PROFORMA FOR THE APPROVAL PROJECT PROPOSAL

PNR **No.: ……** Roll no**:** _____

1. Name of the Student

   _____

2. Title of the Project

   _____

3. Name of the Guide

   _____

4. Teaching experience of the Guide _____

5. Is this your first submission? Yes ☐ No ☐

Signature of the Student Signature of the Guide

Date: …… Date: …….

Signature of the Coordinator

Date: ……

ARISTO

A Project Report
Submitted in partial fulfillment of the Requirements for the award of the Degree of

BACHELOR OF SCIENCE (INFORMATION TECHNOLOGY)

By

Kajal Solanki
Roll No: 69

Under the esteemed guidance of
Mrs. Palak Agrawat
H.O.D.

DEPARTMENT OF INFORMATION TECHNOLOGY
ST. ROCKS COLLEGE OF SCIENCE AND COMMERCE
(Affiliated to University of Mumbai)
MUMBAI, 400092 MAHARASHTRA
2024-2025

DEPARTMENT OF INFORMATION TECHNOLOGY
ST. ROCKS COLLEGE OF SCIENCE AND COMMERCE
(Affiliated to University of Mumbai)
MUMBAI, 400092 MAHARASHTRA

DEPARTMENT OF INFORMATION TECHNOLOGY

CERTIFICATE

This is to certify that the project entitled, **"ARISTO "**, is bonafied work of **KAJAL SOLANKI** bearing **Roll No: 69** submitted in partial fulfillment of the requirements for the award of degree of BACHELOR OF SCIENCE in INFORMATION TECHNOLOGY from University of Mumbai.

Internal Examiner **Coordinator**

External Examiner

Date: **College Seal**

Abstract

Aristo is an intelligent Retrieval-Augmented Generation (RAG) system designed to revolutionize the way students interact with their academic material. The system enables learners to upload their college textbooks in PDF format and query the content directly through an intuitive Streamlit interface. By integrating advanced Natural Language Processing (NLP) techniques and embedding models like all-MiniLM-L6-v2, Aristo efficiently converts textbook content into vector representations stored within a FAISS database. When a user poses a question, the system retrieves the most relevant text segments, combines them, and generates a coherent and contextually precise answer using a Large Language Model (LLM) such as OpenAI's GPT-3.5.

Unlike traditional chatbots or search engines, Aristo confines its responses strictly to the uploaded material, ensuring high accuracy, contextual relevance, and academic reliability. This targeted approach allows students to prepare for exams, write assignments, and clarify concepts without navigating multiple online sources. The system's modular architecture supports database clearing, multiple embeddings, and local deployment, making it scalable for institutional use. Ultimately, Aristo acts as a personalized academic assistant, bridging the gap between static educational resources and interactive AI-driven learning, promoting deeper comprehension, and fostering self-guided study experiences in modern education.

ACKNOWLEDGEMENT

The successful development of *Aristo* would not have been possible without the support, guidance, and contributions of many individuals and resources.

First and foremost, I would like to express my sincere gratitude to **Ms. Palak Agrawat**, whose expertise and valuable insights have been instrumental throughout the project's planning and development phases. Their constructive feedback has greatly contributed to the overall design and functionality of **Aristo**.

Finally, Special thanks go to the open-source community for providing tools, libraries, and resources that have played a key role in the technical aspects of **Aristo**. The collaborative environment fostered by the tech community has been an invaluable asset during this development journey.

Thank you to everyone who contributed to making **Aristo** a reality.

DECLARATION

I hereby declare that the project entitled, "**Aristo**" done at **St. Rocks College of Science and Commerce**, has not been in any case duplicated to submit to any other university for the award of any degree. To the best of my knowledge other than me, no one has submitted to any other university.

The project is done in partial fulfillment of the requirements for the award of degree of **BACHELOR OF SCIENCE (INFORMATION TECHNOLOGY)** to be submitted as final semester project as part of our curriculum.

Kajal Solanki

Table of Contents

Introduction

In today's rapidly evolving educational landscape, students face an overwhelming amount of textual information across multiple subjects and disciplines. With academic resources increasingly shifting from printed textbooks to digital formats, there arises a need for intelligent systems that can simplify the process of extracting relevant knowledge from these dense, information-rich materials. The Aristo project is designed to bridge this gap by integrating Retrieval-Augmented Generation (RAG) technology with modern natural language processing (NLP) capabilities.

Aristo is a personal academic assistant that enables students to upload their college textbooks in PDF format and receive accurate, contextually grounded answers to their questions — answers that are strictly derived from the provided material. This ensures that students obtain information that is both relevant and confined to the syllabus, making the tool particularly valuable for exam preparation, assignment writing, and conceptual understanding.

The system leverages Python as its core programming language, integrating advanced frameworks such as Streamlit for the web interface, PyMuPDF (fitz) for text extraction, and FAISS for efficient vector-based similarity search. The text embeddings are generated using MiniLM (all-MiniLM-L6-v2), a lightweight yet powerful transformer model from Hugging Face. Initially, Aristo uses OpenAI's GPT-3.5 as the answering engine for response generation, with future enhancements planned to incorporate TinyLLM or other locally hosted large language models for offline usage.

By combining these components, Aristo transforms static PDF textbooks into interactive learning tools. It allows users to upload, query, and clear vector databases dynamically — giving full control over what content the system references. This functionality not only personalizes the learning experience but also reinforces student engagement by offering precise, textbook-based answers instead of generalized, internet-sourced information.

Aristo stands as a powerful demonstration of how artificial intelligence can enhance academic productivity, deepen subject understanding, and ultimately reshape the future of digital learning.

1.1 Background

Traditional learning methods often rely heavily on static resources such as printed textbooks, lecture notes, and online references. While textbooks remain an essential part of academic education, they are inherently limited by their non-interactive nature. Students frequently struggle to locate specific concepts, definitions, or explanations within long chapters, leading to inefficient study patterns and increased cognitive load. Moreover, conventional search engines or AI chatbots trained on broad internet data often provide inaccurate or irrelevant information that diverges from prescribed course material.

To address this challenge, Retrieval-Augmented Generation (RAG) has emerged as a cutting-edge approach in the field of natural language processing. Unlike standard large language models that rely solely on pre-trained data, RAG systems combine information retrieval techniques with generative AI models. They first search for relevant textual passages from a user-provided corpus (such as uploaded PDFs) and then generate human-like responses based on that specific context. This ensures accuracy, contextual precision, and domain relevance.

In this context, Aristo has been conceptualized as a RAG-based academic assistant that empowers students to upload and interact directly with their textbooks. Instead of passively reading through pages, students can ask natural language questions — such as "Explain the difference between primary and secondary memory" or "Summarize Chapter 3: Software Engineering Principles" — and receive contextually relevant responses drawn from the provided PDFs.

From a technological perspective, Aristo is grounded in the transformer architecture and modern vector search methodologies. Using MiniLM (all-MiniLM-L6-v2) embeddings ensures that the system remains lightweight and suitable for local computation, while FAISS (Facebook AI Similarity Search) enables rapid retrieval of semantically similar text chunks. The inclusion of Streamlit as the user interface framework simplifies the user interaction process, providing an accessible dashboard where users can upload PDFs, view responses, and manage their vector databases effortlessly.

