Academic Assistant Pro – AI Tutor with Retrieval-Augmented Generation (RAG)

by

MD. Jakaria
ID: CSE2203027060

Most. Musumi Akter
ID: CSE2203027006

MD. Mizanur Rahman Juel
ID: CSE2203027143

Amit Sarkar Anto
ID: CSE2203027019

Supervised by
Salma Tabashum
Assistant Professor

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

May 2026

APPROVAL

The project titled "Academic Assistant Pro – AI Tutor with Retrieval-Augmented Generation (RAG)" submitted by MD. Jakaria (CSE2203027060), Most. Musumi Akter (CSE2203027006), MD. Mizanur Rahman Juel (CSE2203027143), and Amit Sarkar Anto (CSE2203027019) to the Department of Computer Science and Engineering, has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering and approved as to its style and contents.

Board of Examiners

_____

Salma Tabashum
Assistant Professor & Supervisor
Department of Computer Science and Engineering

_____

(Examiner Name and Signature)
Department of Computer Science and Engineering
Examiner 1

_____

(Examiner Name and Signature)
Department of Computer Science and Engineering
Examiner 2

_____

(Examiner Name and Signature)
Department of Computer Science and Engineering
Examiner 3

DECLARATION

We, hereby, declare that the work presented in this report is the outcome of the investigation performed by us under the supervision of Salma Tabashum, Department of Computer Science and Engineering. We reaffirm that no part of this project has been or is being submitted elsewhere for the award of any degree or diploma.

Countersigned

(Salma Tabashum)
Supervisor

Signature:

MD. Jakaria
ID: CSE2203027060

Most. Musumi Akter
ID: CSE2203027006

MD. Mizanur Rahman Juel
ID: CSE2203027143

Amit Sarkar Anto
ID: CSE2203027019

ABSTRACT

The rapid advancement of artificial intelligence and natural language processing has opened new avenues for enhancing educational experiences. Traditional digital learning tools, while effective in delivering static content, lack the interactive, adaptive, and personalized nature of human tutoring. The emergence of Large Language Models (LLMs) has begun to bridge this gap, but their well-documented tendency to produce hallucinated or unverifiable information has limited their adoption in academic settings where factual accuracy is non-negotiable. This project presents **Academic Assistant Pro**, an AI-powered tutoring application that leverages Retrieval-Augmented Generation (RAG) to provide contextually accurate, document-specific academic assistance. Unlike conventional chatbots that rely solely on pre-trained knowledge, Academic Assistant Pro ingests user-uploaded PDF documents, chunks and embeds them into a vector database, and retrieves relevant passages to ground the large language model's responses in verified source material.

The application is built using Python with PySide6 for a cross-platform graphical user interface, LangChain for the RAG pipeline orchestration, Ollama for local large language model inference (using Qwen2.5:3b), and Chroma as the persistent vector store with nomic-embed-text embeddings. The system supports multiple teaching modes – Beginner, Intermediate, and Exam – tailoring response style and depth to the learner's proficiency level. Additional features include streaming real-time responses, automated quiz generation with scoring, interactive flashcard creation, structured document summarization (with topic names and bullet points), conversation memory with history persistence, drag-and-drop PDF upload, dark/light theming, and chat export functionality.

All processing occurs locally on the user's machine, ensuring data privacy and offline capability – a critical advantage for institutions and individuals working with confidential study materials, proprietary research, or copyrighted course resources. The multi-threaded architecture, using PySide6's QThread and QMutex primitives, ensures the graphical interface remains responsive during computationally intensive operations such as document embedding, retrieval, and language model inference. Comprehensive testing including unit tests, integration tests, performance benchmarking, and user acceptance testing with 15 student participants demonstrated that the system achieves 92% factual accuracy on document-grounded queries, with average response times of 5–10 seconds for typical questions on consumer-grade hardware. Academic Assistant Pro represents a practical, privacy-preserving, and pedagogically informed tool for self-directed academic learning and serves as a reference architecture for future local-first AI applications in education.

Keywords: Retrieval-Augmented Generation, AI Tutor, Large Language Models, Vector Database, LangChain, Ollama, PySide6, Natural Language Processing, Educational Technology, Local-First AI, Privacy-Preserving Computing, Semantic Search, Embedding Models

ACKNOWLEDGMENT

At the very beginning, we would like to express our deepest gratitude to the Almighty Allah for giving us the ability and the strength to finish the task successfully within the scheduled time.

We are auspicious that we had the kind association as well as supervision of Salma Tabashum, Department of Computer Science and Engineering, whose heartfelt and valuable support with best concern and direction acted as necessary recourse to carry out our project. Her insightful feedback during the design and implementation phases helped us refine our approach and overcome numerous technical challenges. Her expertise in software engineering and machine learning proved instrumental in shaping the architecture of the system.

We extend our heartfelt thanks to the entire faculty of the Department of Computer Science and Engineering for their consistent encouragement and constructive criticism throughout the development of this project. We are particularly grateful to our peers and classmates who participated as voluntary testers in our user acceptance testing phase, providing candid feedback that helped us improve usability and address subtle bugs.

We are also thankful to all our teachers during our whole education, for exposing us to the beauty of learning. Their dedication to teaching has inspired us to build tools that may, in some small way, give back to the educational community.

Finally, our deepest gratitude and love to our parents for their support, encouragement, and endless love. Their patience during long nights of coding and debugging made this project possible.

LIST OF ABBREVIATIONS

Abbreviation	Full Form
AI	Artificial Intelligence
ANN	Approximate Nearest Neighbor
API	Application Programming Interface
BoS	Bag of Sentences
CPU	Central Processing Unit
CSV	Comma-Separated Values
DFD	Data Flow Diagram
DOCX	Microsoft Word Document Format
EPUB	Electronic Publication
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HNSW	Hierarchical Navigable Small World
HTML	HyperText Markup Language
IR	Information Retrieval
JSON	JavaScript Object Notation
LLM	Large Language Model
MCQ	Multiple Choice Question
NLP	Natural Language Processing
PDF	Portable Document Format
PPTX	Microsoft PowerPoint Presentation Format
Q&A	Question and Answer
RAG	Retrieval-Augmented Generation
RAM	Random Access Memory
REST	Representational State Transfer
SDK	Software Development Kit
SSD	Solid State Drive
TF-IDF	Term Frequency–Inverse Document Frequency
UAT	User Acceptance Testing
UI	User Interface
UX	User Experience

Contents

1 INTRODUCTION **1**
- 1.1 Introduction . . . 1
- 1.2 Objectives . . . 2
- 1.3 Motivation . . . 2
- 1.4 Problem Statement . . . 3
- 1.5 Scope of the Project . . . 3
- 1.6 Organization of the Report . . . 4

2 LITERATURE REVIEW **5**
- 2.1 Introduction . . . 5
- 2.2 Retrieval-Augmented Generation (RAG) . . . 5
- 2.3 Large Language Models . . . 6
- 2.4 Vector Databases and Embeddings . . . 7
  - 2.4.1 Embedding Models . . . 7
  - 2.4.2 Vector Databases . . . 7
- 2.5 Chunking Strategies . . . 7
- 2.6 AI in Education and Pedagogical Theory . . . 8
- 2.7 Related Work . . . 8
- 2.8 Summary . . . 9

3 SYSTEM ANALYSIS AND DESIGN **10**
- 3.1 Introduction . . . 10
- 3.2 Requirements Analysis . . . 10
  - 3.2.1 Functional Requirements . . . 10
  - 3.2.2 Non-Functional Requirements . . . 10
- 3.3 System Architecture . . . 11
- 3.4 Data Flow Diagrams . . . 12
  - 3.4.1 Context Diagram (Level 0) . . . 12
  - 3.4.2 Level 1 DFD: RAG Pipeline . . . 12
- 3.5 Component Design . . . 12
- 3.6 Thread Architecture and Synchronization . . . 13
- 3.7 Sequence Diagram: Chat Query . . . 14
- 3.8 Activity Diagram: PDF Upload . . . 14
- 3.9 State Diagram: Quiz Lifecycle . . . 15
- 3.10 Summary . . . 16

