini in this case), and submits a natural language query. The context toggle (Also show retrieved context) is **unchecked**, so only the **direct answer** from the selected LLM is displayed, without any additional supporting source snippets. The response is followed by a simple binary feedback option for quality assessment.



### 5.1.2 Caching Logic via rag\_chain\_cache To improve efficiency and reduce latency, the backend introduces a **caching mechanism** via a dictionary object named rag\_chain\_cache. This acts as a **memory-based store** for retrieval-generation chains that are reused across multiple queries within the same session or subject.

Each cache entry is indexed using a tuple: (chroma\_dir, llm\_choice). This ensures that for every combination of **subject-specific document collection** and **selected LLM backend**, the system maintains a **ready-to-use pipeline** consisting of the retriever and the LLM interface. If a request comes in and the corresponding chain already exists in memory, the system avoids reloading the embedding model or ChromaDB vector store—saving both time and compute.

This caching logic is seamlessly integrated into the /ask-question endpoint. It allows users to interactively submit multiple questions from the same subject without any degradation in performance, even when switching between different LLM backends like **Groq**, **Gemini**, or **Ollama**.

### 5.1.3 Prompt Construction with Retrieved Context

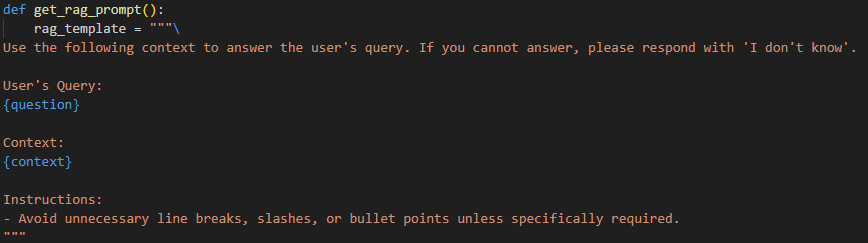
An essential component of the Reader module is the construction of an effective prompt for the LLM. The goal is to inject both the **user’s query** and the **retrieved academic content** into a format that guides the LLM toward accurate, grounded, and focused answer generation.

In this project, we use a **template-based approach** for prompt creation. The LangChain ChatPromptTemplate utility is employed to create consistent and well-structured prompts across different LLMs. Each prompt consists of:

* A system instruction block defining behavior
* The **user query** (e.g., "What is Logistic Regression?")
* The **retrieved context** (top-k semantic chunks from ChromaDB)
* Guidelines to avoid hallucinations or poor formatting

This strategy ensures **factual grounding**, **linguistic clarity**, and **reduced redundancy**, especially when used with models like Gemini or Llama 3, which are sensitive to context design.

**Code Snippet: Prompt Definition (LangChain-compatible)**

****

This prompt is injected into the LLM pipeline right before invoking the model, ensuring each answer is shaped by the most relevant academic content.

### 5.1.3 LLM-Based Answer Generation Workflow

After the retriever provides the top-k most relevant document chunks, these passages, along with the user’s question, are formatted into a prompt and submitted to the selected LLM. The initial LLM-generated answer is then passed through a dedicated polishing stage, which utilizes the **spaCy** natural language processing library for refinement. This polishing process may involve:

* Trimming redundant phrases,
* Removing irrelevant content,
* Standardizing terminology,
* Formatting the answer for clarity or length.

By leveraging spaCy, the system can perform advanced linguistic analysis, such as part-of-speech tagging and named entity recognition, to systematically clean and enhance the LLM output before presenting it to the user.



*Figure 12: LLM-Based Answer Generation and Polishing Workflow*

The final answer is polished and returned to the UI, with optional context and metadata for user review.

## 5.2 Answer Presentation with Context Preview and Metadata

In any Retrieval-Augmented Generation (RAG) system, presenting a fluent answer alone is not enough—**transparency and traceability** are vital. That’s where this answer presentation layer comes in. Once a user’s query is processed and an answer is generated by the selected LLM, the system complements it with a **contextual snapshot** of the retrieved information that influenced the response.

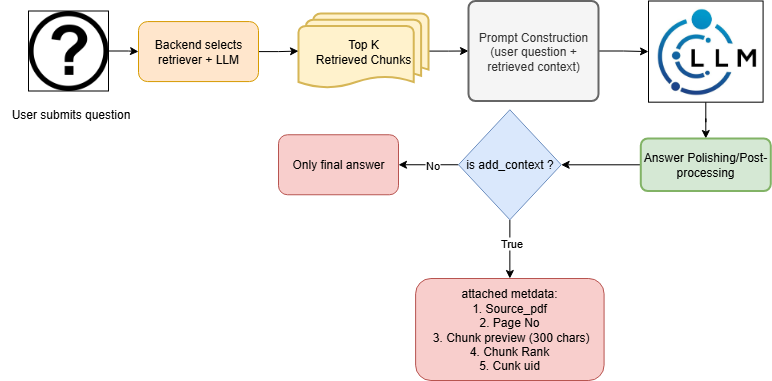
This phase closes the loop between **retrieval**, **reasoning**, and **user interaction**, giving users confidence in the source of the answers they receive. By showing **where the answer came from**, and **how relevant context chunks were selected**, the platform enhances educational usability and avoids black-box behavior.

