AI-POWERED ACADEMIC ASSISTANT

Melis Can

Software Engineering Department, Istanbul Atlas University, (220504019)

Abstract

*The increasing complexity of **academic subjects** and the diversity of student learning needs demand **innovative and adaptive educational technologies**. This project presents the design and implementation of an **AI-powered academic assistant** that leverages **Large Language Models (LLMs)**, **LangChain**, **LangGraph**, and **Retrieval-Augmented Generation (RAG)** to support students through two core modes: **topic explanation and document-based interaction**.*

*In **topic explanation mode**, the assistant uses **LLM reasoning** to generate concept-level academic responses. In **document mode**, users can upload **PDF materials** and request **summaries**, ask **specific questions**, or generate **quizzes** based on the content. All documents are vectorized using **HuggingFace embeddings** and stored **in Chroma** for efficient retrieval.*

*The **system architecture** is **modular** and directed by **LangGraph**, which dynamically routes queries based on **user intent**. Integrated via a **Streamlit interface**, the assistant supports **conversational memory** and provides structured, **context-aware outputs**.*

*Evaluation across various academic topics demonstrated the system's effectiveness in producing **relevant**, **fluent**, and **informative responses**. The **modular**, **agent-based design** offers a **scalable foundation** for building **intelligent academic support tools** aimed at improving **student engagement** and **learning efficiency**.*

1. INTRODUCTION

The rapid advancement of **large language models (LLMs)** and **agent-based AI frameworks** has unlocked new opportunities in **personalized education**. While digital learning platforms are increasingly accessible, students continue to face challenges in **understanding complex topics**, **navigating academic content**, and receiving **immediate**, **context-aware support**.

This project introduces an **AI-powered academic assistant** that utilizes **LLMs**, **LangChain**, **LangGraph**, and **Retrieval-Augmented Generation (RAG)** to assist learners through two primary modes: **topic-level explanation** and **document-based interaction**. In the topic explanation mode, the assistant generates structured and informative responses to conceptual academic queries. In document mode, users can upload PDF materials and receive summarized content, generate multiple-choice quizzes, or ask detailed questions grounded in the uploaded documents.

The assistant is implemented as a **modular**, **agent-directed system** using **LangGraph**, which dynamically routes user input to the appropriate processing chain based on **intent**. It also incorporates **conversational memory** to support coherent multi-turn dialogue. The system's design prioritizes **modularity**, **transparency**, and **extensibility**, and its performance is evaluated based on the fluency, relevance, and informativeness of its outputs across different academic domains.

2. RELATED WORK

In recent years, large language models (LLMs) have enabled the development of interactive academic assistants capable of engaging with students in natural dialogue. Notable examples include **Khanmigo**, developed by Khan Academy, which supports learners through interactive Q&A and guided tutoring, promoting self-directed learning [1]. Similarly, the University of Toronto's **All Day TA** successfully answered over 12,000 student questions in just one semester, showing the high demand for scalable AI agents in academic settings [2].

Another prominent academic agent is **Jill Watson**, which integrates ChatGPT with API-based modular tools to answer classroom queries [3]. Unlike traditional chatbots, Jill Watson offers a reusable and extensible framework for instructional support, emphasizing the potential of modular design in educational AI systems.

These agent-based platforms demonstrate the growing importance of AI in personalized education. However, most of these systems are either closed-source or institution-specific, which limits their adaptability, customizability, and transparency for independent development and research.

On another front, academic retrieval-based systems such as **SciSpace**, **CourseAssist** and **Study Flo** emphasize the effectiveness of Retrieval-Augmented Generation (RAG) techniques [4][5][6]. SciSpace focuses on making scientific research more accessible by analyzing and summarizing academic PDFs, while CourseAssist integrates course materials with LLMs to generate domain-specific answers. Similarly, Study Flo helps graduate researchers find and understand relevant literature by combining AI summarization with search capabilities. While these applications illustrate the utility of combining knowledge retrieval and generative reasoning, they often function as standalone black-box tools with limited extensibility or agent-level reasoning, which restricts their use as general-purpose academic assistants.

3. Overview, Methods and Tools

3.1 Project Goal and Core Functionality

This project aims to develop a modular and intelligent academic assistant that enhances student learning through the integration of **Large Language Models (LLMs)**, **LangChain**, **LangGraph**, and a **Retrieval-Augmented Generation (RAG)** pipeline.

The system operates in two primary modes:

- topic-level explanation
- document-based interaction

In **topic explanation mode**, students can ask **open-ended conceptual questions** and receive structured, context-aware responses. In **document mode**, users upload **PDF materials** and perform a variety of operations including **summarization**, **quiz generation**, and **document-grounded Q&A**.

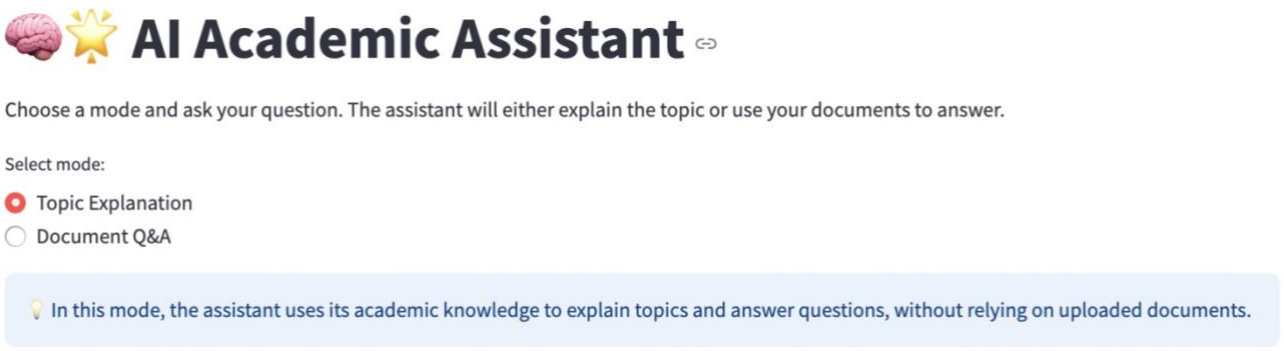

Figure 1. Topic Explanation mode interface

This interface allows users to ask general academic questions. The assistant responds using its internal knowledge base powered by a large language model, without referring to uploaded documents.

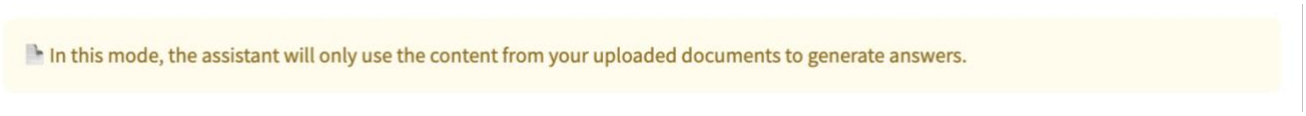

Figure 2. Document Q&A mode interface

In this mode, the assistant answers questions solely based on the content of the uploaded documents, enabling focused and source-grounded responses.