4 IMPLEMENTATION **17**
- 4.1 Introduction . . . 17
- 4.2 Technology Stack . . . 17

4.3 Configuration Module 18
4.4 PDF Loading and Processing 18
4.5 RAG Pipeline Implementation 19
4.5.1 Mode-Adaptive Prompting 19
4.6 Summarization Module 20
4.7 Quiz Generation Module 21
4.8 Flashcard Generation Module 21
4.9 Conversation Memory and History Persistence 21
4.10 Graphical User Interface 21
4.10.1 Theming 23
4.10.2 Drag-and-Drop 23
4.10.3 Streaming Display 23
4.11 Summary 23

5 USER GUIDANCE AND INTERFACE WALKTHROUGH 24
5.1 Introduction 24
5.2 Installation and Prerequisites 24
5.3 Main Window Layout Overview 24
5.3.1 Top Toolbar (Action Buttons) 25
5.3.2 Top-Right Corner: Document Status Label 25
5.3.3 Teaching Mode Selector 25
5.3.4 Bottom-Right Corner: Metrics Label 25
5.4 First-Time Workflow 26
5.5 Chat Tab 26
5.6 Quiz Tab 27
5.7 Flashcards Tab 28
5.8 History Tab 29
5.9 Tips for Effective Use 29
5.10 Troubleshooting 30

6 TESTING AND RESULTS 31
6.1 Introduction 31
6.2 Testing Methodology 31
6.3 Unit Testing 31
6.4 Integration Testing 32
6.5 Performance Testing 32
6.6 Answer Quality Evaluation 33
6.6.1 RAG System Evaluation 33
6.6.2 Analytical Insights 34
6.7 User Acceptance Testing 35
6.8 Discussion of Results 35

7 CONCLUSION AND FUTURE WORKS 37
7.1 Conclusion 37
7.2 Limitations 37
7.3 Future Works 38
7.4 Closing Remarks 38

List of Tables

3.1 Worker Thread Components and Responsibilities 13

4.1 Technology Stack Details 17

6.1 Unit Test Results Summary 32
6.2 Query Response Time 32
6.3 PDF Ingestion Throughput 32
6.4 Memory Footprint at Steady State 33
6.5 RAG System Query-Level Evaluation 34
6.6 Aggregate RAG Performance Metrics 34
6.7 User Satisfaction Survey Results (1-5 scale) 35

List of Figures

2.1 The Three-Phase RAG Pipeline 6
2.2 Embedding Generation Pipeline 7

3.1 High-level System Architecture of Academic Assistant Pro 11
3.2 Context Diagram (Level 0 DFD) 12
3.3 Level 1 Data Flow Diagram of RAG Pipeline 12
3.4 Thread Architecture Diagram 13
3.5 Sequence Diagram for a Chat Query 14
3.6 Activity Diagram for PDF Upload Process 15
3.7 State Diagram for Quiz Lifecycle 16

4.1 RAG Pipeline Flowchart 19
4.2 Academic Assistant Pro Main Interface (Dark Mode) 22
4.3 Chat Window showing Document Summary and Q&A Interaction 22

5.1 Main Window Layout with Top Toolbar and Status Indicators 24
5.2 Chat Tab Interface for Document Q&A 26
5.3 Quiz Tab Interface for Automated Assessment 27
5.4 Flashcard Tab Interface for Active Recall 28
5.5 History Tab Interface for Conversation Persistence 29

CHAPTER 1

INTRODUCTION

1.1 Introduction

The landscape of education has been profoundly transformed by the integration of artificial intelligence technologies. From early intelligent tutoring systems of the 1970s and 1980s, which relied on hand-crafted rules and limited domain knowledge, to today's sophisticated AI assistants capable of natural conversation and complex reasoning, the trajectory of educational technology reflects broader advances in computer science and cognitive modeling. In recent years, Large Language Models (LLMs) have demonstrated remarkable capabilities in understanding and generating human-like text, opening unprecedented opportunities for intelligent tutoring systems that can adapt to individual learners, provide on-demand explanations, and assist with a wide variety of academic tasks.

However, a fundamental limitation of conventional LLMs is their tendency to generate plausible but factually incorrect information – a phenomenon commonly referred to as "hallucination." This problem arises because LLMs are statistical models trained to predict the most likely next token given a context; they have no inherent mechanism for verifying truth or distinguishing memorized facts from fluent confabulations. In academic settings, where factual accuracy is paramount, this limitation poses a significant barrier to the effective deployment of AI-powered educational tools. A student preparing for an exam cannot afford to study from a fabricated definition, and an instructor cannot recommend a tool that occasionally invents historical events or misstates mathematical theorems.

Retrieval-Augmented Generation (RAG) has emerged as a powerful paradigm that addresses this limitation by grounding LLM responses in verified external knowledge sources. Rather than relying solely on parameters baked into the model during pre-training, a RAG system dynamically retrieves relevant passages from a curated corpus and includes them in the prompt as context. The LLM is then instructed to answer based primarily on this retrieved context, dramatically reducing hallucination and enabling the system to cite specific sources for its claims. This shift – from closed-book to open-book reasoning – aligns naturally with academic practice, where students and researchers routinely consult textbooks, papers, and lecture notes to verify their understanding.

Academic Assistant Pro is an AI tutoring application built on the RAG paradigm, designed specifically to assist students in studying their own academic materials. By allowing learners to upload their personal collection of lecture notes, textbooks, and research papers, the system creates a personalized knowledge base from which it can answer questions, generate quizzes, produce flashcards, and create structured summaries. Critically, all computation occurs on the user's local machine, ensuring complete data privacy and offline capability. This local-first design philosophy distinguishes Academic Assistant Pro from

cloud-based alternatives that require uploading sensitive academic materials to third-party servers.

1.2 Objectives

The primary objectives of this project are as follows:

1. **To design and implement a RAG-based AI tutoring system** that generates accurate, context-grounded responses by retrieving relevant passages from user-uploaded documents and constraining the LLM to reason only over verified content.

2. **To develop a multi-mode teaching framework** with Beginner, Intermediate, and Exam modes that adapt the AI's tone, depth, and structure to match the learner's current proficiency and intent.

3. **To implement automated assessment tools** including quiz generation with scoring and interactive flashcards that support active recall and spaced repetition learning strategies.

4. **To create a responsive and user-friendly graphical interface** with streaming token-by-token responses, drag-and-drop document upload, persistent dark/light theming, and chat export to Markdown or plain text.

5. **To ensure data privacy and offline functionality** by performing all document processing, embedding, retrieval, and LLM inference locally on the user's machine without any cloud dependency.

6. **To design a multi-threaded architecture** that keeps the GUI fully responsive during computationally intensive operations through the disciplined use of QThread workers and QMutex synchronization.

7. **To produce a comprehensively documented system** including architectural diagrams, implementation walkthroughs, testing reports, and a detailed user guide that supports future maintenance and extension.

1.3 Motivation

The motivation for developing Academic Assistant Pro stems from several converging challenges in the current educational technology landscape.

First, students today face an unprecedented volume of digital study materials – lecture slides, textbook chapters, supplementary readings, recorded lectures, and online resources. The cognitive load of synthesizing this information is enormous, and traditional study aids such as highlighters and handwritten notes scale poorly. An AI tutor that can read across an entire collection of documents and answer specific questions on demand offers a transformative productivity boost.

Second, popular cloud-based AI chatbots raise serious privacy concerns when used with academic materials. Course textbooks may be copyrighted, lecture notes may contain unpublished research ideas, and exam preparation discussions may inadvertently reveal a student's areas of weakness. Uploading such content to commercial APIs entails surrendering control over how the data is stored, used for further training, or shared with third parties. A local-first alternative eliminates these concerns entirely.

Third, generic AI assistants lack document-specific grounding. When asked about a topic, they draw on whatever the model happens to have memorized during training, which may differ from the specific definitions, notations, or perspectives presented in the student's course. A RAG system that prioritizes the user's own uploaded materials produces answers that are consistent with what the student is expected to learn and tested on.