**Core Goals of This Stage**

* Present the final **LLM-generated answer** in a readable form.
* Optionally show **retrieved context snippets** that contributed to the answer.
* Reveal supporting **metadata** such as source document name, page number, and retrieval rank.
* Enable structured backend caching (via session ID) to support **feedback collection**.

This metadata-driven approach allows users to expand and verify how the LLM interpreted the retrieved data—especially useful for academic use cases where correctness matters.

Below is the visual diagram that captures the flow from question to final answer presentation:



The above flow diagram provides a **comprehensive overview** of the answer generation and context preview process in the RAG system.

The flow begins with a **user question**, which triggers the backend to select the appropriate **retriever and LLM backend** (Gemini, Groq, or Ollama). The retriever then fetches the **Top-K Retrieved Chunks**, which are used alongside the user's query to perform **Prompt Construction**. This unified prompt is sent to the **LLM for response generation**.

A key decision point follows—**is add\_context set to true?** If not, only the **final answer** is returned to the user. However, if context is enabled, the backend includes **additional metadata** in the response, shown in below table:

| **Field** | **Description** |
| --- | --- |
| source\_pdf | File from which the chunk was retrieved (e.g., ML\_Notes.pdf) |
| page | Page number of the document where the chunk exists |
| preview | First ~300 characters of the chunk |
| chunk\_uid | Unique identifier used internally (not shown on UI) |
| rank | Position in top-k list (used for ordering) |

**Sample output generated:**  
  
{

"answer": "The primary purpose of the sigmoid function in Logistic Regression is to transform the linear combination of input features (the linear score) into a probability between 0 and 1. This probability represents the likelihood of the input belonging to the \"positive\" class.",

"qa\_session\_id": "761d38b4-297e-4a35-8ff7-537ba1eef940",

"context": [

{

"source\_pdf": "Machine\_Learning\_ML\_course\_content\_1.pdf",

"page": 12,

“rank”: 1,

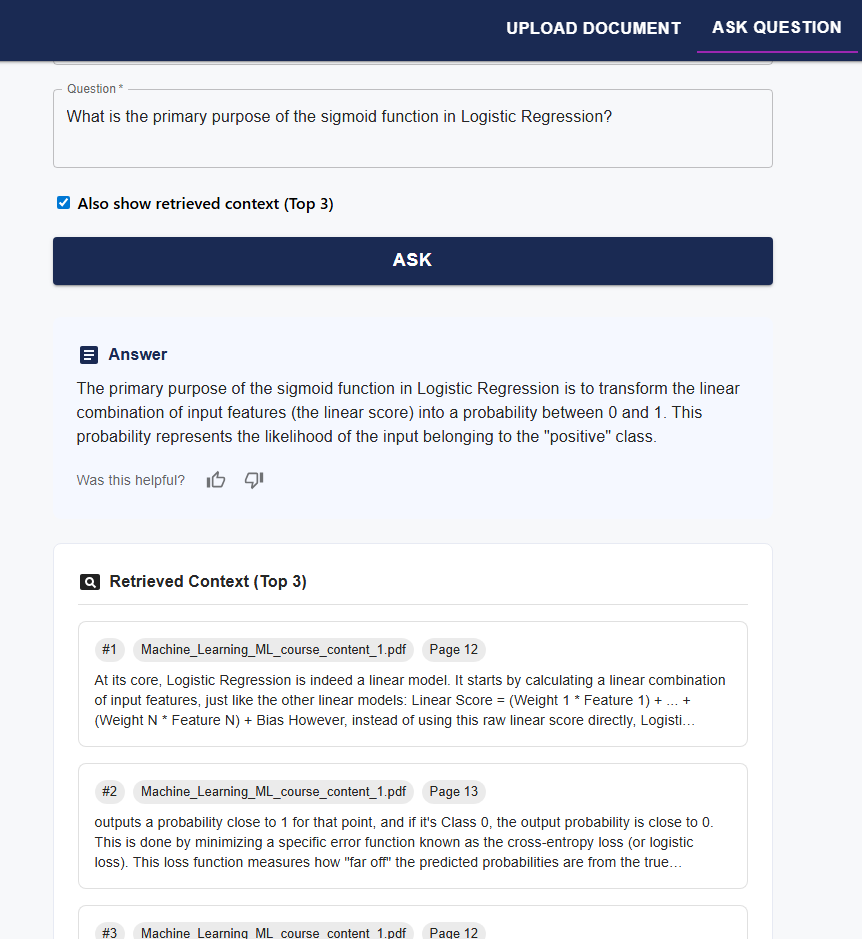
"preview": "At its core, Logistic Regression is indeed a linear model. It starts by calculating a linear combination of input features, just like the other linear models: Linear Score = (Weight 1 \* Feature 1) + ... + (Weight N \* Feature N) + Bias However, instead of using this raw linear score directly, Logisti…"