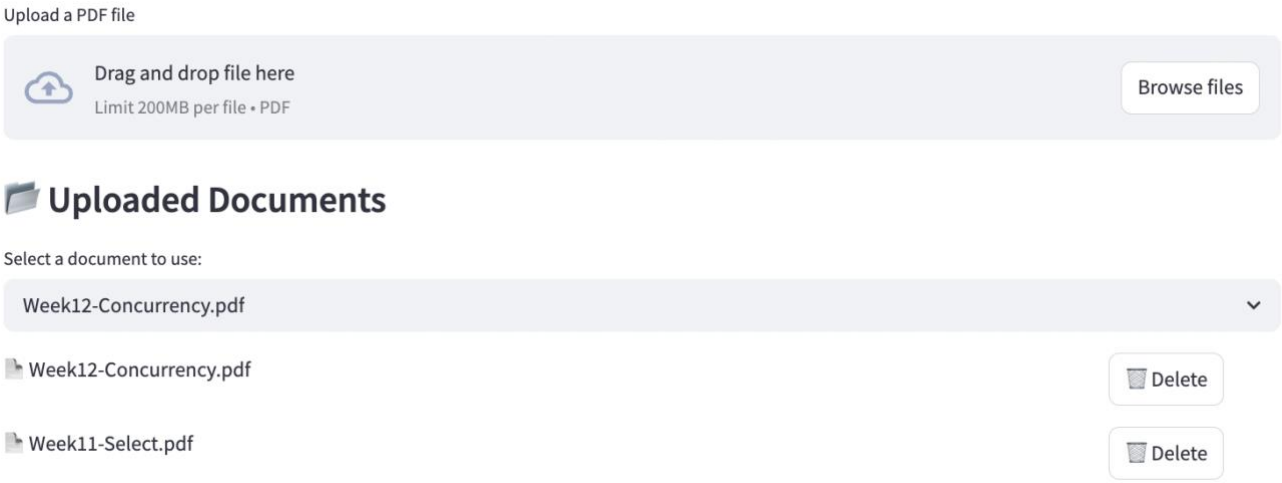

Figure 3. Document Upload and Selection Interface

This interface allows users to upload multiple PDF files and select one as the active document for analysis. The selected file is used for downstream tasks such as summarization, quiz generation, and document-based question answering.

3.2 LangGraph Routing and Hybrid Architecture

To meet the need for accurate and adaptive academic support, the system employs a **hybrid architecture** that combines **LLM reasoning** with **semantic retrieval**. Each user query is processed through a LangGraph agent, which acts as a dynamic decision router. Depending on the selected mode, LangGraph delegates the query to the appropriate chain, the **Explainer Chain for general academic topics** or the **Retriever Chain for document-specific responses**.

LangGraph's state-driven structure enables multi-step, traceable workflows, making the architecture well-suited for explainable AI systems.

3.3 Evolution of the System and Extended Modules

In the early development phase, only the Explainer Chain was implemented, providing general responses using a Groq-hosted LLM. Over time, the system was extended with the Retriever Chain, allowing users to upload academic content and receive precise, source-aware answers based on similarity search.

Additional functionality was integrated through

- **Summarization Chain** for condensing lengthy academic content
- **Quiz Chain** for generating multiple-choice assessments in JSON format

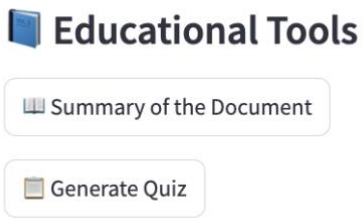

Figure 4. Educational Tools Panel

The Educational Tools panel provides users with quick access to document-based functionalities. Users can generate a summary of the selected PDF or create a multiple-choice quiz based on its content.

3.4 Memory Integration and Conversational Context

The assistant's internal memory is managed via LangChain's **ConversationBufferMemory**, which stores past user–assistant exchanges to preserve context in multi-turn conversations. This enhances coherence in extended dialogue and supports progressive topic exploration.

3.5 Document Processing and Retrieval Pipeline

For document processing, uploaded PDFs are divided into manageable chunks using **RecursiveCharacterTextSplitter** (a LangChain utility that splits text by recursively applying character-based rules, preserving semantic integrity). These chunks are vectorized using **HuggingFace Embeddings** (pretrained language model representations that convert text into high-dimensional vectors) and stored in **Chroma**, an open-source vector database optimized for fast similarity search over text embeddings.

Queries are matched against this database using **cosine similarity** (a distance metric that measures the angle between two vectors to evaluate their semantic similarity), and the **top-k results** (i.e., the k most relevant vectors with the highest similarity scores) are passed to the LLM for synthesis.

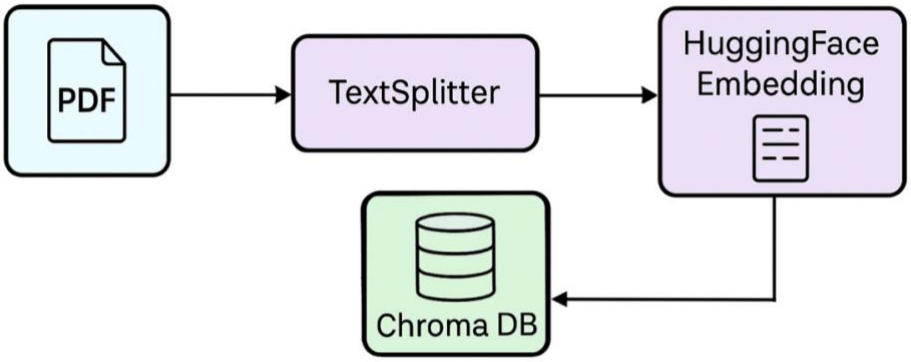

Figure 5. Document Embedding and Retrieval Pipeline

Figure 5 demonstrates how uploaded PDF files are segmented, embedded, and stored for semantic retrieval. This visual representation complements the processing steps described in **Section 3.5**.

3.6 System Components and Development Environment

The overall system architecture includes the following key components:

- **LangChain** for constructing LLM chains and managing document flow [7]
- **LangGraph** for agent-style routing and multi-step orchestration [8]
- **HuggingFace Embeddings** for converting text into semantic vectors
- **Chroma DB** for vector storage and top-k retrieval [9]
- **Streamlit** for a lightweight and interactive user interface
- **Groq SDK** for LLM inference via Groq-hosted models like **LLaMA 4 Scout** and **Gemma 7B**

Development was carried out using **Python 3.11** in a **Conda-managed virtual environment** within **VS Code**. Environment variables and API keys were managed using *.env* files to ensure secure configuration. The system's modular architecture supports isolated testing and development of each component, allowing for rapid iteration and future extension.