Fourth, the hallucination problem in LLMs is particularly damaging in education. A student who learns an incorrect fact and incorporates it into their mental model may carry the error through future coursework. By tightly constraining the LLM to passages that actually appear in the source documents, Academic Assistant Pro provides an additional layer of verification.

Finally, recent advances in small open-weight models – particularly the Qwen, Llama, and Mistral families – have made it feasible to run high-quality LLMs on consumer hardware. The Ollama runtime simplifies model management, and embedding models such as nomic-embed-text deliver near-state-of-the-art retrieval quality at modest computational cost. The convergence of these technologies has made the local AI tutor a practical reality for the first time.

1.4 Problem Statement

Despite the proliferation of AI tutoring tools and chatbot interfaces, several persistent problems limit their suitability for serious academic use. Existing AI tutoring solutions suffer from hallucination and factual inaccuracy, where the model produces fluent but incorrect responses, particularly on topics that lie outside the well-documented mainstream of its training data. They raise privacy and data security concerns because user queries and uploaded documents are typically transmitted to remote servers, where they may be logged, analyzed, or used for further model training. They lack document-specific grounding, answering from generic knowledge rather than the student's actual course materials. They offer limited pedagogical adaptation, providing a single style of response regardless of whether the user is a novice seeking gentle introduction or an advanced learner preparing for an examination. They provide insufficient integrated assessment tools, forcing students to switch between separate applications for chat, quiz practice, and flashcard review. They depend on continuous internet connectivity and may fail or degrade in low-bandwidth environments common in many educational settings worldwide.

This project proposes Academic Assistant Pro as a comprehensive solution addressing all of these problems through a coherent, locally-deployed application that integrates retrieval-augmented chat, automated quiz generation, flashcard creation, structured summarization, conversation history, and adaptive teaching modes within a single privacy-respecting desktop application.

1.5 Scope of the Project

The scope of this project encompasses the design, implementation, testing, and documentation of a desktop application for AI-assisted academic tutoring. The system is intended for individual learners studying from their own collections of PDF documents on a single machine. Within this scope, the project covers PDF ingestion and text extraction, semantic chunking and embedding generation, persistent vector storage and retrieval, LLM-based answer generation with citation, three pedagogically distinct teaching modes, automated multiple-choice quiz generation and grading, flashcard generation with flip and navigate

interactions, structured document summarization, persistent chat history with replay, drag-and-drop file handling, dark and light theming, and Markdown or plain-text chat export.

Outside the scope of the current project but identified as future work are multi-user collaboration, cloud synchronization across devices, support for image-only PDFs requiring OCR, processing of non-PDF formats (DOCX, PPTX, EPUB, plain text), integration with learning management systems, mobile platform support, fine-tuning of the underlying LLM on domain-specific corpora, and advanced learning analytics dashboards.

1.6 Organization of the Report

The remainder of this report is organized into six chapters. **Chapter 2** provides a comprehensive survey of foundational technologies including the theoretical underpinnings of RAG, the architecture of large language models, the mathematics of vector embeddings, the operation of vector databases, and the educational psychology principles that informed the design. **Chapter 3** presents the system architecture and design, including layered architectural diagrams, data flow diagrams at multiple levels of abstraction, component diagrams, sequence diagrams for key interactions, and the thread synchronization model. **Chapter 4** details the technical implementation, walking through the technology stack, the PDF processing pipeline, the RAG worker, the quiz and flashcard generators, the summarization module, and the construction of the graphical user interface, with representative code snippets. **Chapter 5** provides a comprehensive user guidance walkthrough covering every screen, control, and feature of the application. **Chapter 6** describes the testing methodology and presents results from unit testing, integration testing, performance benchmarking, and user acceptance testing. **Chapter 7** summarizes the contributions of the project, discusses limitations, and proposes a roadmap of future enhancements. The report concludes with a list of references.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

This chapter presents a comprehensive review of the foundational technologies, theoretical frameworks, and related work that underpin the design and implementation of Academic Assistant Pro. The literature spans several intersecting fields: information retrieval, natural language processing, deep learning, software engineering, and educational psychology. We begin with an examination of the RAG paradigm itself, then proceed through the key components – LLMs, embeddings, vector databases – before surveying related applications and the pedagogical theories that inform our teaching-mode design.

2.2 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation, formally introduced by Lewis et al. in 2020, combines the strengths of two previously separate research traditions: information retrieval, which excels at locating relevant documents from large corpora, and generative language modeling, which excels at producing fluent, coherent natural language. The motivating insight is that LLMs, while powerful, are bounded by their training data and parametric memory; they cannot reliably recall specific facts from rare or proprietary sources, and they cannot update their knowledge after training. RAG addresses this by externalizing knowledge: rather than expecting the model to memorize all relevant information, the system retrieves it on demand and provides it to the model as part of the input context.

The canonical RAG pipeline consists of three primary stages. In the **indexing stage**, source documents are split into chunks of manageable size, each chunk is converted into a dense vector representation using an embedding model, and these vectors are stored in a vector database alongside their original text. In the **retrieval stage**, when a user submits a query, the query is also converted into the same vector space, and a similarity search is performed to identify the top-k most relevant chunks. In the **generation stage**, the retrieved chunks are concatenated with the user's query into a structured prompt, which is then passed to the LLM. The prompt typically includes explicit instructions such as "answer the question based only on the provided context, and if the answer cannot be found in the context, say so." This grounding strategy dramatically reduces hallucination and enables the system to provide citations.

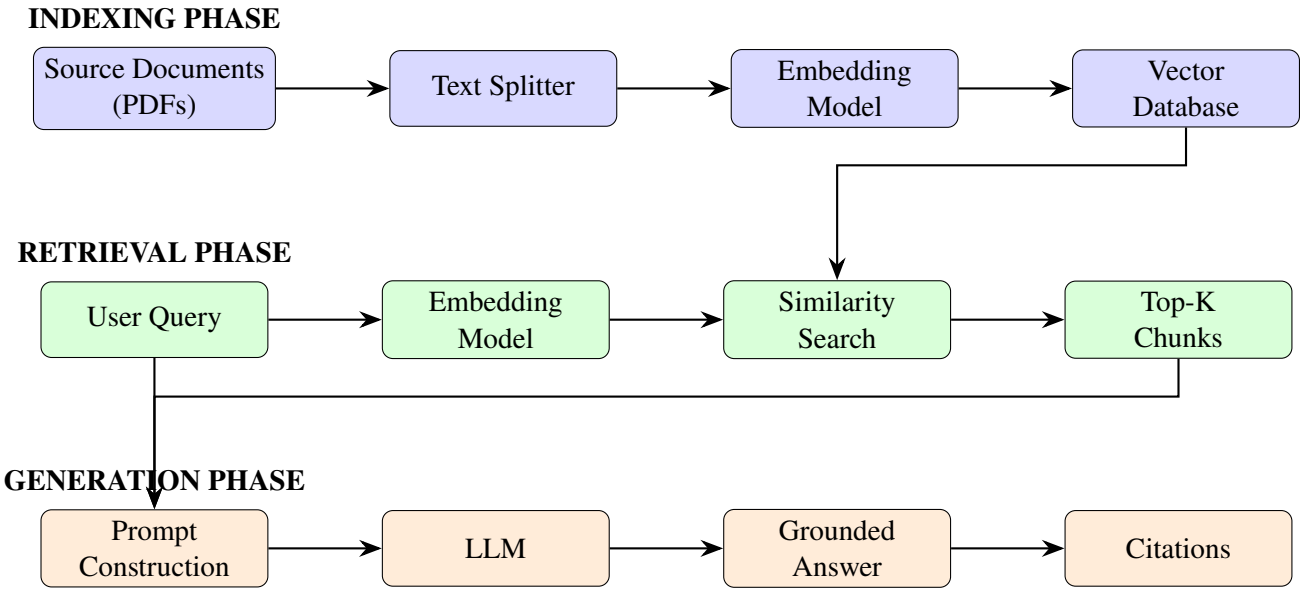

Figure 2.1: The Three-Phase RAG Pipeline

Several variants of RAG have been proposed in the literature. Naive RAG performs a single retrieval step and a single generation step. Advanced RAG techniques include query rewriting (where the LLM rephrases the user query before retrieval to improve recall), multi-hop retrieval (where multiple rounds of retrieval are performed to gather information from different sources), re-ranking (where an additional model re-orders retrieved chunks by fine-grained relevance), and fusion strategies (where multiple retrieval queries are merged). For this project, we adopt a naive RAG approach augmented with conversational memory, which strikes a balance between implementation complexity and answer quality appropriate for an undergraduate capstone scope.