The project's evolution also aligns with the growing trend of AI-driven educational support systems. Similar to how virtual assistants help in productivity, Aristo aims to become a virtual study companion, capable of understanding academic language, retrieving precise content, and generating human-like explanations that enhance student comprehension.

In short, the background of Aristo lies at the intersection of education technology (EdTech), artificial intelligence, and information retrieval systems, combining these domains to make self-study more interactive, efficient, and accessible.

1.2 Objectives

The primary objective of Aristo is to create a Retrieval-Augmented Generation (RAG) system that transforms traditional academic study materials into an interactive, AI-driven learning experience. To achieve this vision, the project defines a comprehensive set of objectives, both technical and functional, as outlined below:

1.2.1 Core Objectives

a. To design and implement a RAG-based question-answering system that restricts its responses to the contents of user-uploaded PDFs, ensuring accuracy and academic integrity.

b. To enable seamless PDF uploads and text extraction using PyMuPDF, converting diverse textbook formats into structured textual data suitable for processing.

c. To perform intelligent text chunking and embedding, ensuring that the model can efficiently capture contextual meaning from large bodies of text.

d. To store and manage embeddings using FAISS, allowing for fast and efficient retrieval of semantically similar text chunks when responding to user queries.

e. To integrate a large language model (LLM) such as GPT-3.5 initially, and later transition toward a local LLM (TinyLLM), for privacy-friendly and cost-effective response generation.

f. To build an interactive and user-friendly Streamlit interface where students can upload files, ask questions, manage their data, and view AI-generated answers.

1.2.2 Educational Objectives

a. To assist students in exam preparation by enabling them to ask questions from uploaded textbooks and

receive precise, syllabus-specific explanations.

b. To simplify academic research and assignment writing by allowing students to generate accurate and well-structured answers derived directly from their course materials.

c. To promote active learning and comprehension, encouraging students to interact with their textbooks rather than passively reading.

d. To enhance accessibility by offering an easy-to-use digital study tool that can operate locally, reducing dependency on internet access.

1.2.3 Development Objectives

a. To establish a modular and scalable architecture, ensuring that the system can later integrate with additional features such as note generation, quiz creation, or personalized study recommendations.

b. To optimize performance and memory usage, especially when handling large textbook PDFs with thousands of pages.

c. To provide database management options that let users clear, retain, or merge embeddings from multiple documents for broader or more focused responses.

d. To ensure security, data privacy, and transparency, especially when dealing with user-uploaded academic materials.

By meeting these objectives, Aristo aims not only to serve as a practical AI study tool but also as a prototype for the next generation of context-aware educational assistants.

1.3 Scope of the Project

The scope of Aristo encompasses both its technical boundaries and functional capabilities. It defines what the system currently achieves and outlines potential areas for future enhancement.

1.3.1 Functional Scope

Aristo's functionality revolves around its ability to ingest, process, and query academic documents:

a. PDF Upload & Extraction: Users can upload one or multiple textbooks in PDF format. The system extracts clean textual data using PyMuPDF, ensuring accuracy even with scanned or structured documents.

b. Text Chunking & Embedding: Extracted text is divided into manageable chunks that are embedded via MiniLM (all-MiniLM-L6-v2), producing semantic representations suitable for similarity matching.

c. Vector Storage via FAISS: The embeddings are stored and indexed in a FAISS vector database, enabling high-speed retrieval during query processing.

d. Query Handling & Response Generation: When a user asks a question, the query is embedded and compared with stored vectors. The top matching passages are combined to form a context, which is then passed to the LLM (e.g., GPT-3.5 or TinyLLM) to generate a final answer.

e. Streamlit Interface: The user interface allows seamless interaction — file uploads, queries, clearing or retaining embeddings, and viewing conversational outputs.

1.3.2 Technical Scope

a. The system currently uses Python as the development language, leveraging key AI and data libraries such as FAISS, PyMuPDF, Transformers, and Streamlit.

b. It supports local embedding computation and cloud-based generation (via OpenAI API), ensuring flexibility for both online and offline use cases.

c. The architecture is modular, allowing future replacement or enhancement of components such as the embedding model, database system, or language model.

1.3.3 Educational and Research Scope

a. Aristo can be used by students, teachers, and researchers across disciplines to quickly extract meaningful insights from academic material.

b. It supports exam preparation, concept clarification, assignment writing, and research assistance, making it a valuable academic tool.

c. Educational institutions could deploy Aristo internally, allowing entire classrooms to upload courseware and interact with it intelligently.

1.3.4 Future Expansion Scope

a. While the current version focuses on RAG-based textbook question answering, future iterations could include:

b. Integration of local LLMs for completely offline use.

c. Support for multiple file types (DOCX, PPT, TXT, etc.).

d. Addition of summarization, note-making, and flashcard generation modules.

e. User authentication and persistent memory for personalized learning experiences.

f. Enhanced analytics and performance tracking, showing students which chapters or topics, they interact with most.

In summary, the scope of Aristo is both broad and flexible. It lays the foundation for a new paradigm in AI-assisted education, one that merges accessibility, contextual precision, and interactivity into a single intelligent study platform.

System Analysis

2.1 Existing System

Overview

Before Aristo, the primary ways students interact with textbooks and study material are manual reading, note-taking, keyword searching in PDFs, web searches, and generic question-answering chatbots. Each of these approaches has strengths but also significant limitations when the goal is accurate, syllabus-constrained answers for exam preparation or assignment work.

2.1.1 Manual Study (Traditional)

Process: Read textbook chapters, create notes, highlight, and re-read.

Strengths: Deep engagement with material, encourages retention when done well.

Limitations: Time-consuming; inefficient for locating specific concepts; requires strong self-discipline; no immediate, targeted clarifications.

2.1.2 PDF Search & Indexing Tools (e.g., Adobe Reader Search)

Process: Use built-in text search by keywords or phrases.

Strengths: Quick to locate explicit words/phrases; useful for fact lookup.

Limitations: Keyword matching is brittle (synonyms/fuzzy queries fail); lacks semantic search; difficult to synthesize multiple passages; unhelpful for conceptual or summarized answers.

2.1.3 Online Search Engines (Google, Bing)

Process: Enter queries on web search and glean information from web pages.

Strengths: Vast knowledge, quick results, broad perspectives.

Limitations: Results not constrained to course material; possible misinformation; time lost triaging sources; inconsistent alignment with syllabus and textbook wording.

2.1.4 Generic Chatbots / LLMs

Process: Ask general LLMs (e.g., ChatGPT) questions and get generated answers based on pretraining.

Strengths: Natural language answers, explanatory capabilities, synthesis of knowledge.

Limitations: Hallucinations; answers may not be grounded in the user's course content; no guarantee of textbook alignment or citation; privacy concerns when uploading proprietary course material to third-party platforms.

2.1.5 Learning Management Systems (LMS) / e-Readers

Process: Some institutions provide PDF viewers, highlight features, and annotation tools.