4. ARCHITECTURE, ALGORITHMS, MODELS AND DATA

4.1 System Architecture Overview

The system architecture of the AI-powered academic assistant follows a modular, chain-based design that is orchestrated using LangGraph and LangChain. As illustrated in **Figure 6**, user input is dynamically routed through a LangGraph state machine, which determines whether the query should be processed by the **Explainer Chain** or the **Retriever Chain**. These are the two primary functional modules that support topic-level and document-based interactions, respectively.

This architecture is implemented in the ***langgraph_chain.py*** file, where each module is defined as a separate ***RunnableLambda*** node within a ***StateGraph***. The routing decision is based on the ***mode*** field in the user input (e.g., ***"explain"*** or ***"qa"***), which allows the assistant to delegate the task without modifying the graph's overall structure.

LangGraph ensures modular control and interpretable execution by defining decision logic through directed edges and conditional entry points. Each node in the graph represents a callable chain that is responsible for a specific academic function. In document-based mode, the **Retriever Chain** may further invoke additional post-processing chains, such as the **Summarization Chain** and the **Quiz Generator Chain**, depending on the user's selected task. These chains, while technically independent LangChain-based components, operate as logical extensions of the Retriever Chain.

This modular and layered design allows each chain to be developed and tested independently, supports traceable multi-step workflows, and ensures scalability for future academic features.

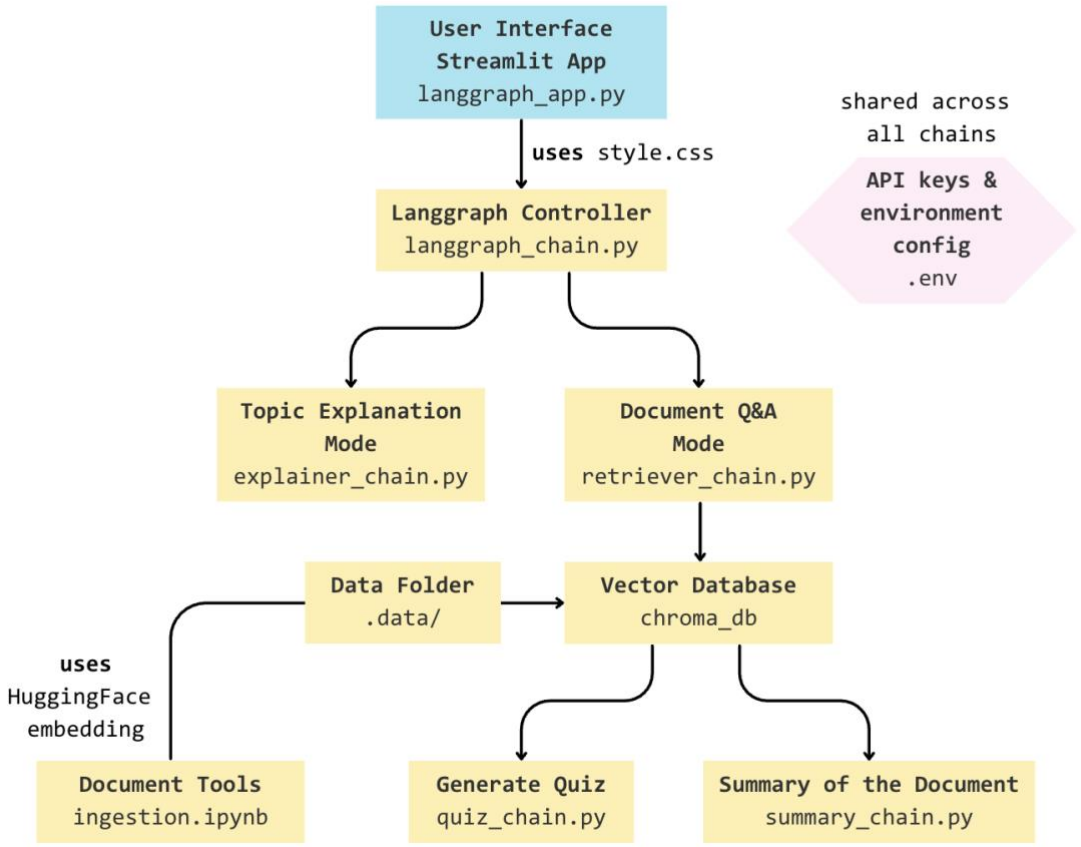

Figure 6. System Architecture Flow

This diagram (**Figure 6)** presents the modular structure of the AI Academic Assistant. The user interface communicates with a central controller that manages two main modes: topic explanation and document-based Q&A. Supporting components handle document ingestion, vector storage, summarization, and quiz generation. Shared configurations and styling ensure consistent and maintainable operation.

4.2 LangGraph Flow and State Management

The decision-making logic of the academic assistant is implemented using LangGraph, a graph-based framework that enables conditional execution and modular task routing. At the heart of this architecture is the StateGraph, a structure that defines how user input should be processed based on the selected mode.

The LangGraph flow is defined in the ***langgraph_chain.py*** file. This file declares a set of nodes (functions wrapped as RunnableLambda objects) and connects them using directed edges. The graph includes a **START** node, decision branches for ***"explain"*** and ***"qa"*** modes, processing nodes for the Explainer and Retriever chains, and an **END** state that signals completion.

Each chain, whether Explainer or Retriever, is encapsulated in a callable node. The system determines the correct chain by evaluating the ***"mode"*** key from the input dictionary. For example, if the mode is ***"explain"***, the Explainer Chain node is invoked; if it is ***"qa"***, the Retriever Chain is selected.