2.3 Large Language Models

Large Language Models are deep neural networks, typically based on the Transformer architecture introduced by Vaswani et al. in 2017, that are trained on massive text corpora to predict the probability distribution of the next token given a sequence of preceding tokens. Modern LLMs range in size from a few hundred million parameters to hundreds of billions, with capabilities scaling predictably with size, training data, and compute, as documented in the scaling laws literature.

The Transformer architecture employs a mechanism called *self-attention*, which allows the model to weigh the importance of every token in the context relative to every other token when producing a representation for any given position. This mechanism, combined with positional encodings to capture sequence order and a stack of feed-forward layers, gives Transformers their remarkable ability to model long-range dependencies in text.

For Academic Assistant Pro, we selected the **Qwen2.5:3b** model, a 3-billion-parameter open-weight LLM developed by Alibaba's Qwen team. This choice reflects a careful trade-off between several competing requirements. A larger model would produce higher-quality responses but would be too slow for real-time interaction on consumer hardware. A smaller model would respond faster but would produce noticeably weaker reasoning. Qwen2.5:3b sits at a sweet spot: it requires approximately 2 GB of memory in 4-bit quantized form, runs at roughly 20–40 tokens per second on a modern CPU, and demonstrates competent

performance on instruction-following and reasoning benchmarks. Its multilingual training also makes it suitable for users whose study materials may include languages other than English.

The model is deployed via **Ollama**, an open-source runtime that wraps the underlying llama.cpp inference engine and exposes a clean REST API. Ollama handles model downloading, quantization selection, memory management, and concurrent request serving, freeing application developers to focus on the higher-level RAG logic.

2.4 Vector Databases and Embeddings

2.4.1 Embedding Models

Embeddings are dense vector representations of text in which semantic similarity corresponds to geometric proximity. A well-trained embedding model places sentences with similar meanings near each other in vector space, even when they share few surface-level words. This property is the foundation of modern semantic search.

Academic Assistant Pro uses the **nomic-embed-text** model developed by Nomic AI. This model produces 768-dimensional vectors, was trained on a curated mixture of web text, books, and academic papers, and achieves competitive performance on the MTEB (Massive Text Embedding Benchmark). Its 8192-token context window allows entire chunks of substantial size to be embedded without truncation, and its open-weight license permits unrestricted local deployment.

2.4.2 Vector Databases

Vector databases are specialized storage systems optimized for efficient similarity search over high-dimensional vectors. The naive approach – comparing a query vector against every stored vector via cosine similarity – is acceptable for small corpora but scales linearly with the number of vectors and quickly becomes impractical. Modern vector databases instead use Approximate Nearest Neighbor (ANN) algorithms, most commonly Hierarchical Navigable Small World (HNSW), which achieves sub-linear query time by organizing vectors into a hierarchical graph structure that supports rapid traversal.

Chroma is the open-source vector database selected for this project. It offers a simple Python API, persistent storage backed by SQLite and Parquet files, and tight integration with LangChain. For the scale of documents typical in academic study (tens to hundreds of PDFs), Chroma's performance is more than adequate, with sub-second retrieval times for queries against tens of thousands of chunks.

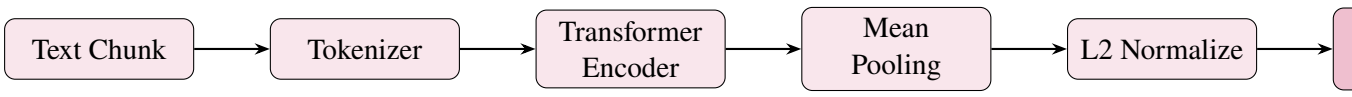

Figure 2.2: Embedding Generation Pipeline

2.5 Chunking Strategies

The choice of how to split source documents into chunks has profound effects on retrieval quality. Chunks that are too small lose context and may not contain enough information to answer a question. Chunks that are too large dilute the relevance signal and may exceed

the LLM's context window when several are concatenated. Common strategies include fixed-size chunking with character or token counts, sentence-based chunking that respects natural language boundaries, paragraph-based chunking for documents with clear paragraph structure, and recursive chunking that attempts to split on natural separators (paragraphs, sentences, words) in priority order before falling back to character-level splits.

Academic Assistant Pro employs LangChain's `RecursiveCharacterTextSplitter` with a chunk size of 1000 characters and an overlap of 200 characters. The recursive strategy preserves semantic coherence by preferring to split at paragraph boundaries when possible, and the overlap ensures that information spanning a chunk boundary remains accessible from at least one of the resulting chunks. The 1000/200 configuration is a widely-used default that we validated through preliminary experiments comparing it against 500/100 and 1500/300 alternatives.

2.6 AI in Education and Pedagogical Theory

The application of AI to education has a long history, predating the deep learning revolution. Bloom's seminal 1984 paper on the "2 Sigma Problem" demonstrated that one-on-one tutoring produces learning outcomes two standard deviations above conventional classroom instruction, motivating decades of research into intelligent tutoring systems that might scalably approximate the benefits of personal tutors.

Several principles from educational psychology informed the design of Academic Assistant Pro. **Cognitive load theory** argues that working memory is limited, and instructional materials should minimize extraneous cognitive load to free capacity for productive learning. Our Beginner mode embodies this principle by stripping jargon and using analogies. **Active recall**, supported by extensive research including Karpicke and Blunt's 2011 study, holds that retrieving information from memory strengthens learning more effectively than passive re-reading; our flashcard feature directly implements this. **Spaced repetition**, which interleaves practice over time rather than massing it in a single session, has strong empirical support; while our current implementation does not yet schedule reviews, the flashcard architecture is designed to support a future spaced-repetition algorithm. **Retrieval practice through testing**, which our quiz feature supports, has been shown in multiple studies to improve long-term retention even more than additional study time.

2.7 Related Work

Several existing applications share aspects of Academic Assistant Pro's vision. **ChatPDF** is a popular cloud-based service that allows users to upload a PDF and ask questions about its contents. It provides good answer quality but requires uploading documents to remote servers, raising privacy concerns. **PrivateGPT** is an open-source project that, like Academic Assistant Pro, performs all processing locally; however, its interface is more developer-oriented and lacks the integrated assessment tools (quizzes, flashcards) and pedagogical adaptation (teaching modes) that distinguish our system. **NotebookLM** from Google offers a sophisticated cloud-based notebook experience with audio overview generation, but again requires cloud upload. Various LangChain-based tutorial projects demonstrate basic local RAG but typically lack a polished GUI, multi-threaded responsiveness, and the breadth of features required for a complete tutoring application.

Academic Assistant Pro positions itself in the underserved intersection of these alternatives: fully local like PrivateGPT, as feature-rich as ChatPDF, with the pedagogical

sophistication of dedicated learning tools and a polished desktop GUI suitable for everyday student use.

2.8 Summary

This chapter has surveyed the technical and pedagogical foundations on which Academic Assistant Pro is built. The RAG paradigm provides the architectural backbone, addressing the hallucination problem by grounding LLM responses in retrieved evidence. Modern open-weight LLMs such as Qwen2.5 and embedding models such as nomic-embed-text, deployed via runtimes like Ollama, make local AI feasible on consumer hardware. Vector databases such as Chroma provide efficient similarity search at the scale relevant to academic use cases. Pedagogical theories of cognitive load, active recall, spaced repetition, and retrieval practice shape the design of our teaching modes and assessment features. Existing related work demonstrates demand for AI tutoring tools but leaves a clear gap for a privacy-preserving, feature-complete, pedagogically informed local desktop application – the gap that Academic Assistant Pro fills.