Strengths: Centralized repository for course material; teacher-provided resources.

Limitations: Mostly read-only; lack intelligent query answering or contextual summarization constrained to selected content.

2.1.6 Existing RAG / PDF-QA Tools (Emerging Tools)

Process: Standalone implementations exist that combine embeddings with LLMs for document QA.

Strengths: Directly answer from a corpus; cite sources; faster than manual reading.

Limitations: Many such tools are either cloud-only, lack a simple UI for students, or are not optimized for educational workflows (e.g., lacking ability to clear vectors, manage multiple textbooks, or operate locally for privacy and offline use).

Gap Analysis

a. Scope Control: Most tools either answer from general knowledge or require complex configuration to restrict to a document. Students need an easy, reliable way to ensure answers come only from their specific textbooks.

b. Simplicity & Accessibility: Existing RAG prototypes often require developer expertise or cloud dependencies; students prefer an easy, UI-based flow.

c. Privacy & Offline Use: Many systems depend on cloud LLMs which raises privacy concerns and cost; local-first options are limited.

d. Educational Features: Absent features include textbook-scoped QA, ability to clear/merge embeddings, and student-centric UI elements (e.g., question history, exportable summaries).

e. Performance & Cost: Cloud-based RAG chains incur API costs and latency; local embeddings + local LLMs can be faster and cheaper at scale but require careful tech selection.

Conclusion: The existing ecosystem does not fully satisfy the specific educational need Aristo targets: an easy-to-use, textbook-scoped, privacy-respecting, efficient QA system with explicit controls for vector DB management and future offline LLM capability.

2.2 Proposed System:

System Overview

Aristo is proposed as a modular RAG system that enables students to upload PDF textbooks, automatically extract and embed textual content, and then answer user queries using only that uploaded content as context. It is designed to be usable both locally and with optional cloud components. The system will be built in Python and delivered through a Streamlit web interface for accessibility and rapid iteration.

High-Level Architecture

1. Streamlit UI: File upload, query window, settings (k, chunk size, clear DB, retain DB), session/history, and admin functions.

2. PDF Processing Layer (PyMuPDF): Reliable extraction of text from PDFs, including handling of multi-column layouts and basic pre-processing (cleaning, whitespace normalization).

3. Chunking Module: Splits extracted text into chunks using semantic-aware strategies (sliding window, overlap, chapter/section boundaries).

4. Embedding Module (all-MiniLM-L6-v2): Local embedding generation via Hugging Face Transformers for each chunk.

5. Vector Store (FAISS): Efficient indexing, storage, and retrieval of chunk embeddings with metadata

(source file, page number, chunk id).

6. Query Processor: Accepts a natural language query; generates an embedding; retrieves top-k similar chunks.

7. Prompt Builder: Constructs a context prompt by combining retrieved chunks plus user query and instructions (system/prompt engineering) for the LLM.

8. LLM Integration: Initially OpenAI GPT-3.5 via API for response generation; later optional TinyLLM or other local LLM for offline operation.

9. Response Renderer: Shows the generated answer, includes citations (which pages/chunks used), and optionally provides highlighted text excerpts.

10. Persistence & Management: Options to persist FAISS index across sessions, clear DB, upload multiple textbooks, and track user query history.

Key Functionalities

1. Upload & Process PDFs: Single or batch upload; automatic text extraction and chunking.

2. Local Embedding Generation: Use of MiniLM locally to avoid sending raw text to the cloud and reduce costs.

3. FAISS-based Retrieval: Fast nearest-neighbor retrieval supporting millions of embeddings when scaled.

4. Configurable Retrieval Parameters: k (top-k), chunk overlap, chunk size, vector distance metric (cosine).

5. Prompt Templates & Safety: Pre-defined templates to ensure responses rely on provided text; reduce hallucinations by instructing LLM clearly to reference only given context.

6. DB Management: Clear embeddings, add new textbooks (merge vectors), or create separate named collections for different subjects or semesters.

7. Export & Logs: Export question-answer pairs, citation logs, and save sessions for revision.

8. Admin Features: System stats (index size), resource utilization, and settings for local vs cloud LLM usage.

Design Principles

1. Document-Grounded Responses: All answers must be traceable to the uploaded textbooks, with clear citations (page and chunk id).

2. User Control & Transparency: Users decide whether to retain or purge embeddings; system reports which passages were used.

3. Modularity: Each component (embedding, retrieval, LLM) can be swapped with minimal changes—important for future migration to local LLMs.

4. Performance & Cost Balance: Local embeddings (MiniLM) + FAISS for retrieval; LLMs used only for conditional generation to limit cloud API expense.

5. Privacy: Default local-first behavior; when cloud LLM is used, minimal context is sent, and the user is informed.

Workflows

1. Initial Upload & Indexing: Student uploads PDF → text extraction → chunk & embed → store in FAISS. Option to name collection or associate metadata (subject, semester).

2. Query Flow: User enters query → system embeds query → FAISS retrieves top-k chunks → prompt builder constructs prompt → call to LLM → result displayed with citations and optional raw supporting text.

3. DB Lifecycle: User can keep the index for future sessions, clear it, or merge new uploads into an existing collection. Option to create "exam mode" by temporarily clearing other textbooks.

4. Offline Mode: When local LLM is available (TinyLLM), the same flow will run fully offline for privacy and cost savings.

Expected Benefits Over Existing Systems

1. Focused answers restricted to the exact uploaded textbooks → better exam relevancy.

2. Faster, cheaper retrieval due to local embeddings and FAISS.

3. Transparency and citation make results verifiable.

4. Student-friendly Streamlit UI reduces technical friction.

5. Offline capability ensures privacy and accessibility in low-connectivity environments.

2.3 Requirement Analysis

2.3.1. Functional Requirements

1. User Management & Session

- Allow users to start a session without login (optional later: user accounts).
- Maintain session-level history of queries (local storage or server-side).

2. File Upload & Management

- Accept PDF uploads via Streamlit UI (single/multiple).
- Display metadata (filename, number of pages, size).
- Provide ability to remove uploaded files and associated embeddings.
- Support named collections (e.g., "DSA_Sem3") so users can switch between sets of PDFs.

3. PDF Processing

- Extract text om PDFs using PyMuPDF, retaining page references.
- Handle common PDF layout issues (multi-column, headers/footers) with heuristics.

- Provide basic OCR fallback or guidance for scanned PDFs (optional integration with Tesseract in advanced release).

4. Chunking & Preprocessing

- Split text into chunks using size and overlap parameters (configurable).
- Preserve chunk metadata (document id, page range, character offsets).
- Implement stopword cleaning, whitespace normalization, and simple de-duplication.

5. Embedding & Indexing

- Use Hugging Face's all-MiniLM-L6-v2 to generate embeddings locally.
- Normalize embeddings (L2 or cosine-ready).
- Add embeddings to a FAISS index and persist index to disk.
- Support multiple FAISS indexes (per collection) for management.

6. Query Processing & Retrieval

- Accept ee-text queries om user.
- Embed queries and retrieve top-k chunks om FAISS.
- Highlight matching chunks and show their source pages.