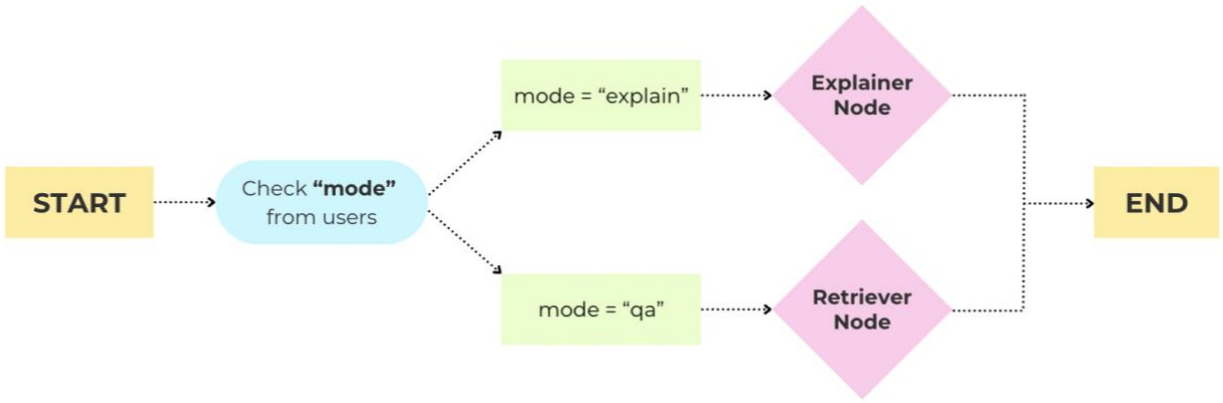

Figure 7. LangGraph Routing Flow

Figure 7 illustrates the conditional routing logic implemented using LangGraph. Based on the user's selected mode, the system activates either the Explainer Node or the Retriever Node before proceeding to the END state.

This setup allows for high modularity and traceability. Each component is loosely coupled, meaning new nodes (such as a summarization-only chain or a diagnostic tool) could be added in future versions without disrupting the main flow. Additionally, because all nodes comply with LangChain's Runnable interface, they can be independently tested and reused.

LangGraph's stateful execution model not only enables this flexibility but also supports the development of explainable AI systems by making the flow of data and decisions explicit and traceable.

4.3 Explainer Chain and Prompt Design

The Explainer Chain is responsible for generating concept-level academic responses to open-ended queries. This chain is triggered when the user selects the **"explain"** mode in the application interface. It is implemented in the ***explainer_chain.py*** file using **LangChain's LLMChain structure**, and is designed to provide detailed, structured, and context-aware explanations on a wide range of topics.

The chain architecture consists of three main components:

a) Prompt Template with Conversation Context

Unlike simple static prompts, the Explainer Chain uses a **multi-message chat-style prompt** defined via *ChatPromptTemplate.from_messages*. This approach enables the model to respond more intelligently by incorporating prior conversation history.

The prompt is composed of two roles:

```
# Prompt template with conversation context
prompt_template = ChatPromptTemplate.from_messages([
    ("system", "You are an academic assistant. Use the previous conversation to inform your response."),
    ("human", "{chat_history}\n\nUser: {topic}\nAI:")
])
```

- The **system message** sets the assistant's behavior and ensures consistency in academic tone and role.
- The **human message** injects the current topic along with any prior chat history. This allows the LLM to maintain coherence across multiple user turns and generate more contextually relevant explanations.

By incorporating *{chat_history}*, the chain supports multi-turn dialogue, making the experience more natural and useful for in-depth academic exploration.

b) LLM Integration via Groq API

The chain uses **Groq-hosted LLMs**, specifically **meta-llama/llama-4-scout-17b-16e-instruct**, which is known for fast response generation and high-quality academic reasoning. The integration is handled through LangChain's Groq module, which allows prompt execution via a simple interface while abstracting away API-level complexity.

```
# Define the LLM
llm = ChatGroq(
    model_name="meta-llama/llama-4-scout-17b-16e-instruct",
    api_key=groq_api_key,
    temperature=0.3
)
```

The model is accessed with a low temperature value to ensure factual, deterministic, and repeatable outputs, essential qualities for academic tools.

c) Execution Flow and LangGraph Compatibility

When the Explainer Chain is invoked, the **"topic"** input is passed along with *chat_history* to the prompt. The formatted prompt is then sent to the LLM, and the response is returned as structured text. The entire chain is encapsulated in a RunnableLambda wrapper to ensure compatibility with **LangGraph's StateGraph** execution model.

This makes the Explainer Chain an independently callable unit (node) within the system's routing logic. It can be tested in isolation, reused in other workflows, and modified without affecting other components of the graph.

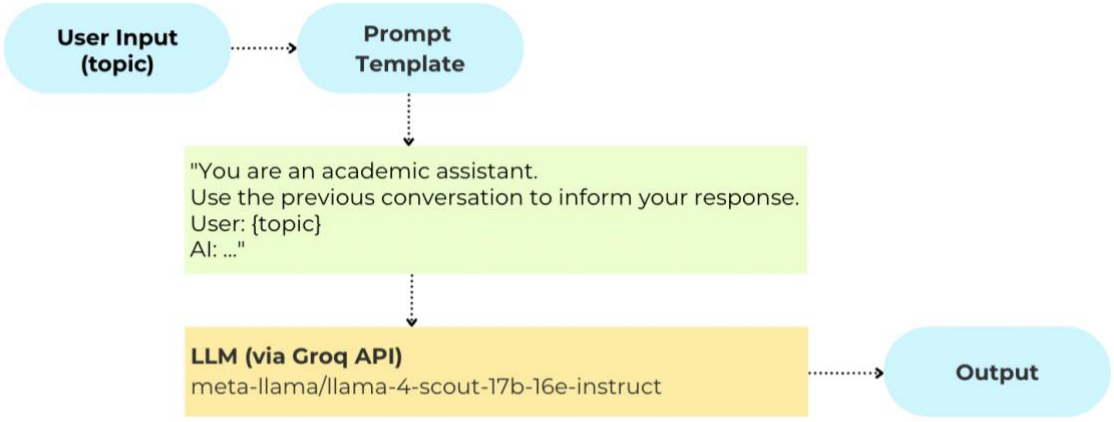

Figure 8. Explainer Chain Architecture

Figure 8 illustrates the full execution flow of the Explainer Chain module. User input is embedded into a structured prompt that includes conversation history and is processed by a Groq-hosted LLM to generate the final output.

Benefits of Design

This architecture offers several design benefits, as outlined below:

- **Modularity:** Clear separation of prompt logic, model invocation, and routing behavior.
- **Context-awareness:** By embedding chat history, the system simulates continuity in conversations.
- **Scalability:** Can be expanded with different prompt templates, models, or even response post-processing if needed.