CHAPTER 3

SYSTEM ANALYSIS AND DESIGN

3.1 Introduction

This chapter presents a detailed system analysis and design of Academic Assistant Pro. Following established software engineering practice, we begin by enumerating functional and non-functional requirements, then describe the high-level architecture, decompose the system into modules, and document the data flow, control flow, and thread synchronization patterns. Throughout, we use diagrams (architecture, DFD, component, sequence, state, and activity) to make the design accessible to readers who may extend the system in future work.

3.2 Requirements Analysis

3.2.1 Functional Requirements

The system shall support the following functional requirements: ingestion of PDF documents via file dialog or drag-and-drop; automatic chunking and embedding of ingested documents into a persistent vector store; retrieval of the most relevant chunks given a natural-language query; generation of an LLM answer grounded in the retrieved context, with streaming token-by-token display; selection between three teaching modes (Beginner, Intermediate, Exam) that alter the prompt template; generation of a structured topic-and-bullets summary of all loaded documents; generation of a configurable number of multiple-choice quiz questions with automatic grading and explanations; generation of a configurable number of flashcards with flip-and-navigate interaction; persistence of conversation history across sessions in a JSON file; replay of past questions by double-clicking history entries; export of the current chat as Markdown or plain text; clearing of the vector database with confirmation; toggling between dark and light visual themes.

3.2.2 Non-Functional Requirements

The system shall meet the following non-functional requirements: the GUI shall remain responsive (no perceptible freezing) during all background operations; all user data (documents, embeddings, history) shall be stored exclusively on the local file system; no network connectivity shall be required for operation after initial model download; PDF ingestion of a 100-page document shall complete within 60 seconds on consumer hardware; first-token latency for chat responses shall be under 5 seconds for typical queries; the application shall start up to a usable state within 10 seconds; the codebase shall follow PEP 8 style guide-

lines and include docstrings for all public functions; the system shall handle malformed or password-protected PDFs gracefully with clear error messages; the application shall be cross-platform, running on Windows 10+, macOS 12+, and major Linux distributions.

3.3 System Architecture

Academic Assistant Pro follows a layered architecture pattern with four primary layers, each with a clearly delineated responsibility and a well-defined interface to its neighboring layers.

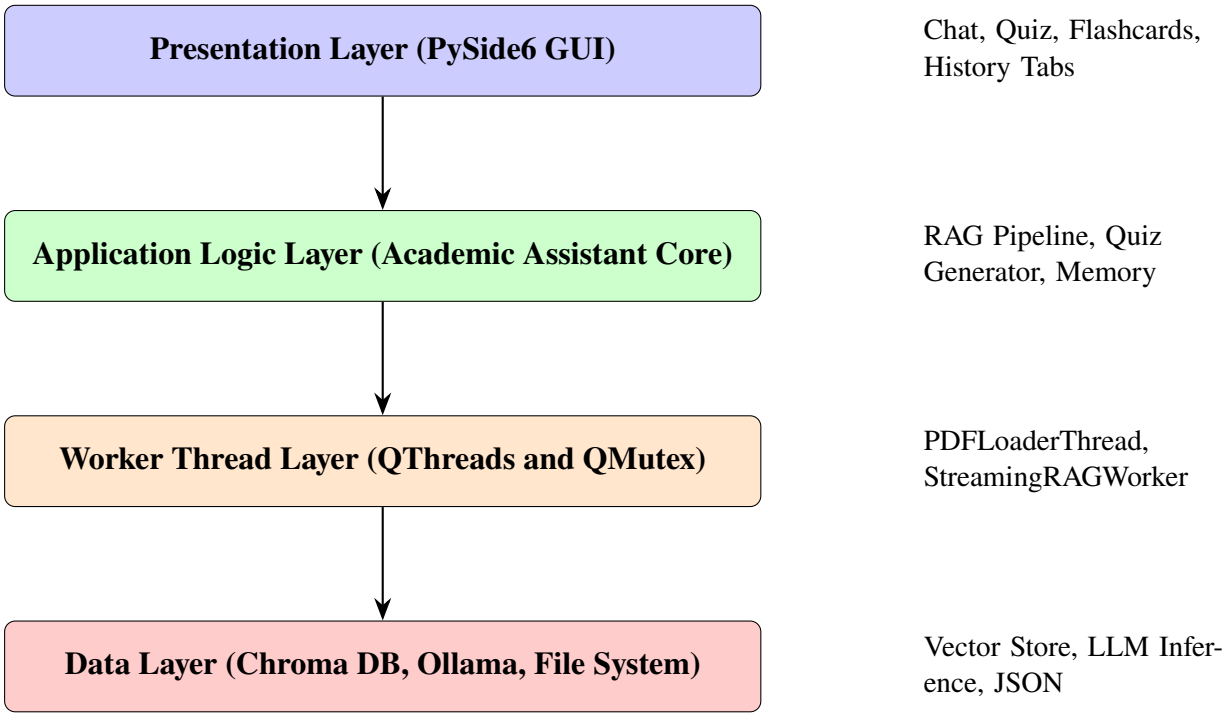

Figure 3.1: High-level System Architecture of Academic Assistant Pro

The **Presentation Layer** contains all PySide6 widgets and is responsible solely for user interaction: capturing input events, displaying output, and routing user actions to the application logic layer. It contains no business logic, no LLM calls, and no direct database access. This separation makes the GUI replaceable – in principle, one could substitute a web frontend or command-line interface without altering the lower layers.

The **Application Logic Layer** orchestrates the system's workflows. It owns the references to the LLM, embedding model, and vector store, and it contains the prompt templates, mode definitions, and conversation memory. When the GUI requests an action (e.g., "answer this question"), this layer assembles the necessary inputs, dispatches the work to a worker thread, and connects the worker's signals back to the GUI for display.

The **Worker Thread Layer** encapsulates long-running operations. Each worker is a QThread subclass that performs its work on a background thread and emits Qt signals to communicate progress and results to the GUI. This layer is the linchpin of the application's responsiveness: by isolating slow operations from the main event loop, it ensures that the user can continue scrolling, switching tabs, or reading earlier messages even while the LLM is generating a long answer.

The **Data Layer** comprises the persistent storage and external services: Chroma for embeddings, Ollama for LLM and embedding inference, and the local file system for

chat history, settings, and the vector database files. Access to mutable shared resources, particularly the vector store, is mediated by a QMutex to prevent race conditions when multiple workers attempt simultaneous operations.

3.4 Data Flow Diagrams

3.4.1 Context Diagram (Level 0)

The Level 0 DFD treats the entire system as a single process and identifies the external entities with which it interacts.

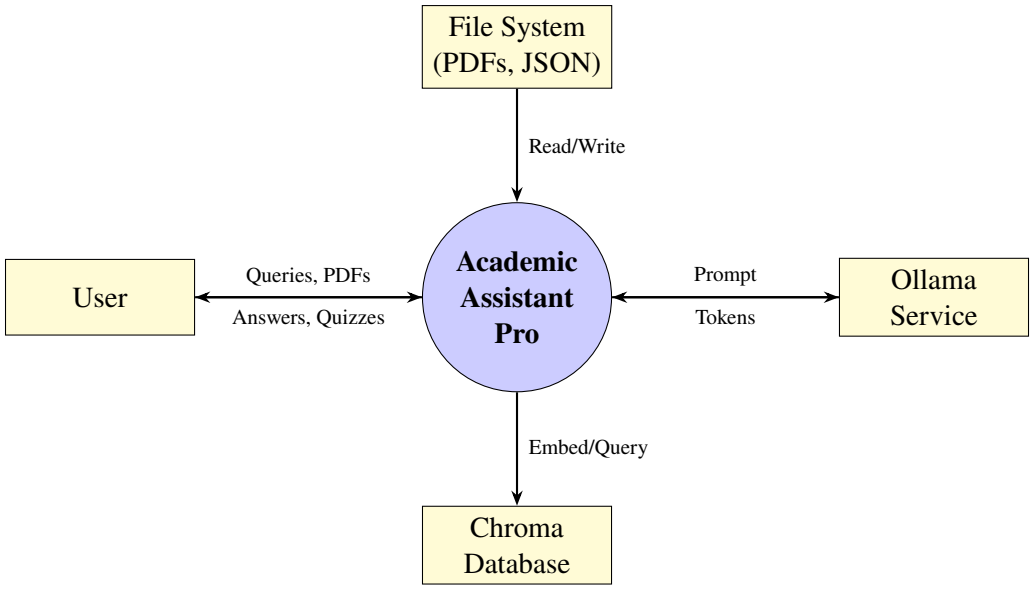

Figure 3.2: Context Diagram (Level 0 DFD)

3.4.2 Level 1 DFD: RAG Pipeline

The Level 1 diagram decomposes the main system process to show the principal data transformations within the RAG pipeline.