}

]

}

This metadata is also cached server-side using a qa\_session\_id, enabling accurate feedback submission in the next phase.



The image demonstrates how the system displays both the LLM-generated answer and the top 3 retrieved context chunks when the user enables the context preview option. Each snippet includes metadata like the source PDF and page number, enhancing traceability and user trust.

## 5.4 Feedback and Retrieval Adaptation

A powerful feature of this RAG-based QA system lies in its ability to learn from user interactions and adapt its retrieval process accordingly. Once an answer is generated and displayed to the user, the system provides an intuitive feedback interface—allowing the user to indicate whether the provided answer was **helpful or not**. This binary feedback mechanism is accompanied by an optional text comment for unhelpful responses. The feedback process serves two key roles: (1) collecting user insights on the system's accuracy, and (2) dynamically improving future retrieval quality through **feedback-aware reranking**.

Behind the scenes, a structured session-level feedback architecture is employed. When a question is asked, a unique qa\_session\_id is generated and cached in-memory (QA\_CACHE) along with three key artifacts: the original user query, a snapshot of the top retrieved chunks (each tagged with metadata like file name, page, and rank), and a short excerpt of the generated answer. This snapshot is crucial—it allows the system to later associate the user’s feedback with specific context chunks, even if the full documents have changed or the session has ended.

Once feedback is submitted, a JSONL-formatted log entry is written to a local file (feedback.jsonl). Each line corresponds to a structured feedback object and includes:

* The session ID
* Binary helpful status
* Optional user comment
* Retrieval snapshot (list of chunks returned)
* Timestamp and backend LLM used

**Sample feedback.jsonl entry**

{

"qa\_session\_id": "f6705050-bef6-42c3-825f-04b204cc91b2",

"helpful": false,

"comment": "Example feedback provided",

"llm\_backend": "gemini",

"timestamp": "2025-08-09T16:25:11.830668",

"query": "What is the primary purpose of the sigmoid function in Logistic Regression?",