7. Prompt Building & LLM Interaction

- Construct constrained prompts that include retrieved passages and explicit instruction to "answer only om the given text."
- Send prompts to OpenAI GPT-3.5 initially (with API key input) and return the generated answer.
- Provide configuration for switching to a local LLM (TinyLLM) in future iterations.
- Optionally show the full prompt that was sent (for transparency/debugging).

8. Response Presentation

- Display the answer in Streamlit with citation blocks referencing supporting chunks (document, page, chunk id).
- Allow the user to request the supporting text excerpts or view the original PDF page.

9. Vector DB Lifecycle & Controls

- Option to clear the FAISS index (per collection or globally).
- Option to merge embeddings om new uploads into an existing collection.

- Provide an “exam mode” that temporarily isolates a single book’s embedding set for focused QA.

10. Export & Logging

- Export Q&A transcript as text or markdown.
- Keep logs of which chunks were used for audits and revision.

11. Admin & Diagnostics

- Show index size, number of vectors, storage usage.
- Provide simple diagnostics (embedding speed, retrieval latency).

2.3.2 Non-Functional Requirements

Non-functional requirements are critical to ensuring that the system performs efficiently and remains reliable over time.

1. Performance

- Embedding generation should be efficient; for a standard laptop, embedding throughput should be at least several hundred chunks/hour depending on hardware.
- Retrieval latency for a query should be sub-second for small indices (thousands of chunks) and <200–500ms for indices up to several hundred thousand vectors on capable hardware.

2. Scalability

- The architecture should scale horizontally for enterprise deployment (index sharding, persisted storage).
- Support index persistence and loading/unloading indexes to manage memory.

3. Accuracy & Reliability

- Retrieval should be semantically relevant — measured via offline evaluation metrics (recall@k).
- Ensure prompt engineering reduces hallucinations; include fallback for low-confidence retrievals.

4. Security & Privacy

- Default local-first behavior; when cloud LLMs are used, the system must inform user about data transmitted.
- Option to anonymize metadata and not store identifiable information.
- Secure storage of API keys (local config files with permission warnings).

5. Usability

- Easy Streamlit-based UI requiring minimal setup for students.
- Clear instructions and error messages for common issues (unsupported PDFs, missing fonts).

6. Maintainability

- Modular codebase; each component (chunking, embedding, FAISS management, LLM interface) should have clear interfaces and unit tests.
- Documentation, sample scripts, and reproducible environment setup (requirements file or conda environment).

7. Cost-effectiveness

- Local embeddings reduce cloud costs; allow users to use ee tiers or low-cost options for LLM calls.
- Provide estimation of expected API usage and cost when cloud LLM is enabled.

2.3.3 Hardware Requirements

Minimum, recommended, and scalable options listed.

1. Minimum (for small-scale, development, or demo)

Component	Specifications
CPU	4-core CPU (x86_64), e.g., Intel i5 or equivalent.
RAM	8 GB (but may be limiting for large PDFs or many embeddings).
Storage	50 GB HDD/SSD (fast SSD preferred for FAISS persistence).
GPU	Not required for MiniLM embeddings; optional for local LLM inference.
Network	Internet required only for cloud LLM usage and initial package downloads.

2. Recommended (for comfortable use and moderate datasets)

Component	Specifications
CPU	6–8 core CPU (Intel i7 / Ryzen 7).
RAM	16–32 GB (helps with loading models & FAISS indexes).
Storage	250+ GB SSD (faster read/write for indexes and intermediate files).
GPU	NVIDIA GTX 1660 / RTX 2060 or better for faster embedding or local LLM runs.
Network	Reliable broadband when cloud LLM used.

3. High-End / Production (for institutional deployment)

Component	Specifications
CPU	16+ cores.
RAM	64+ GB
Storage	NVMe SSD 1TB+
GPU	NVIDIA A100 / V100 or multiple GPUs for local LLM serving at scale.
Network	High bandwidth and low latency to support concurrent users.

2.3.4 Software Requirements

Core Software & Libraries

1. Operating System: Linux (Ubuntu 20.04+ recommended), macOS (Monterey+), Windows Subsystem for Linux (WSL) supported.
2. Python: 3.10+ (use virtualenv/conda).
3. Streamlit: Latest stable (for UI).
4. PyMuPDF (fitz): For PDF text extraction.
5. Transformers & Tokenizers (Hugging Face): For MiniLM embeddings.
6. SentenceTransformers: (makes MiniLM usage easier).
7. FAISS (faiss-cpu or faiss-gpu): For vector indexing and retrieval.
8. OpenAI Python SDK: For GPT-3.5 interactions (optional).
9. TinyLLM runtime or equivalent: For local LLM deployment (future).
10. Numpy / Scipy / pandas: Utilities and data handling.
11. Torch (PyTorch): Required for transformer models and some local LLMs.
12. Tesseract OCR (Optional): For scanned PDFs requiring OCR.
13. Docker (optional): For reproducible deployment; containerize the stack.

Development & Tools

1. Git: Source control.
2. VS Code / PyCharm: IDE.
3. Make / Scripts: Build and environment scripts.
4. Testing: pytest or similar frameworks.

2.3.5 Selection of Technology and Justification

This section explains the choices made for technologies and components in Aristo and justifies each decision in the context of performance, cost, maintainability, and educational suitability.

1. **Programming Language: Python**

Justification:

Python is the de-facto standard for machine learning and NLP due to rich ecosystem (Transformers, PyTorch, FAISS bindings).

Rapid prototyping is possible, and many educational institutions are already familiar with Python.

Streamlit is a Python-native library, enabling rapid UI development.

2. **UI Framework: Streamlit**

Justification:

a. Streamlit allows building a functional, interactive web interface with minimal boilerplate — ideal for prototypes and MVPs targeted at students.

b. Provides instant hot-reload and easy deployment options (Streamlit sharing, local host).

c. Integrates well with Python ML components; embedding results and logs can be displayed easily.

d. Good trade-off between ease-of-use and functionality; while not a full-fledged front-end framework, it's perfect for an educational tool like Aristo where development speed and simplicity matter.

3. **PDF Processing: PyMuPDF (fitz)**

Justification:

a. PyMuPDF offers reliable and high-quality text extraction from PDFs and good page-level metadata.

b. It handles many PDF quirks, is fast, and has a Pythonic API.

c. Alternative libraries (pdfminer, PyPDF2) have limitations in layout handling and performance; PyMuPDF is a balanced choice for multi-column or complex textbooks.

d. Provides direct access to page images if needed for future features like highlighted page previews.

4. **Embedding Model: all-MiniLM-L6-v2 (MiniLM)**

Justification:

a. MiniLM is small, fast, and produces strong semantic embeddings suitable for semantic search tasks.

b. Low resource requirements make it feasible to run locally on student machines or modest servers.

c. The tradeoff between quality and performance is excellent for academic text retrieval where latency and cost matter.

d. Using Hugging Face / SentenceTransformers makes integration straightforward and stable.

e. Enables offline-first operation without continuous cloud costs.