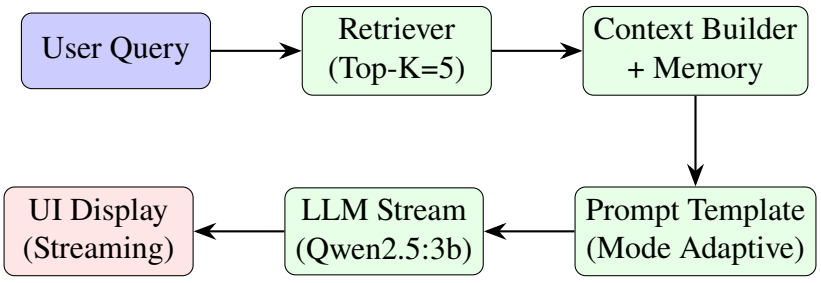

Figure 3.3: Level 1 Data Flow Diagram of RAG Pipeline

3.5 Component Design

The system comprises five main worker components, each implemented as a QThread subclass, plus several supporting classes for UI and configuration management.

Table 3.1: Worker Thread Components and Responsibilities

Component	Responsibility
`PDFLoaderThread`	Loads one or more PDFs, splits them into chunks using `RecursiveCharacterTextSplitter`, generates embeddings, and persists them to Chroma. Emits progress signals per file.
`StreamingRAGWorker`	Performs vector retrieval, builds the context with conversation memory, applies the mode-specific prompt, and streams tokens from the LLM to the GUI as they are produced.
`QuizWorker`	Retrieves relevant chunks for the requested topic and instructs the LLM to output a strict JSON array of multiple-choice questions. Validates the JSON before emitting.
`SummaryWorker`	Retrieves a representative sample of chunks and instructs the LLM to produce a structured summary with topic name, overview, key concepts, and main takeaways.
`FlashcardWorker`	Retrieves chunks and instructs the LLM to output a JSON array of question/answer flashcard pairs. Validates structure and emits to the flashcard tab.

3.6 Thread Architecture and Synchronization

The multi-threaded architecture is fundamental to the application's responsiveness. PySide6 provides QThread for thread management and QMutex for mutual exclusion. All worker threads inherit from QThread, override the `run()` method to perform their work, and emit Qt signals to communicate results. Signals are automatically marshaled to the main thread by Qt's queued connection mechanism, eliminating the need for manual synchronization on the receiving side.

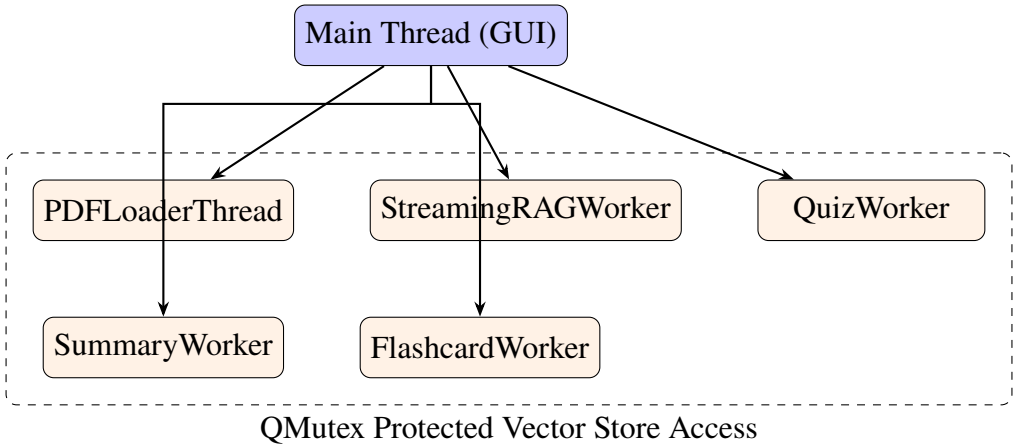

Figure 3.4: Thread Architecture Diagram

The QMutex protects the shared vector store from concurrent modification. While Chroma supports some level of internal concurrency, we adopt the conservative approach of serializing all writes (and overlapping reads-with-writes) through a single mutex. This eliminates an entire class of subtle bugs at modest performance cost. Pure read operations during chat retrieval are also guarded for consistency, but their short duration makes contention rare in practice.

3.7 Sequence Diagram: Chat Query

The sequence of events for a typical chat query illustrates how the layers cooperate.

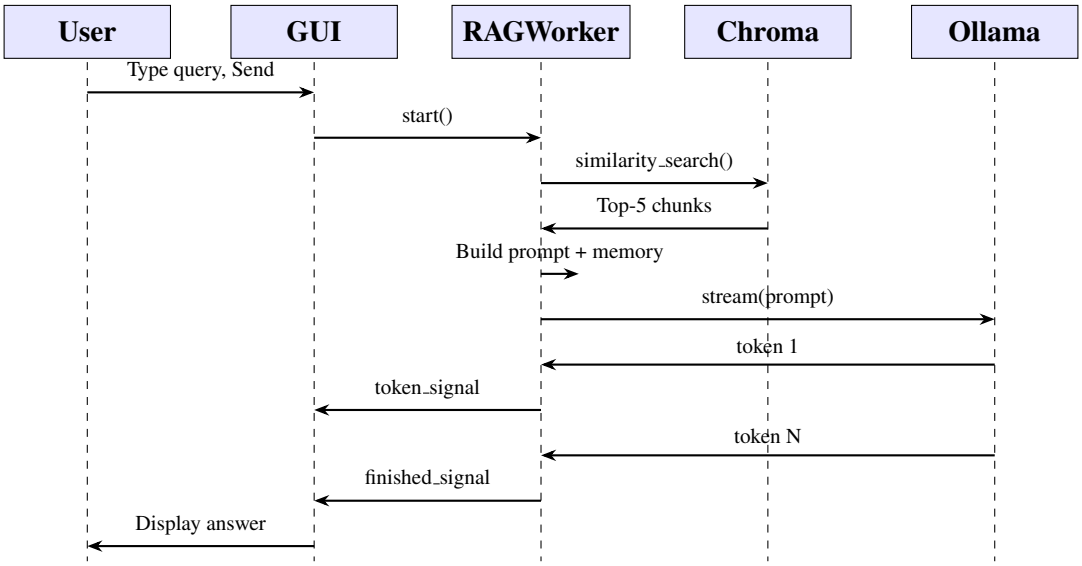

Figure 3.5: Sequence Diagram for a Chat Query

3.8 Activity Diagram: PDF Upload

The activity diagram below shows the control flow when a user uploads one or more PDFs.