"retrieval\_snapshot": [

{

"chunk\_uid": "Machine\_Learning\_ML\_course\_content\_1.pdf\_12\_abc123",

"source\_pdf": "Machine\_Learning\_ML\_course\_content\_1.pdf",

"page": 12,

"rank": 1

}

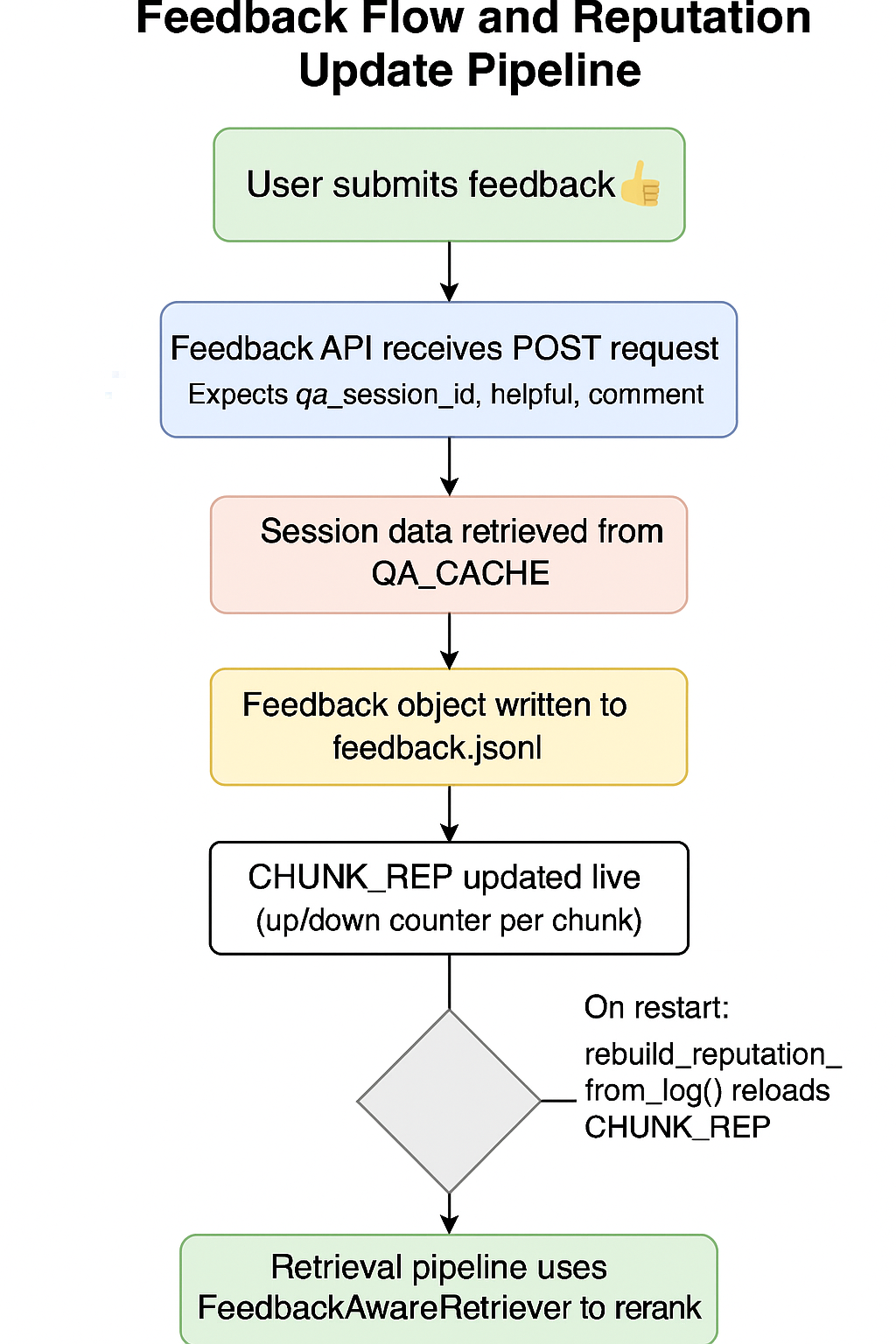
],

"answer\_excerpt": "The sigmoid function is used to transform the linear score into a probability between 0 and 1..."

}

This log not only provides an auditable trail of user feedback, but also powers a **real-time feedback-aware reranking system**. Upon startup (or refresh), the backend invokes rebuild\_reputation\_from\_log(), which parses the log file and rebuilds a global reputation dictionary (CHUNK\_REP). This dictionary maps each chunk’s unique ID to a running count of upvotes and downvotes, stored in-memory.

The below diagram illustrates the complete end-to-end **feedback flow** and how it influences **retrieval reranking** in the system.



### 5.4.1 Feedback-Aware Reranking Logic

To make practical use of feedback data, the system wraps the default retriever with a custom class called ***FeedbackAwareRetriever***. This class inherits from BaseRetriever and reorders the top-k retrieved chunks by applying a small penalty to chunks that have **more downvotes than upvotes**. Specifically:

* Chunks with negative feedback are demoted.
* Those with positive or no feedback retain their ranking.
* The magnitude of penalty is governed by a tunable hyperparameter β (default 0.2).

This ensures that the retrieval step evolves over time, pushing unhelpful chunks lower in the list while promoting context segments that have proven to be valuable to past users. The reranker respects original ranking for all unknown chunks, thereby maintaining stability for newer or rarely used contexts.

A screenshot of a computer

AI-generated content may be incorrect.

As shown above, the user is presented with an option to provide binary feedback (thumbs up/down) along with an optional text comment. This helps the system capture qualitative insights when an answer is not helpful.

A screenshot of a computer

AI-generated content may be incorrect.

Upon submitting feedback, the system acknowledges the user's input with a success message. This feedback is then logged and contributes to the chunk’s reputation for future answer reranking.

# Chapter 6 Document Summarization & Prompt-Driven Q/A Generation

This module enables educators to **automatically generate exam-ready question papers** from uploaded academic documents. By leveraging LLM-based summarization and carefully structured prompts, it transforms large textual content into meaningful and balanced sets of questions across various formats (MCQs, one-liners, fill-in-the-blanks, etc.). This pipeline ensures:

* Context-aware content summarization,
* Prompt-driven question generation,
* UI-backed control over document selection, difficulty, and type,
* Exportable output (PDF) for easy use.