5. **Vector Database: FAISS**

Justification:

a. FAISS is a mature, high-performance library for nearest neighbor search and indexing.

b. Supports many index types (IVF, HNSW, PQ) enabling scalability from small to very large corpora.

c. Has GPU support for production-level speedups while also providing CPU-friendly options for local setups.

d. Wide community adoption and proven performance for semantic search use-cases.

6. **Large Language Model: OpenAI GPT-3.5 (initial) → TinyLLM (future)**

Justification for GPT-3.5 (initial):

a. GPT-3.5 offers high-quality text generation and is accessible via a stable API, ensuring good baseline outputs for early-stage evaluation.

b. Allows rapid development without the complexity of hosting an LLM.

c. Useful for early user testing, prompt design, and UX validation.

Justification for TinyLLM (future/local):

a. TinyLLM (or other efficient local LLMs) enables fully offline operation, better privacy, and lower long-term cost.

b. Local LLMs reduce reliance on third-party services and provide customizable control over inference behavior and prompt engineering.

c. Transition plan: design modular LLM interface so switching from OpenAI to local is straightforward.

7. **Frameworks & Libraries: Transformers, SentenceTransformers, PyTorch**

Justification:

a. Transformers and SentenceTransformers provide pre-trained models and utilities for embedding text conveniently.

b. PyTorch is a common runtime for these models, offering GPU acceleration on supported hardware.

c. These libraries are well-maintained and have community-proven implementations for embedding tasks.

8. **Optional OCR: Tesseract**

Justification:

a. Some textbooks may be scanned or image-only PDFs; Tesseract offers a free OCR solution to convert images to text.

b. While Tesseract isn't perfect, for many scanned pages it will provide workable text that can then be chunked and embedded.

c. Offered as optional because OCR can be resource-intensive and may require post-correction.

9. **Data Storage & Persistence: Local Disk / Optional Cloud Storage**

Justification:

a. Persisting FAISS indexes and embeddings to disk allows reloading indexes between sessions without re-embedding.

b. For institutions, optional cloud storage (S3, etc.) can be offered for centralized storage and multi-user access.

c. Local-first approach places privacy and control with users; cloud options are optional for scale.

10. **DevOps & Packaging: Docker, Virtual Environments**

Justification:

a. Docker ensures reproducible environments for deployment on institutional servers.

b. Virtual environments (venv/conda) are adequate for local development and student use.

c. Provide both options for flexibility.

d. Monitoring, Logging & Security

e. Use Python logging for event tracing and optional Sentry for error tracking in deployed instances.

f. Secure API keys stored locally (with clear user instructions) and not embedded in source code.

g. Optionally support encrypted indices or access control for shared deployments.

Closing Notes on System Analysis

This system analysis aims to provide a comprehensive foundation for designing and implementing Aristo. It highlights the limitations of existing approaches, describes a clear, modular proposed system, and lists exhaustive functional and non-functional requirements that will guide development, testing, and deployment. The technology selections are made to optimize for:

- Educational relevance (answers limited to uploaded textbooks),
- Performance (local embeddings + FAISS retrieval),
- Privacy (local-first behavior and optional offline LLMs),
- Usability (Streamlit-based UI),
- Scalability & Maintainability (modular design enabling replacement of components).

System Design

3.1 Module Division

The Aristo system is divided into several interdependent modules, each performing a distinct function to ensure smooth data flow from PDF upload → Text extraction → Embedding → Retrieval → Response Generation. Each module plays a critical role in maintaining modularity, scalability, and accuracy of the RAG pipeline. Below is a detailed explanation of every major module:

1 User Interface Module (Streamlit Frontend)
This is the primary interaction point between the student and the Aristo system. It provides a clean, intuitive, and minimal interface for uploading PDFs, entering queries, and viewing AI-generated responses.
Responsibilities:

Display a dashboard with the following sections:

PDF Upload Panel for adding one or more textbooks or notes.
Chat or Query Input Box for student questions.
Response Display Section for showing model outputs.
Database Management Controls to clear or maintain FAISS embeddings.
Handles file selection, progress indicators, and streaming of responses in real time.
Ensures interactive user experience through Streamlit widgets and layout.

Key Advantages:

Eliminates technical complexity for students.
Real-time updates make query processing transparent.
Portable and web-accessible interface.

2 PDF Processing Module
This module is responsible for extracting clean, structured text data from the uploaded PDFs using PyMuPDF (fitz).

Responsibilities:

Read and parse multiple PDFs.
Extract textual content page-wise.
Handle non-textual data gracefully (e.g., images, tables, diagrams).
Clean and preprocess text by removing unwanted symbols, headers, or spacing.

Process Flow:

Load PDF using PyMuPDF.
Iterate through all pages and extract text.
Store extracted text temporarily in memory or local storage.
Send processed text to the chunking module.

Importance:
Without this module, the LLM cannot understand PDF content. The text extraction ensures raw educational material is converted into usable text segments.

3 Text Chunking Module
The goal of this module is to divide large bodies of extracted text into smaller, semantically meaningful chunks to make them suitable for embedding and retrieval.

Responsibilities:

Segment text into overlapping chunks of configurable size (e.g., 500–700 tokens).
Maintain semantic coherence between sentences.
Assign unique IDs or references to each chunk for FAISS mapping.

Importance:
Efficient chunking ensures that the LLM receives contextually accurate passages when queried, improving precision and relevance.

4 Embedding Module
This module is the backbone of the RAG process. It converts text chunks into vector embeddings using the MiniLM model (all-MiniLM-L6-v2).

Responsibilities:

Load the embedding model locally (via Hugging Face Transformers).
Convert all chunks into 384-dimensional vector representations.
Store embeddings along with their corresponding text IDs.

Importance:
Embeddings are numerical fingerprints that enable FAISS to compare similarity between queries and text chunks efficiently.

Performance Optimization:

Uses batch embedding for large texts.
Caches embeddings for repeated PDF uploads.

5 Vector Database Module (FAISS)
The FAISS (Facebook AI Similarity Search) database stores and retrieves embeddings based on similarity scores.

Responsibilities:

Store all generated embeddings in an optimized FAISS index.
Perform vector similarity search (cosine or L2 distance).
Retrieve top-k most relevant chunks in response to user queries.
Handle database clearing or multi-PDF indexing as per user control.

Features:
Supports incremental addition of new embeddings.
Allows complete database reset to restrict context to new textbooks.
Provides lightning-fast retrieval even for thousands of chunks.

Importance:
This module ensures that responses remain contextually bounded within the provided study materials, maintaining academic relevance.

6 Query Processing and Retrieval Module
This module handles all user interactions after a query is entered.

Responsibilities:

Accept a user query and embed it using the same MiniLM model.
Perform FAISS similarity search to identify top k matches.
Combine retrieved text chunks into a contextual reference prompt.
Forward the combined context to the LLM interface module.

Importance:
This ensures that only the most relevant textbook passages are supplied to the LLM, thereby improving accuracy and minimizing hallucination.

1.7 LLM Response Generation Module
This module manages the interaction with the OpenAI GPT-3.5 API (and later with a TinyLLM local model).

Responsibilities:

Format the combined context and query into a prompt.
Send the request to the chosen model (cloud or local).
Receive, clean, and format the model-generated answer.
Display the result on the Streamlit interface.