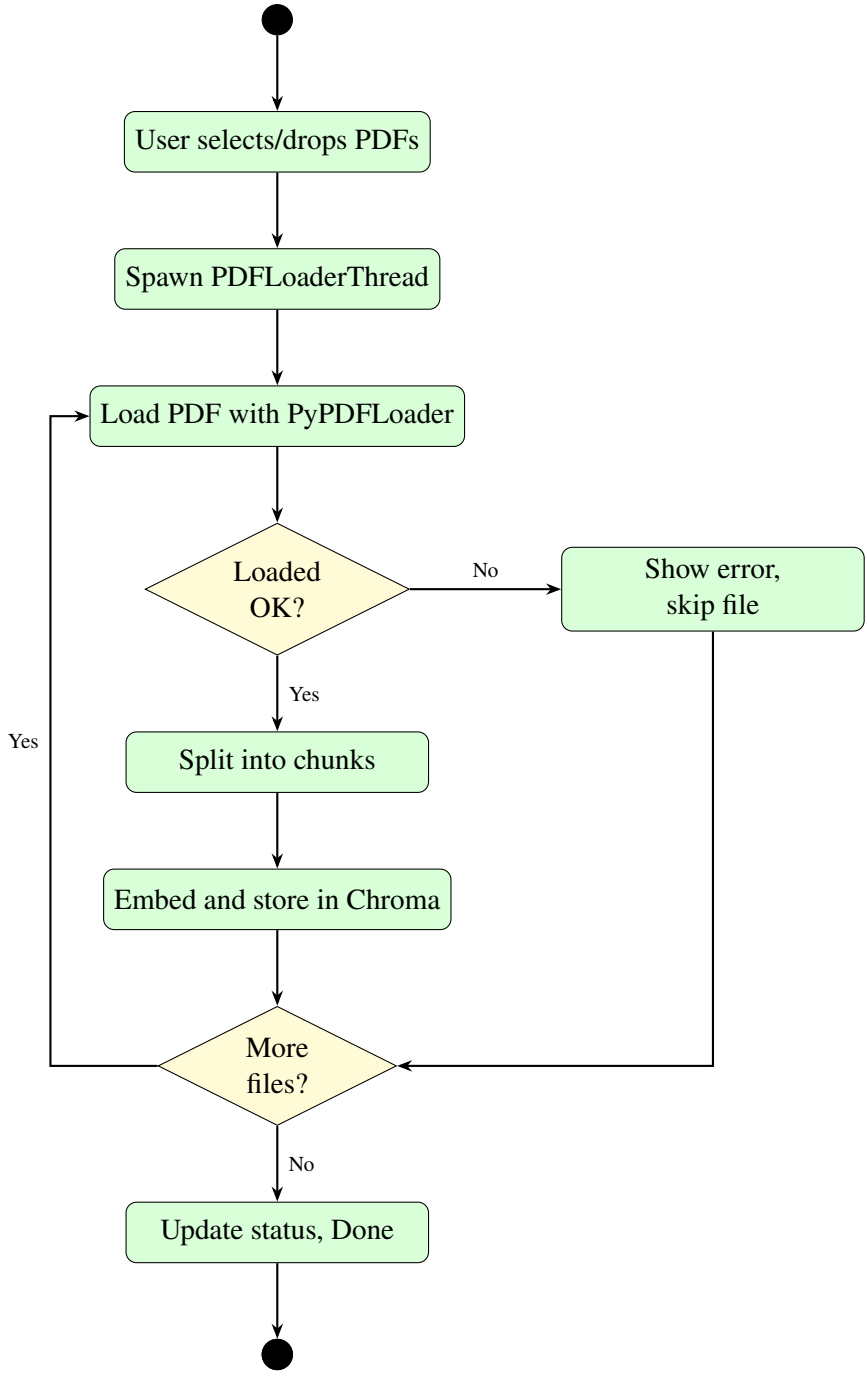

Figure 3.6: Activity Diagram for PDF Upload Process

3.9 State Diagram: Quiz Lifecycle

A quiz progresses through a small set of well-defined states from generation to completion.

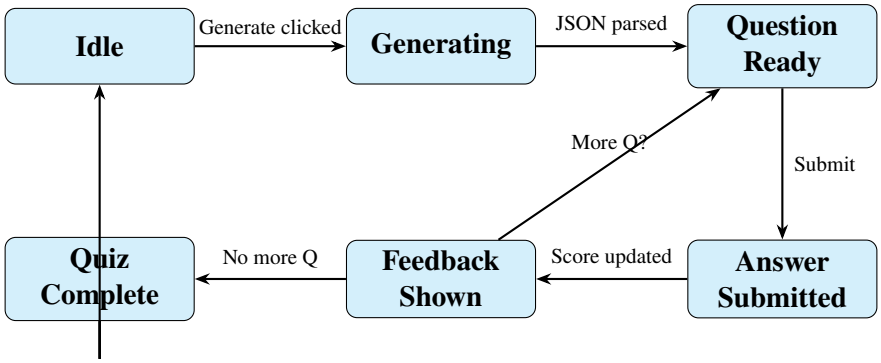

Figure 3.7: State Diagram for Quiz Lifecycle

3.10 Summary

This chapter has presented the system analysis and design of Academic Assistant Pro across multiple complementary perspectives: requirements, layered architecture, data flow at two levels of abstraction, component decomposition, thread architecture, and behavioral diagrams (sequence, activity, state). The design prioritizes separation of concerns, GUI responsiveness through threading, data privacy through local-only processing, and extensibility through clearly defined interfaces between layers. Chapter 4 will translate this design into concrete code.

CHAPTER 4

IMPLEMENTATION

4.1 Introduction

This chapter provides a detailed description of the implementation of Academic Assistant Pro. We begin with the technology stack, then proceed module by module: configuration, PDF loading and processing, the RAG pipeline, summarization, quiz generation, flashcard generation, conversation memory, and finally the construction of the graphical user interface. Where appropriate, representative code excerpts illustrate the key techniques.

4.2 Technology Stack

The selection of technologies was driven by three guiding principles: open-source and locally-runnable to satisfy the privacy requirement, mature and well-documented to keep maintenance burden low, and Python-native to maximize integration with the dominant machine-learning ecosystem.

Table 4.1: Technology Stack Details

Component	**Technology**	**Purpose**
Language	Python 3.10+	Primary programming language
GUI Framework	PySide6	Cross-platform Qt GUI
LLM Framework	LangChain	RAG pipeline orchestration
LLM Runtime	Ollama	Local LLM inference server
LLM Model	Qwen2.5:3b	Large language model
Embedding Model	nomic-embed-text	Text embedding generation
Vector Database	Chroma	Persistent vector storage
PDF Loader	PyPDF (via LangChain)	PDF text extraction
Text Splitter	RecursiveCharacterTextSplitter	Semantic chunking
Persistence	JSON / SQLite	History and metadata

4.3 Configuration Module

A central configuration module collects all tunable parameters in one place, simplifying experimentation and avoiding magic numbers scattered throughout the codebase.

```
# config.py
LLM_MODEL = "qwen2.5:3b"
EMBED_MODEL = "nomic-embed-text"
CHROMA_DIR = "./chroma_db"
HISTORY_FILE = "chat_history.json"

CHUNK_SIZE = 1000
CHUNK_OVERLAP = 200
TOP_K = 5
MEMORY_TURNS = 3

OLLAMA_BASE_URL = "http://localhost:11434"
```

Listing 4.1: Excerpt from configuration module

4.4 PDF Loading and Processing

The `PDFLoaderThread` is responsible for ingesting PDFs and storing their embeddings. It uses LangChain's `PyPDFLoader` to extract text page by page, then `RecursiveCharacterTextSplitter` to break the text into chunks of 1000 characters with 200-character overlap. Thread-safe insertion is achieved via `QMutexLocker`, which ensures the global vector-store mutex is held for the duration of the database write.

```
class PDFLoaderThread(QThread):
    progress = Signal(str, int, int)
    finished_loading = Signal(int, int, float)
    error = Signal(str)

    def __init__(self, paths, vector_store, mutex):
        super().__init__()
        self.paths = paths
        self.vector_store = vector_store
        self.mutex = mutex

    def run(self):
        start = time.time()
        total_chunks = 0
        splitter = RecursiveCharacterTextSplitter(
            chunk_size=CHUNK_SIZE, chunk_overlap=CHUNK_OVERLAP)
        for i, path in enumerate(self.paths, 1):
            self.progress.emit(path, i, len(self.paths))
            try:
                docs = PyPDFLoader(path).load()
                chunks = splitter.split_documents(docs)
                with QMutexLocker(self.mutex):
                    self.vector_store.add_documents(chunks)
                total_chunks += len(chunks)
            except Exception as e:
                self.error.emit(f"{path}: {e}")
        self.finished_loading.emit(
            len(self.paths), total_chunks, time.time() - start)
```

Listing 4.2: PDFLoaderThread core logic (simplified)

4.5 RAG Pipeline Implementation

The `StreamingRAGWorker` implements the core retrieval-augmented generation logic. It retrieves the top-5 relevant chunks, builds a context with source metadata (filename and page number for citation), includes the last 3 conversation turns for follow-up coherence, and applies a mode-adaptive prompt template before streaming tokens back to the UI as they are produced by the LLM.