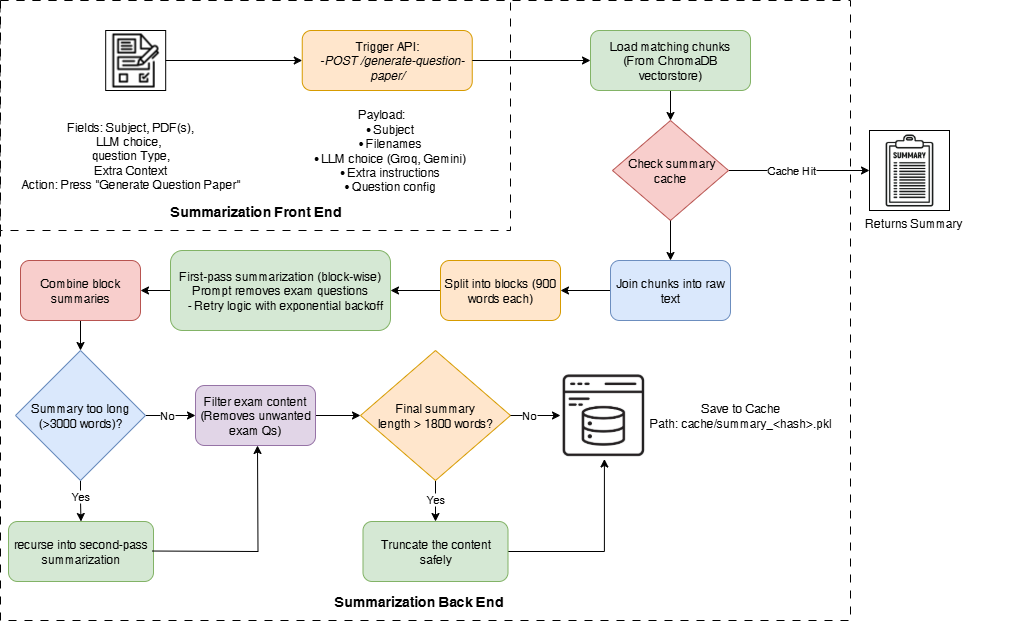
## 6.1 Summarization Pipeline: Reducing Context to Its Core

As educational handouts and academic PDFs grow in size, feeding them directly into an LLM becomes infeasible due to token limits. Not all information in these documents is equally useful either — raw content often contains redundant data, boilerplate text, or sections unrelated to assessment. To address these limitations, this system incorporates an **automated summarization pipeline**, which extracts **key educational insights** and prepares them for downstream tasks such as question generation. The summarization step also ensures that even long-form input can be compressed into a **token-efficient, meaningful summary**, enabling prompt-driven Q/A to operate effectively.

The entire pipeline spans multiple components — from frontend UI where the user selects files, all the way to recursive summarization, filtering, and result caching. Designed for **performance, stability, and reusability**, the process involves recursive LLM summarization with token budget enforcement, multi-level retries, and intelligent caching to avoid repeated computations. Furthermore, it supports various LLM backends (Groq, Gemini, Ollama) and ensures compatibility with token constraints of each. This makes it both robust and LLM-agnostic.

The final output is a **clean, exam-friendly summary**, typically under 1800 words, structured enough to allow educators or the backend engine to formulate meaningful assessments. This summary is passed on to the next stage (detailed in Section 6.2) where prompts are constructed and Q/A generation takes place.

## 6.2 End-to-End Summarization Pipeline



**Frontend Overview**

The summarization pipeline begins when the user interacts with the UI form titled **"Generate Question Paper"**. The user is required to select a **Subject**, one or more **PDF files** associated with that subject, choose an **LLM backend** (like Groq or Gemini), and optionally provide any **extra instructions** to guide the summarization and question generation process.

Once selections are made and the **"Generate Question Paper"** button is pressed, a POST request is triggered to the backend API endpoint /generate-question-paper/. The payload of this request is rich — it includes the user’s selected files, the target LLM, the configuration for how many and what kind of questions to generate, and additional user-defined context. This API call becomes the entry point for the backend pipeline, passing the entire context for processing.

**Backend Processing Pipeline**

**1. Chunk Retrieval and Cache Check**

The backend first calls get\_chunks\_by\_filenames(...) to extract all relevant chunks from ChromaDB whose metadata matches the user-selected PDFs. These chunks, essentially short segments of the educational content, are stitched together into a single string. Before doing any heavy computation, the system checks if a summary already exists in the cache using cache\_summary\_load(...). If a match is found (based on a hashed key of subject, files, LLM, and context), the summary is returned immediately, saving time and compute.

**2. Recursive Summarization Logic**

If no cached summary exists, the system begins a **deep summarization process**. First, the raw text is split into manageable blocks (≈900 words each). Each block is then passed through the llm\_summarize\_func(...) function, which communicates with the chosen LLM using a prompt that enforces strict summarization rules (e.g., no exam questions, structured output). This prompt is carefully designed and enforced with **retry logic** in case of transient failures.