4.4 Retriever Chain and Document QA Logic

The **Retriever Chain** is responsible for handling document-based academic queries. This chain is triggered when the user selects the **"qa"** mode, and is implemented in the ***retriever_chain.py*** file using LangChain's **ConversationalRetrievalChain**. It integrates semantic search over vectorized documents with large language model (LLM) reasoning to generate grounded and accurate responses.

a) Retrieval Pipeline

When a user uploads a document and asks a question, the Retriever Chain follows this pipeline:

1. **Semantic Chunking & Vectorization:** The uploaded PDFs are split into overlapping text chunks using LangChain's *RecursiveCharacterTextSplitter* during the ingestion phase (see Section 4.7). Each chunk is embedded using HuggingFace Embeddings and stored in a Chroma vector database. This preprocessing step ensures that the retriever can return semantically meaningful and focused document segments.
2. **Similarity Matching:** When a query is received, it is embedded and compared to the document chunks using cosine similarity. The top-k most relevant chunks are retrieved.
3. **LLM Querying:** These retrieved chunks, together with the user query, are passed to the LLM (served via Groq API). The model generates an answer using the retrieved content as context.

b) Implementation Details

The current Retriever Chain setup avoids LangChain memory wrappers by passing the chat history manually. It is instantiated and called as follows:

```
# Create a ConversationalRetrievalChain with source documents returned
qa = ConversationalRetrievalChain.from_llm(
    llm=llm,
    retriever=retriever,
    return_source_documents=True
)

# Send the query with previous chat history
response = qa({
    "question": query,
    "chat_history": st.session_state.chat_history  # Stored in session
})
```

- **llm:** Groq-hosted *meta-llama/llama-4-scout-17b-16e-instruct*
- **retriever:** Chroma-based vector retriever with *all-MiniLM-L6-v2* embeddings
- **prompt:** ChatPromptTemplate used in backend chain for context fusion
- **chat_history:** Passed manually from st.session_state
- **output:** Returns both the generated answer and the source metadata

Note: The Retriever Chain uses a custom multi-message prompt via *ChatPromptTemplate*, allowing the LLM to reason over both chat history and retrieved document chunks.

c) Output Formatting and Traceability

Each response includes:

- The generated answer
- The metadata of the source document (file name, page number)

This makes the assistant's outputs verifiable and encourages critical thinking by directing students to the exact document page used for the answer.

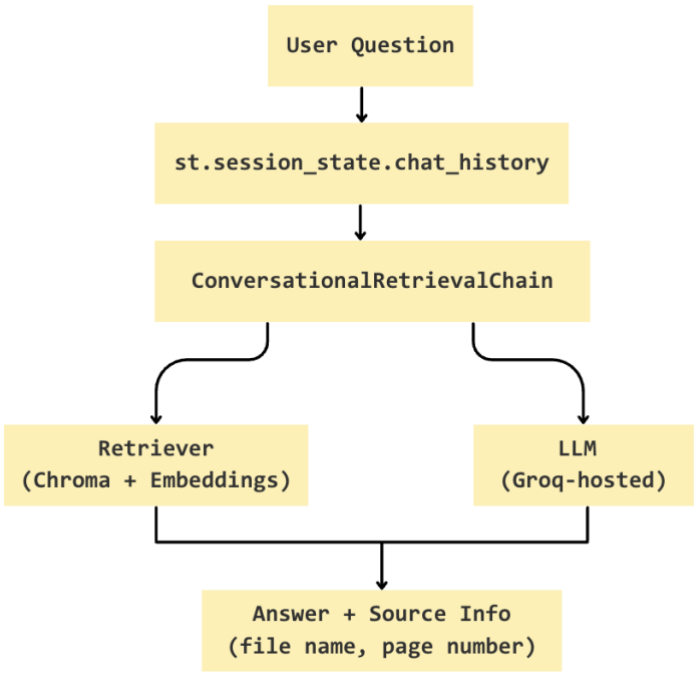

Figure 9. Retriever Chain Architecture

Figure 9 shows how the Retriever Chain processes user input using memory and prompt templating before generating a context-aware answer with the help of a retriever and LLM.

4.5 Summarization Module

The Summarization Module enables users to generate concise summaries of academic PDF documents uploaded to the system. It is implemented using ***LangChain's load_summarize_chain()*** method in **map-reduce mode**, which allows for scalable summarization of long documents by breaking them into chunks.

a) Summary Request Flow

When a user selects a document and clicks **"Generate Summary"**, the system:

1. Loads the selected PDF and extracts all pages as a list of *Document* objects.
2. Splits the full content into manageable chunks using *RecursiveCharacterTextSplitter* (chunk size = 1000, overlap = 100).
3. Passes the chunks to a summarization chain, which first summarizes each chunk individually (map step), then merges these into a final overall summary (reduce step).

b) LLM and Chain Configuration

The chain is created with:

```
# Create the summarization chain using the "map_reduce" strategy
chain = load_summarize_chain(llm, chain_type="map_reduce")
```

- **llm:** A Groq-hosted large language model (meta-llama/llama-4-scout-17b-16e-instruct)
- **chain_type:** The map_reduce strategy enables efficient summarization of large multi-page documents

c) Output Handling

Once the summarization is complete, the output is returned as a single text string. This summary is displayed in the Streamlit interface.

4.6 Quiz Generator Module

The Quiz Generator Module enables users to automatically generate multiple-choice questions based on the content of an uploaded PDF. This feature is particularly useful for self-assessment, revision, and comprehension testing.

a) Prompt and Chain Structure

This module uses LangChain's LLMChain to generate quiz questions from raw document text. The prompt is carefully designed to elicit exactly 10 multiple-choice questions, each with 4 options and 1 correct answer.

The prompt explicitly instructs the LLM to output a valid JSON array, ensuring the results can be directly parsed and displayed in the frontend.

```
# Define the prompt template for generating a quiz
prompt = PromptTemplate.from_template(
"""
You are a quiz generator.

Your task is to generate exactly 10 multiple-choice questions based on the following document content.
Each question must have 4 answer options and only one correct answer.

Return the result strictly in **valid JSON array** format like this:

[
  {{
    "question": "What is select() used for?",
    "options": ["To read input", "To monitor multiple descriptors", "To send signals"],
    "answer": "To monitor multiple descriptors"
  }},
  ...
]

Do not include any explanations or markdown. Just the JSON array only.

Content:
{content}
"""
)
```

b) Model and Output Format

- **LLM:** *meta-llama/llama-4-scout-17b-16e-instruct*, accessed via Groq
- **Chain:** LLMChain with a custom PromptTemplate
- **Input:** Plain text content from the uploaded document
- **Output:** JSON array containing question–option–answer sets

The resulting JSON is parsed and rendered as an interactive quiz in the application UI.

c) Error Handling

If the LLM fails to return a valid JSON array (e.g., due to formatting errors or incomplete output), the system returns a placeholder question that indicates failure, allowing the frontend to handle the issue gracefully.

4.7 Ingestion and and Vector Database Creation

The ingestion pipeline is responsible for transforming user-uploaded PDF documents into vectorized knowledge chunks that can be semantically retrieved later by the Retriever Chain. This process ensures accurate and context-aware document Q&A functionality.

a) Document Loading and Parsing

Documents are parsed using *PyPDFLoader*, which extracts text content from each page of the uploaded PDFs. This method allows for precise segmentation of academic materials into manageable units for embedding.

b) Chunking and Embedding

Extracted documents are split into overlapping text segments using *RecursiveCharacterTextSplitter*, configured with a chunk size of 1000 characters and an overlap of 100 characters. These chunks preserve semantic coherence and ensure better retrieval accuracy.