Importance:
This is the module that delivers intelligent, human-like explanations, interpretations, or summaries based solely on the user's uploaded material.

8 Database Management and Control Module
This is an administrative feature accessible from the dashboard.

Responsibilities:

Allow users to clear embeddings from the FAISS index.
Maintain multiple embedding sets for different subjects or textbooks.
Handle versioning or naming of embeddings for identification.

Importance:
Provides flexibility to users—whether they want a single textbook focus or a multi-subject context space.

9 Logging and Performance Monitoring Module
Handles system-level tracking for debugging, usage, and optimization.

Responsibilities:

Log all actions (PDF uploads, embeddings, query responses, FAISS operations).
Record time taken for each process.
Provide metrics like retrieval speed, model latency, and embedding efficiency.

Importance:
Helps developers optimize system performance and troubleshoot issues.

3.2 Data Dictionary

The Data Dictionary defines all data entities, attributes, and their relationships within the Aristo system.
It represents how data moves between components, how it's stored, and what metadata accompanies it.

Entity Name	Attribute Name	Data Type	Description
User_Input	query_id	Integer	Unique identifier for each query submitted by a user
	user_query	Text	The natural language question entered by the student
	timestamp	Datetime	Time when the query was submitted
Uploaded_PDF	pdf_id	Integer	Unique ID assigned to each uploaded PDF
	pdf_name	String	File name of the uploaded PDF
	pdf_path	String	File storage path in the local system
	upload_time	Datetime	Timestamp when the file was uploaded
Extracted_Text	text_id	Integer	Identifier for each extracted section of text
	pdf_id	Integer (FK)	Links extracted text to its source PDF
	content	Text	The raw textual data extracted from PDF pages
Text_Chunks	chunk_id	Integer	Identifier for each text chunk
	text_id	Integer (FK)	Links chunk to original extracted text
	chunk_text	Text	Actual segment of text
	chunk_index	Integer	Order of the chunk in document
	embedding_vector	Array (Float)	Vector representation generated by MiniLM
FAISS_Index	index_id	Integer	Identifier for FAISS record
	chunk_id	Integer (FK)	Link to associated text chunk
	vector_data	Binary	Encoded embedding data stored for retrieval
	similarity_score	Float	Calculated similarity between query and text chunk
Model_Response	response_id	Integer	Unique response identifier
	query_id	Integer (FK)	Corresponds to user query
	generated_answer	Text	Output text from the LLM
	response_time	Float	Time taken to process and generate response
	model_used	String	Indicates whether GPT-3.5 or TinyLLM was used
System_Log	log_id	Integer	Log entry identifier
	activity_type	String	Type of activity (upload, embed, query, clear)
	details	Text	Description of the event
	timestamp	Datetime	Log time