After the first-pass block-wise summarization, the outputs are stitched together. If the combined result is still too long (e.g., over 3000 words), a **second recursive summarization pass** is triggered using the same logic, compressing the content even further while preserving its structure.

**3. Filtering, Final Compression, and Caching**

Once a reasonable-sized summary is generated, it passes through a filtering function filter\_summary(...) that removes any residual "exam questions" or irrelevant formatting. If the summary still exceeds the 1800-word budget, it’s **safely truncated or re-summarized** to fit within token constraints for future LLM calls. Finally, the processed summary is stored using cache\_summary\_save(...) under a uniquely hashed filename. The backend then returns the summary (and later, the generated questions) as a structured JSON response to the frontend.

**Example API Usage**

**API Endpoint:** POST /generate-question-paper/

**Request JSON:**

{

"subject": "Machine Learning",

"filenames": ["ML\_Handout\_1.pdf"],

"llm\_choice": "groq",

"question\_config": {

"total\_questions": 6,

"difficulty": "medium",

"distribution": {

"one\_liner": 2,

"true\_false": 1,

"multiple\_choice": 2,

"descriptive": 1

}

},

"extra\_context": "Focus more on regression and classification topics."

}

## 6.3 Prompt Construction and Response Generation

This section outlines the backend mechanism that transforms a **finalized summary** into a well-structured **set of academic questions**. Once summarization is complete, the system invokes a dedicated logic to construct a precise **LLM prompt** and send it to the chosen model (e.g., **Groq Llama3**, **Gemini Flash**). The model is instructed to return a **strictly validated JSON** object containing question-answer pairs, formatted according to a predefined schema. This ensures **consistency**, **validity**, and **clean parsing** for further frontend use or export (e.g., to PDF).

**Key Features of Prompt Construction Logic**

* **System Prompt Template:** Defines the role of the LLM (exam generator) and enforces a strict output format with all required fields (type, question, options, answer).
* **User Prompt:** Dynamically built based on user inputs — including total question count, difficulty, and distribution across question types.
* **Strict Output Schema:** The response is parsed and validated using QuestionPaper Pydantic schema (LangChain for Groq; Gemini schema for Google API).
* **Structured Output Mode:**

1. **For Groq**, **json\_mode** ensures the output strictly follows the schema.
2. **For Gemini**, **response\_schema** validation is used via Google’s SDK.

A white diamond with black background

AI-generated content may be incorrect.

**1. Prompt Formation**

The prompt begins with a **system message**, which instructs the LLM to act as a strict exam paper generator and follow a **hardcoded schema**. This avoids unpredictable outputs and ensures that fields like type, question, options, and answer are present and correctly formatted.

A **user message** is then assembled dynamically using:

* The user’s selected **question distribution** (e.g., 2 MCQs, 1 descriptive),
* The **difficulty level** (easy/medium/hard),
* Optional **extra context or constraints**, and
* The full **deep summary** from the previous step.

This prompt ensures that the LLM operates within a clear instructional boundary.

**2. LLM Invocation**

* If **Groq** is selected, the system uses **LangChain's ChatPromptTemplate** and binds the model to a schema using .with\_structured\_output() in json\_mode. This ensures that the LLM generates **valid Pydantic-compatible JSON** directly — eliminating the need for fragile string parsing.
* If **Gemini** is selected, the backend uses **Google's native SDK** with generate\_content(...), supplying a response\_schema to validate the structure. The model responds with a **strict JSON block**, and any format violations are caught at the SDK level.

**3. Structured Output and Return**

Once the response is received:

* If the output passes validation, it's returned as part of the final payload to the frontend (i.e., summary + questions).
* If any failure occurs (e.g., schema mismatch), a fallback error message is returned indicating the issue.

This strict mechanism improves **reliability**, **integrity**, and **user confidence** in the generated question papers.

**Sample response:**

{

"summary": "This chapter introduces fundamental concepts in supervised machine learning, covering algorithms such as linear regression, logistic regression, and decision trees. It explains the process of training models using labeled datasets, evaluating performance through metrics like accuracy and F1-score, and techniques to prevent overfitting, including regularization. The section concludes with a discussion on the importance of feature selection and preprocessing for optimal model performance……",

"questions": {

"questions": [

{

"type": "one\_liner",

"question": "What is the primary goal of supervised learning?",

"options": null,

"answer": "To learn a mapping from input to output using labeled data."