The resulting text chunks are converted into high-dimensional vectors using ***HuggingFaceEmbeddings***, specifically the "*all-MiniLM-L6-v2*" model.

c) Chroma Vector Database

The embedded chunks are stored in a local Chroma vector database:

```
vectordb = Chroma.from_documents(docs, embedding_function, persist_directory="./chroma_db")
vectordb.persist()
print("✅ Created and saved new Chroma vector database.")
```

- This operation is repeated every time the ingestion pipeline is run, meaning the vector database is **rebuilt from scratch**.

The ingestion process is modular, reproducible, and optimized for semantic document retrieval. It plays a foundational role in enabling accurate, source-grounded answers during document-based Q&A.

4.8 Application Launch and Interface Integration

The final integration layer of the AI Academic Assistant ties together modular LangGraph-based chains, retrieval pipelines, and document utilities through an intuitive user interface built with **Streamlit**. The application offers two main modes of interaction: **Topic Explanation** and **Document Q&A**, both routed through the unified backend logic.

a) Streamlit Frontend

The Streamlit-based UI provides a seamless academic experience through:

- A mode selector to switch between topic-based and document-based reasoning
- A dynamic chat window with scrollable history and styled messages
- File upload and deletion interface for PDF documents
- A multi-step quiz solving section
- On-demand execution of document summarization and quiz generation

Upon user input, a query and mode are combined into a state dictionary and passed to *langgraph_chain.invoke(state)* for backend processing.

b) Educational Tools and Dynamic Document Handling

When "Document Q&A" mode is selected:

- Users can upload PDFs, which are parsed using *PyPDFLoader*
- Pages are embedded using *HuggingFaceEmbeddings* and stored in Chroma
- Users can:
  - Click **"Summary of the Document"** to generate a high-level overview (via the Summarization Chain)
  - Click **"Generate Quiz"** to produce 10 multiple-choice questions in JSON format
  - Solve the quiz within the same interface and view their performance

c) Visual Design and Styling

The UI is enhanced by a custom CSS stylesheet, which defines:

- Left/right-aligned chat bubbles for user and assistant roles
- Scrollable chat window
- Responsive layout and sticky input box
- Consistent spacing and padding for a professional look

This improves readability, accessibility, and engagement for academic users.

d) Launch and Reset Functionality

The application is launched using: `streamlit run langgraph_app.py`

It reads environment variables securely from .env and allows full session control with:

- "Clear Chat History" to wipe conversation memory
- "Reset Database" to delete and rebuild the vector index

This layer transforms a modular backend into an interactive educational assistant. With a user-friendly Streamlit interface, dynamic document handling, and real-time retrieval and explanation chains, the assistant is fully operational as a self-contained academic tool.

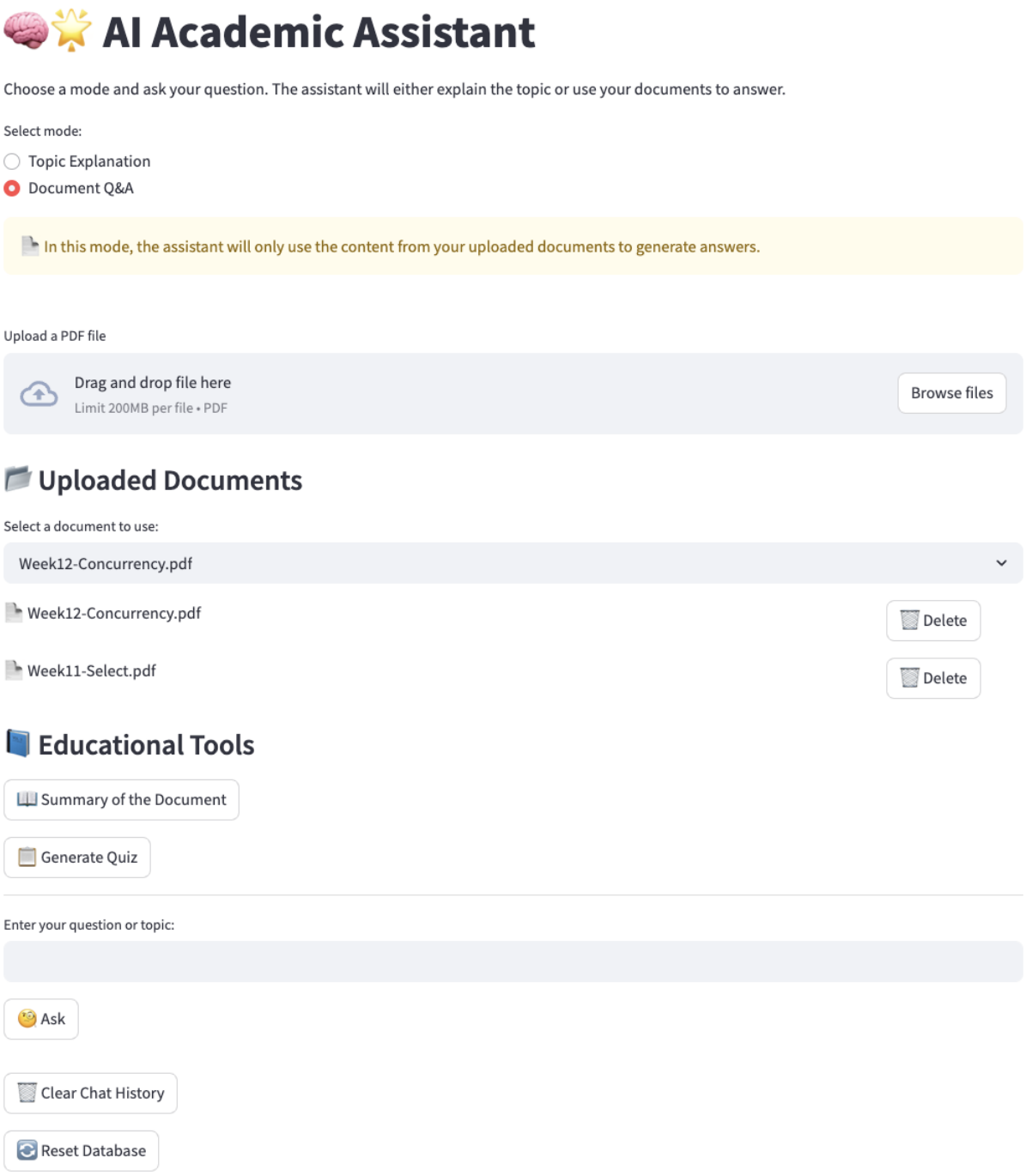

Figure 10. Streamlit-Based User Interface of the Academic Assistant

Figure 10 displays the front-end interface built with Streamlit. Users can select between Topic Explanation and Document Q&A modes, upload PDF files, and interact with educational tools such as summarization and quiz generation. The layout also includes interactive chat input, history clearing, and vector database reset functionality.