3.3 E-R Diagram

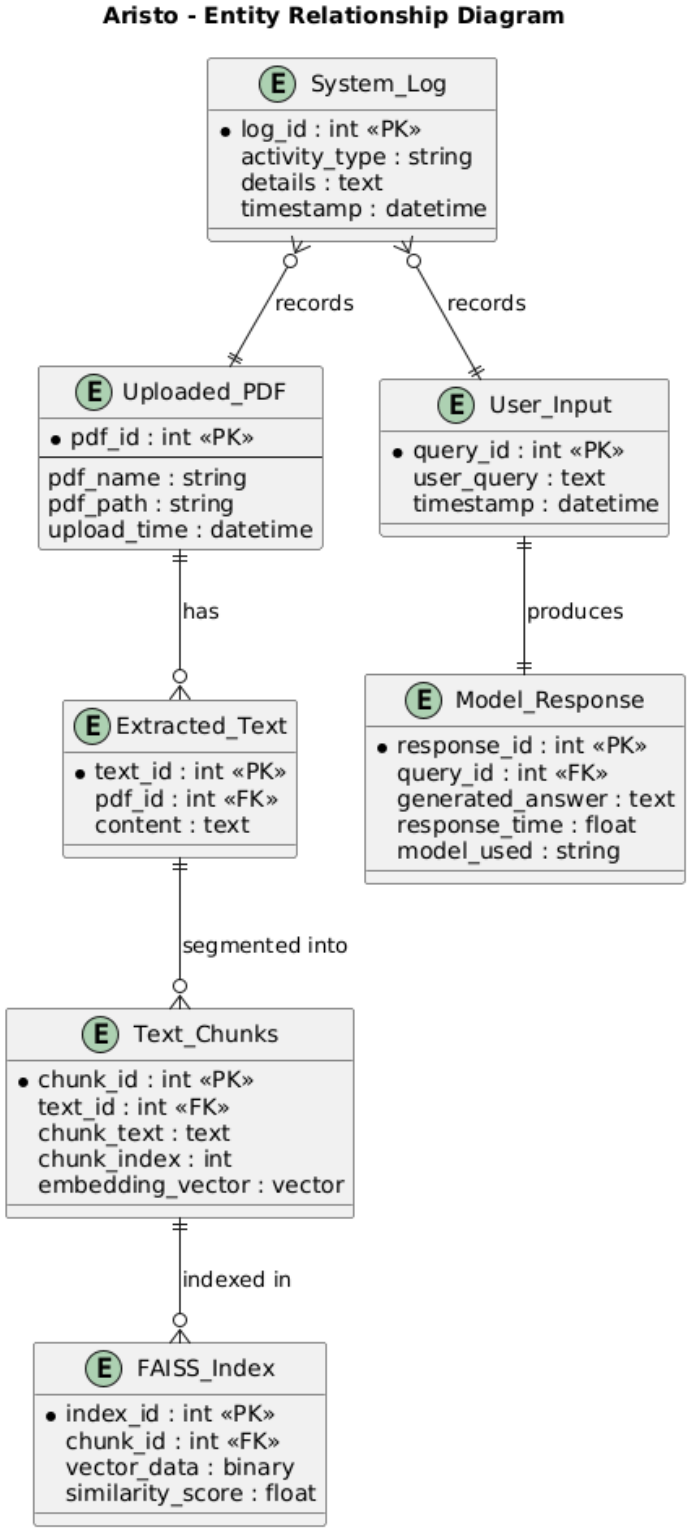

Figure 1: ER Diagram

3.4UML Diagrams

3.4.1 DFD Level 1

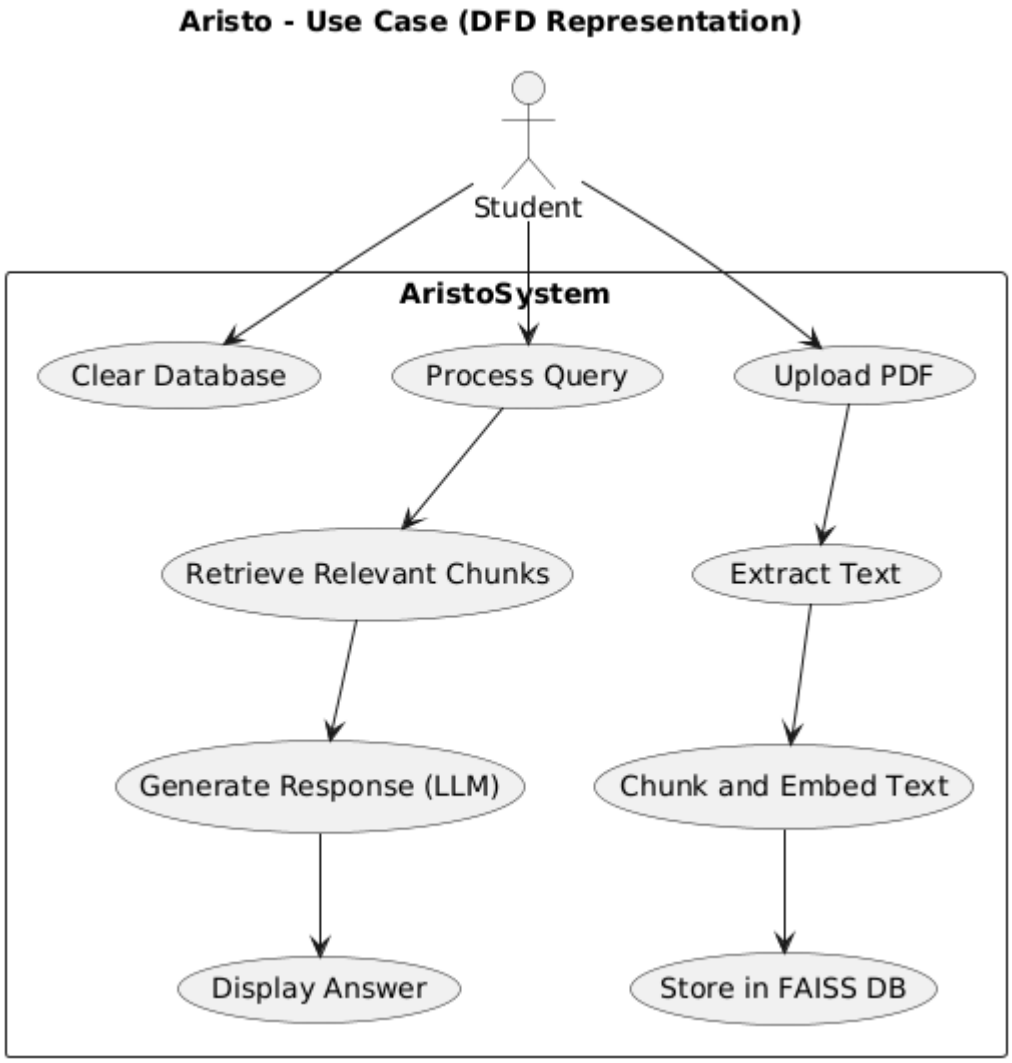

Figure 2: DFD Level 1

3.4.2 Class Diagram:

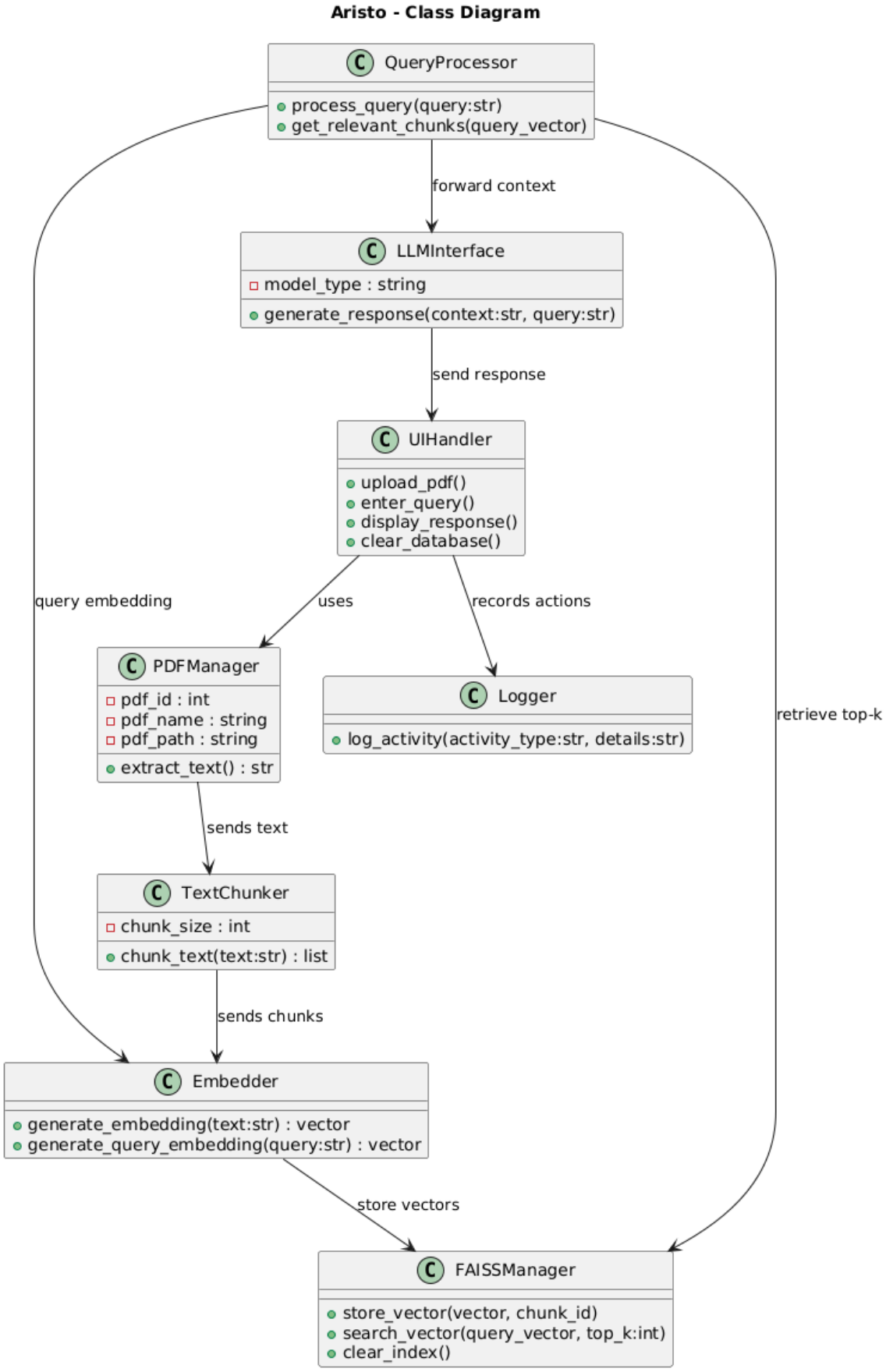