},

{

"type": "true\_false",

"question": "Decision trees cannot be used for classification tasks.",

"options": null,

"answer": "False"

},

{

"type": "fill\_blank",

"question": "\_\_\_\_\_\_ is a technique used to prevent overfitting by adding a penalty to the loss function.",

"options": null,

"answer": "Regularization"

},

{

"type": "multiple\_choice",

"question": "Which metric is commonly used to evaluate classification models?",

"options": ["MSE", "F1-score", "MAE", "R-squared"],

"answer": "F1-score"

}

]

}

}

## 6.4 Educator View: Summary and Generated Questions

The below screenshot displays the top portion of the **“Generate Question Paper”** module, where the user configures input parameters for the generation task.  
The user selects the **Subject**, **PDF(s)** from the ingested corpus, and the desired **LLM backend** (Groq or Gemini).  
  
In the “Question Setup” section, different types of questions can be toggled along with their desired **count**, and a difficulty level (Easy/Medium/Hard) is chosen.

A screenshot of a computer

AI-generated content may be incorrect.

Image X: Input Configuration & Question Setup

This section allows the user to provide **extra context or educator instructions**, such as focusing on specific chapters or topics.  
Once “Generate Question Paper” is clicked, a deep LLM-generated **summary** of the selected documents is displayed in an accordion view.  
This summary is factual, structured, and optimized to serve as a foundation for high-quality question generation, ensuring the questions are tightly aligned with the source material.

A screenshot of a computer

AI-generated content may be incorrect.

Image X: Additional Instructions & Summary Preview

Here, the generated set of **Q/A pairs** is displayed according to the configuration provided earlier.  
Questions span **multiple types** including one-liners, true/false, fill-in-the-blank, MCQs, and descriptive.  
Each question is clearly labeled and optionally includes answer options where applicable.  
The final step allows the user to **download the complete question paper as a PDF**, including both questions and answers neatly formatted.

A screenshot of a computer

AI-generated content may be incorrect.

Generated Questions Preview & Export

# Chapter 7 Evaluation & RAGAS-Based Benchmarking

Evaluation is a vital part of any LLM-based pipeline, especially for educational question-answering systems. To ensure both the correctness of the answer and the faithfulness of its supporting context, this project integrates the **RAGAS framework** (Retrieval-Augmented Generation Assessment Scores) to benchmark LLM outputs using structured metrics.

The evaluation process measures how well the retrieved context supports the final answer, how semantically aligned the generated answer is to the ground truth, and how factually grounded the result is. The RAGAS pipeline allows **single model evaluation** as well as **multi-model benchmarking**, providing detailed metric-wise comparisons across models like **Groq (Llama3)**, **Gemini**, and **Ollama**.

RAGAS metrics are categorized into the following sections:

* **Retrieval Metrics**  
  Evaluate how relevant the retrieved context is with respect to the ground truth.  
  *(e.g., Context Precision, Context Recall, Faithfulness)*
* **Nvidia Metrics**  
  Proposed by Nvidia to measure accuracy and context-groundedness in enterprise QA tasks.  
  *(e.g., Answer Accuracy, Context Relevance, Response Groundedness)*
* **Language Metrics**  
  Focus on textual similarity and generation quality.  
  *(e.g., Factual Correctness, Semantic Similarity, ROUGE Score, BLEU Score, etc.)*

Each metric category is explored in a separate section (6.2–6.4), where detailed visualizations and interpretations are provided.

## 6.1 RAGAS Evaluation Pipeline

This section outlines the entire evaluation setup, including how LLM-generated answers are evaluated against reference ground truths using the RAGAS metric suite. The backend supports both **single** and **multi-model** evaluation, returning results in structured format suitable for tables, charts, and further visualization.

**Evaluation Modes Supported**

* **Single Model Evaluation**:
  + Generates answers using the selected LLM.
  + Runs all applicable RAGAS metrics.
  + Displays per-question results with precision.
* **Multi-Model Benchmarking**:
  + Runs the same question set across multiple LLMs.
  + Compares all metrics across LLMs.
  + Shows results as bar graphs, radar charts, and per-question breakdowns.