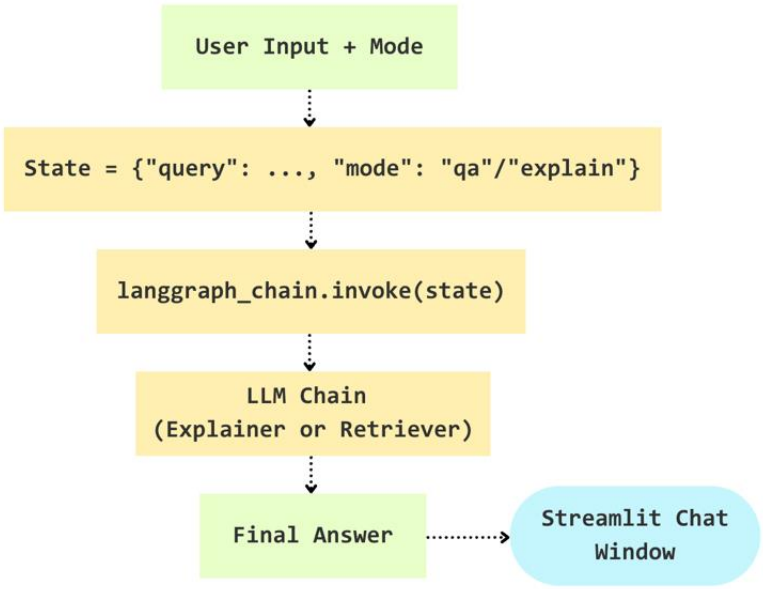

Figure 11. Mode Routing Flow (UI → Backend)

Figure 11 illustrates how user input and selected mode are routed through the LangGraph chain to dynamically invoke the appropriate backend logic.

5. EXPERIMENTS, ANALYSIS AND PERFORMANCE

5.1 Evaluation Goals

The evaluation phase of this project aims to assess the functionality, accuracy, and contextual coherence of the AI-powered academic assistant across its core modules. The following four aspects are selected as key performance targets:

a) Topic Explanation Accuracy

The system's ability to generate concept-level academic explanations using LLM-based reasoning is evaluated based on factual correctness, clarity, and adherence to academic tone.

b) Document Q&A with Source Traceability

In Document Q&A mode, the assistant is expected to generate answers strictly based on uploaded PDFs. Evaluation focuses on whether the responses are directly supported by document content and whether source references (filename and page number) are correctly identified and displayed.

c) Conversational Memory Consistency

The assistant is tested in multi-turn dialogue settings to measure whether it maintains contextual continuity and utilizes prior conversation history effectively through ConversationBufferMemory.

d) Quiz Generation Validity and Format Compliance

The quiz generation module is evaluated based on its ability to produce valid JSON output with 10 multiple-choice questions per document. Each quiz is assessed for content relevance, format integrity, and the uniqueness and plausibility of answer options.

5.2 Functional Test Cases

A. Topic Explanation – Concept Clarity

Query: *"What is TCP congestion control?"*

Evaluation: The assistant responded with a detailed explanation covering the definition, core principles, and various congestion control algorithms such as Reno, NewReno, Cubic, and BBR. The explanation accurately described the goal of congestion control, how TCP dynamically adjusts its transmission rate, and how specific algorithms function in different network environments.

The response demonstrated a high level of factual correctness, clarity, and adherence to an academic tone. It avoided any hallucinated or irrelevant content, and concluded with an optional follow-up suggestion in a conversational but appropriate manner.

This result confirms that the Topic Explanation module is capable of producing technically accurate, well-structured, and contextually appropriate academic responses based solely on the LLM's internal knowledge.

Criterion	Result
Factual Accuracy	Complete
Clarity	Clear
Academic Tone	Appropriate
Relevance	Consistent

Table 1. Evaluation Summary of Topic Explanation Output

B. Document Q&A – Retrieval Testing

Query: *"What does the select() system call do, and how is it used in I/O multiplexing?"*

Evaluation: The assistant provided a technically sound and comprehensive answer based entirely on the uploaded document (*Week11-Select.pdf*). The explanation included the purpose of the select() system call in the context of I/O multiplexing, outlined its core functionality, and listed the arguments it takes (maxfdp1, readset, writeset, exceptset, and timeout). It also explained return conditions and summarized the usage with clarity and structure.

The assistant's response demonstrates a strong alignment with the source document, maintaining factual accuracy, technical clarity, and appropriate academic tone. The citation of the specific page (Page 9) further confirms proper document grounding.

This result validates that the Document Q&A mode can accurately retrieve relevant information from source documents, synthesize it coherently, and present it in a well-structured academic response.

Criterion	Assessment
Factual Accuracy	Aligned with source
Clarity	Well-articulated
Academic Tone	Formally appropriate
Relevance to Query	Direct and specific
Source Reference	Yes (Page 9)

Table 2. Evaluation Summary of Document Q&A Response

C. Memory Continuity – Multi-turn Question Resolution

Query Sequence:

1. *"What is the purpose of the select() system call?"*
2. *"How many arguments does select() take and what are they?"*
3. *"What happens when the timeout value is 0?"*

Evaluation: The assistant accurately answered all three queries in sequence by preserving context across turns. It began with a general explanation of the select() system call and its use in I/O multiplexing, followed by a detailed enumeration of the function's five arguments. In the third response, the assistant correctly referred back to the timeout argument and explained its behavior when set to 0.

The responses were technically sound, consistent with each other, and demonstrated an understanding of prior dialogue without requiring question repetition. This confirms that the memory feature in the Document Q&A mode enables multi-turn, context-aware interactions based solely on the uploaded PDF file.

Criterion	Result
Factual Accuracy	Complete
Clarity	Clear
Academic Tone	Appropriate
Relevance	Consistent
Context Awareness	✓ Preserved

Table 3. Evaluation Summary of Memory Continuity in Document Q&A

D. Document Summarization Evaluation

Input Document Overview (*Testing-05.pdf)*

The uploaded document provides a concise overview of key software testing concepts, particularly focusing on:

- **Static Testing:** Techniques for analyzing code without execution, including code reviews, static analysis, and walkthroughs.
- **Dynamic Testing:** Methods that involve executing software to identify defects and validate behavior under real usage conditions.
- **Review Process:** Structured procedures for reviewing software artifacts, detailing roles (e.g., manager, moderator, reviewer) and types of reviews (informal, walkthrough, technical, inspection).
- **Key Success Factors:** Practical strategies that enhance review effectiveness, such as setting clear objectives, providing training, and integrating reviews into organizational culture.