Figure 3: Class Diagram

3.4.3 Sequence Diagram:

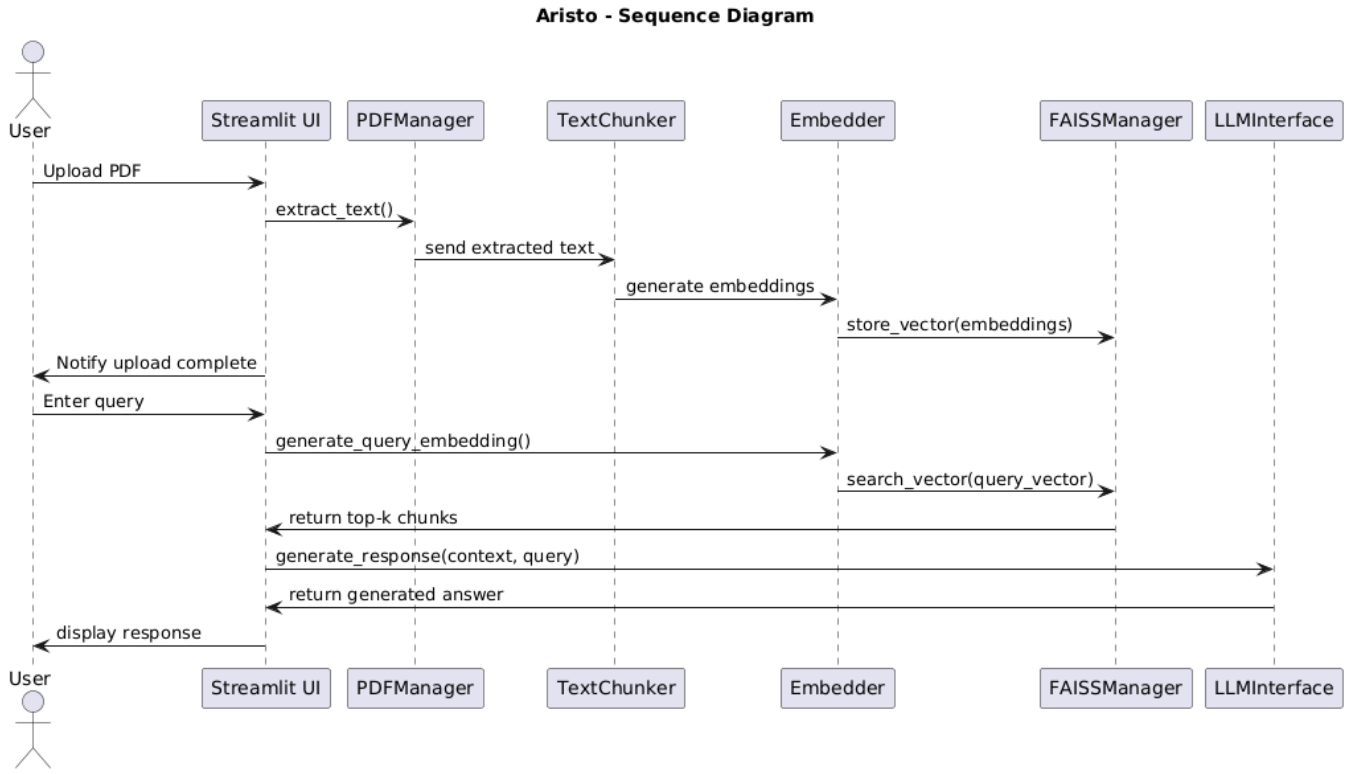

Figure 4: Sequence Diagram

3.4.4 Activity Diagram:

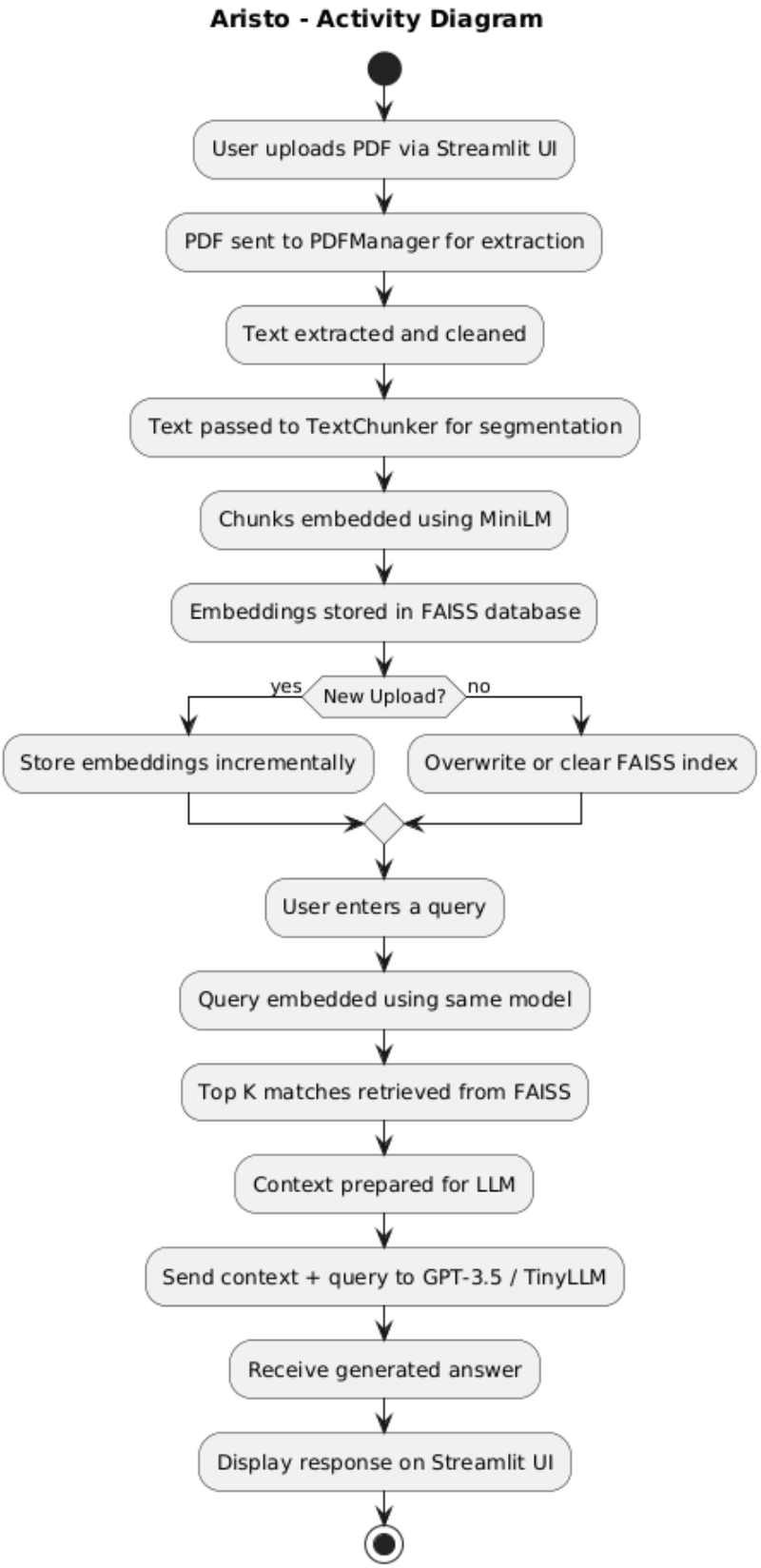

Figure 5: Activity Diagram