Below is the flowchart depicting the complete RAGAS evaluation pipeline — from input to metric scoring and caching:

# Chapter 6 Bibliography / References

* Binita Saha, Utsha Saha et al. (Jan 2025) “QuIM-RAG: Advancing Retrieval-Augmented Generation with Inverted Question Matching for Enhanced QA Performance”, arXiv: 2501.02702
* M. Shah, P. Zhou, et al. (Oct. 2024), “A Comprehensive Survey of Retrieval-Augmented Generation (RAG): Evolution, Current Landscape and Future Directions” (ResearchGate)
* Praban Narak (Apr. 2024), “RAG on Complex PDF using LlamaParse, Langchain and Groq”, Medim.com blog
* Rohan Anil, Sebastian Borgeaud, et al. (Dec 2023) “Gemini: A Family of Highly Capable Multimodal Models”, arXiv: 2312.11805
* Sonal Prabhune and Donald J. Berndt (Nov 2024), “Deploying Large Language Models with Retrieval Augmented Generation”, arXiv: 2411.11895v1
* Y. Liu, Y. Wang, et al. (Dec. 2023), “Retrieval-Augmented Generation for Large Language Models” arXiv: 2312.10997
* Bo Ni, Zheyuan Liu, et al.(Feb 2025), “Towards Trustworthy Retrieval Augmented Generation for Large Language Models: A Survey”, arXiv: 2502.06872
* Boci Peng, Yun Zhu et al. (Aug. 2024), “Graph Retrieval-Augmented Generation: A Survey”, arXiv: 2408.08921
* Alec M. (Mar. 2024), “Retrieval Augmented Generation with Groq API”, Groq blog
* Franklin Lee, Tengfei Ma (June 2025), “The Budget AI Researcher and the Power of RAG Chains”, arXiv: 2506.12317v1
* Jinhyuk Lee, Feiyang Chen et al. (Mar. 2025), “Generalizable Embeddings from Gemini”, Xiv: 2503.07891
* Petko Georgiev, Ving Ian Lei, et al. (Mar 2024)**, “Gemini 1.5: Unlocking multimodal understanding across millions of tokens”,** arXiv: 2403.05530
* Wenqi Fan, Yujuan Ding et al. (Aug. 2024) “A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models” ACM digital library: doi/10.1145/3637528.3671470
* Monica Riedler, Stefan Langer (Oct. 2024)**, “Optimizing RAG with Multimodal Inputs for Industrial Applications”, arXiv: 2410.21943v1**

**Check list of items for the Final report**

Is the Cover page in proper format? Y

Is the Title page in proper format? Y

Is the Certificate from the Supervisor in proper format? Has it been signed? Y

Is Abstract included in the Report? Is it properly written? Y

Does the Table of Contents page include chapter page numbers? Y

Does the Report contain a summary of the literature survey? Y

Are the Pages numbered properly? Y

Are the Figures numbered properly? Y

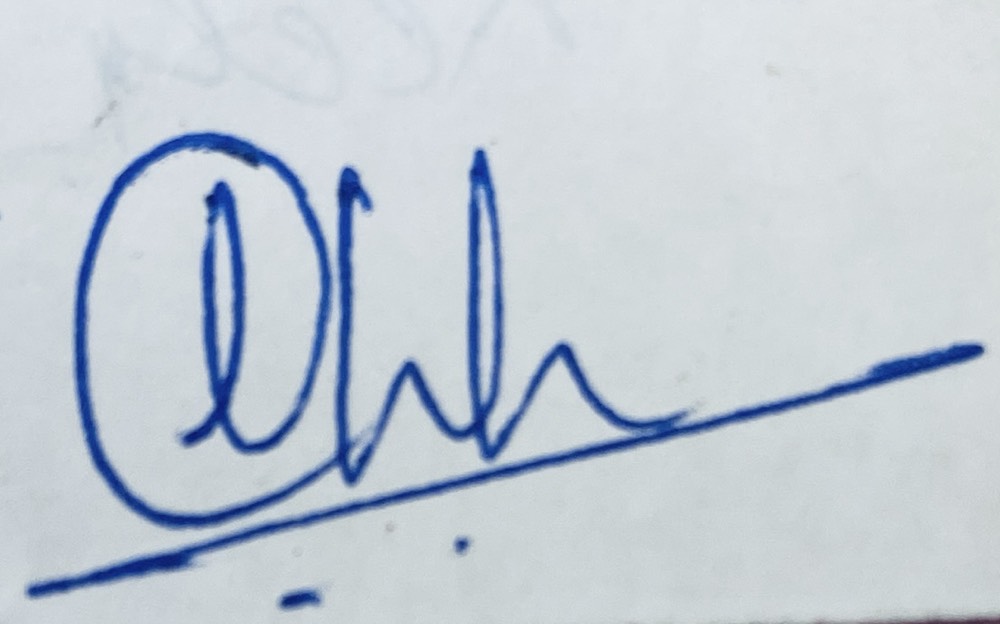
Are the Tables numbered properly? Y

Are the Captions for the Figures and Tables proper? Y

Are References/Bibliography given in the Report? Y

Have the References been cited in the Report? Y

Is the citation of References / Bibliography in proper format? Y

Verified and Signed by Supervisor Signed by Student   
  
  A close-up of a text

AI-generated content may be incorrect.