This document serves as a foundational reference for understanding quality assurance practices in software engineering, suitable for both instructional and evaluative purposes.

Evaluation: The summarization module accurately captured the structure and content of the source document, presenting it in a well-organized and concise format. The inclusion of key headings and bullet-point explanations makes the output easy to scan and useful for both study and review purposes.

Criterion	Result	Notes
Coverage	High	Main sections (Static/Dynamic Testing, Review Process) are fully covered.
Conciseness	Clear	Avoids unnecessary detail; uses concise, bullet-point format.
Faithfulness	Accurate	Stays faithful to source content with no distortion of meaning.
Usefulness	Practical	Preserves topic integrity while enabling fast information access.

Table 4. Evaluation Summary of Document-Based Summarization Output

E. Quiz Generation and Solving

Evaluation: The AI-powered assistant successfully generated a 10-question multiple-choice quiz based on the uploaded document, Testing 05.pdf. The quiz content was directly aligned with the document topics, such as static and dynamic testing, review process, and quality assurance strategies. The questions were clear, relevant, and appropriately structured for academic purposes.

After solving the quiz, the scoring system correctly identified the user's answers, highlighted mistakes, and provided immediate feedback. This confirms the effectiveness of the quiz generation module in transforming technical documents into interactive evaluation tools.

Criterion	Result	Notes
Question Quality	High	Questions are well-formed and match the document content.
Answer Accuracy	Correct	The correct options reflect the source material faithfully.
Clarity	Clear	No ambiguity or grammatical issues observed in the quiz text.
Educational Value	Effective	Reinforces learning by enabling immediate assessment and feedback.

Table 5. Evaluation Summary of Quiz Generation and Solving

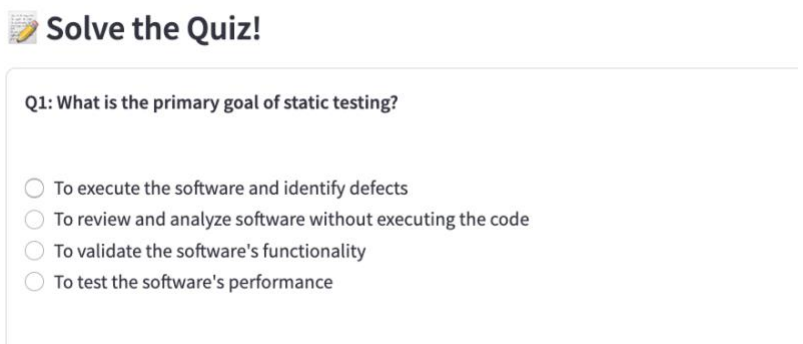

Figure 12. Example of the multiple-choice quiz based on document content.

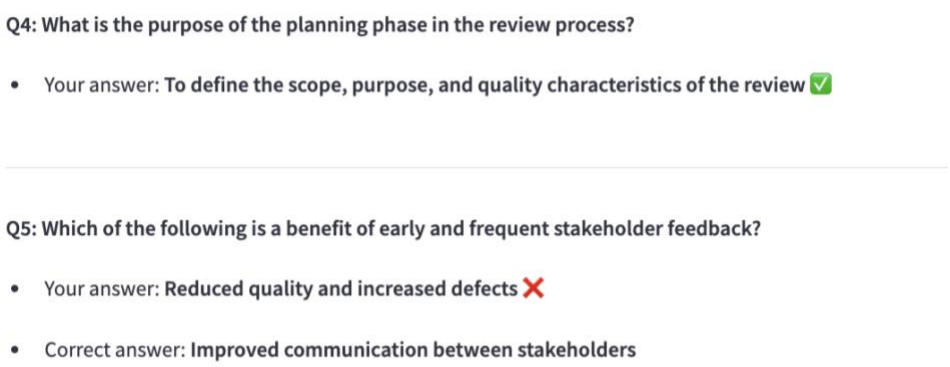

Figure 13. Example of quiz results output based on document content.

6. CONCLUSIONS

The project explored the development and assessment of a modular AI-powered academic assistant built on **LangChain**, **LangGraph**, and **Retrieval-Augmented Generation (RAG)** technologies. The assistant was evaluated across several operational modes, including topic explanation, document-based question answering, summarization, and quiz generation. Performance across these tasks demonstrated consistent accuracy in both general reasoning and document-grounded contexts.

Test results highlighted the assistant's ability to produce context-aware, coherent, and academically appropriate outputs. The incorporation of conversational memory using LangChain's *ConversationBufferMemory* enabled continuity across multi-turn interactions, while Chroma-supported vector retrieval facilitated effective access to relevant document content. The summarization and quiz modules further supported active learning and self-assessment.

The system's architecture offers a robust foundation for future enhancements such as feedback integration, adaptive personalization, or domain-specific optimization. These capabilities position the assistant as a practical tool for enhancing student engagement and supporting academic success.

7. REFERENCES

[1] Khan Academy, "Khanmigo – Your AI learning guide," *Khanmigo*, 2024. [Online]. Available: https://www.khanmigo.ai/

[2] J. Gans, "AI professor handled 12,000 student queries in 12 weeks," *Financial Times*, Mar. 4, 2024. [Online]. Available: https://www.ft.com/content/daa0f68d-774a-4e5e-902c-5d6e8bf687dc

[3] A. Goel, D. Joyner, and M. Najafi, "Jill Watson: A virtual teaching assistant using large language models," *arXiv preprint arXiv:2405.11070*, May 2024. [Online]. Available: https://arxiv.org/abs/2405.11070

[4] SciSpace, "Your AI Copilot for Research," *SciSpace*, 2024. [Online]. Available: https://scispace.com/

[5] M. A. Kowsari and H. S. Thompson, "CourseAssist: A personalized LLM-based assistant for computer science education," *arXiv preprint arXiv:2407.10246*, Jul. 2024. [Online]. Available: https://arxiv.org/abs/2407.10246

[6] Reddit user u/studyflowdev, "I built an AI research assistant for students and researchers," *Reddit /r/PhDProductivity*, Jun. 2024. [Online]. Available:https://www.reddit.com/r/PhdProductivity/comments/1ekp9xh/i_built_an_ai_research_assistant_for_students/?rdt=61918

[7] LangChain AI, "LangChain – Context-aware reasoning agents powered by LLMs," *GitHub Repository*, 2024. [Online]. Available: https://github.com/langchain-ai/langchain

[8] LangChain AI, "LangGraph – Stateful multi-actor agent framework," *GitHub Repository*, 2024. [Online]. Available: https://github.com/langchain-ai/langgraph

[9] Chroma, "Chroma – The AI-native open-source vector database," *GitHub Repository*, 2024. [Online]. Available: https://github.com/chroma-core/chroma