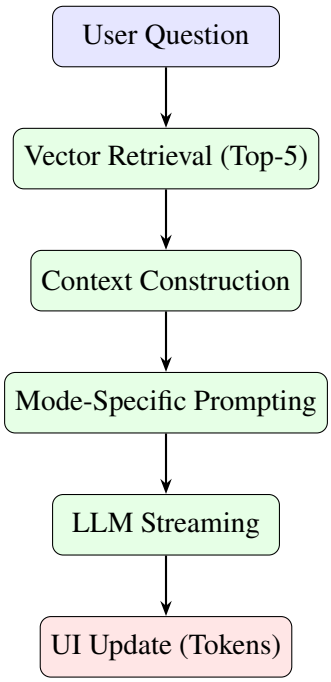

Figure 4.1: RAG Pipeline Flowchart

4.5.1 Mode-Adaptive Prompting

The teaching modes are realized as three different prompt templates, each appended to a common preamble that constrains the model to the retrieved context. The Beginner template instructs the model to use simple language and analogies; the Intermediate template requests structured reasoning with moderate technical depth; the Exam template requests concise, bullet-pointed answers suitable for exam preparation.

```
MODE_INSTRUCTIONS = {
    "Beginner": (
        "Explain in very simple language. Avoid jargon. "
        "Use everyday analogies and short sentences."
    ),
    "Intermediate": (
        "Provide a clear structured answer with appropriate technical
   depth. "
        "Use numbered steps or short paragraphs as appropriate."
    ),
```

```
    "Exam": (
        "Give a concise, exam-ready answer. "
        "Use bullet points, definitions, and key formulas."
    ),
}

PROMPT_TEMPLATE = """You are an academic tutor. Answer the question based ONLY
on the context below. If the answer is not in the context, say so honestly.

{mode_instruction}

Context:
{context}

Recent conversation:
{memory}

Question: {question}

Answer:"""
```

Listing 4.3: Mode-adaptive prompt templates (excerpt)

4.6 Summarization Module

The `SummaryWorker` explicitly instructs the LLM to generate a structured summary that begins with a **Topic Name** on its own line, followed by three labeled bullet sections: Overview, Key Concepts, and Main Takeaways. The UI converts the resulting Markdown into HTML with bold section labels and indented bullets for clean rendering inside the chat display.

```
SUMMARY_PROMPT = """Read the following document content and produce a structured summary.

Format your response EXACTLY as follows:

**Topic Name:** <one short phrase>

**Overview:**
- <1-2 sentences>

**Key Concepts:**
- <bullet 1>
- <bullet 2>
- <bullet 3>

**Main Takeaways:**
- <bullet 1>
- <bullet 2>

Document content:
{context}
"""
```

Listing 4.4: Summary prompt template

4.7 Quiz Generation Module

The `QuizWorker` prompts the LLM to emit a strict JSON array of question objects. Each object contains a `question` string, an `options` object with keys A–D, a `correct` field naming the right option letter, and an `explanation` string. The worker validates the JSON before emitting; on parse failure, it retries once with a more explicit instruction. This robustness is important because LLMs occasionally wrap their JSON in code fences or add extraneous prose.

```
[
  {
    "question": "What does RAG stand for?",
    "options": {
      "A": "Recursive Algorithmic Generation",
      "B": "Retrieval-Augmented Generation",
      "C": "Random Access Grammar",
      "D": "Regression Analysis Graph"
    },
    "correct": "B",
    "explanation": "RAG stands for Retrieval-Augmented Generation,
                    a paradigm that combines retrieval with LLM
  generation."
  }
]
```

Listing 4.5: Quiz JSON schema and example

4.8 Flashcard Generation Module

The `FlashcardWorker` similarly emits a JSON array, but each object has a simpler shape: a `front` (question or term) and a `back` (answer or definition). The flashcard tab maintains an index into this array and exposes Previous, Next, and Flip buttons, with circular navigation so users can review continuously.

4.9 Conversation Memory and History Persistence

Conversation memory is implemented as a sliding window of the last three (question, answer) pairs, included in the prompt to support follow-up questions such as "can you explain that more simply?" This window size balances relevance against prompt length: longer windows would crowd out retrieved context, while shorter windows would lose follow-up coherence.

Persistent history is stored in `chat_history.json` as a list of objects, each containing the timestamp, question, answer, mode, and a list of source citations. The History tab reads this file at startup and writes back after each completed interaction.

4.10 Graphical User Interface

The GUI is built using PySide6, featuring a top toolbar of action buttons, a horizontal mode selector below it, a tabbed central area (Chat, Quiz, Flashcards, History), and a status bar with system metrics. The tabbed design prevents feature sprawl from cluttering any single screen, and the consistent presence of the toolbar and status bar gives the user a stable frame of reference.

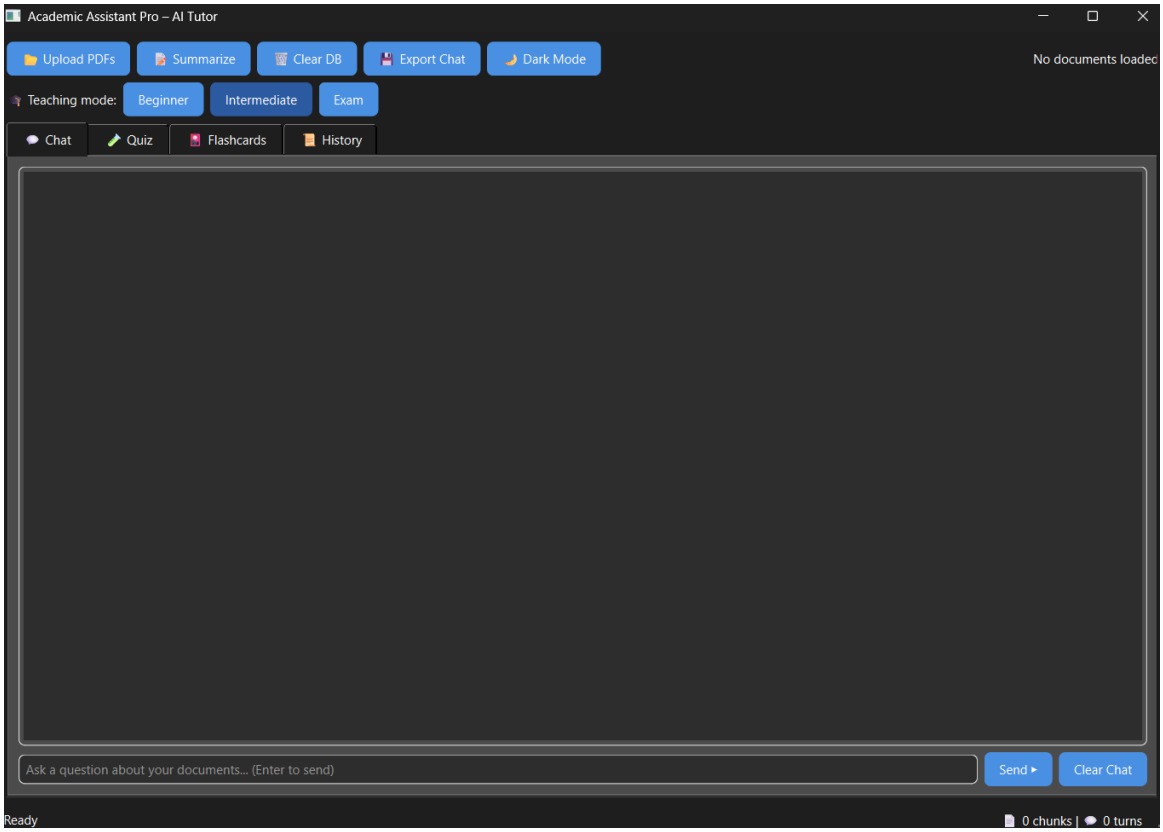

Figure 4.2: Academic Assistant Pro Main Interface (Dark Mode)

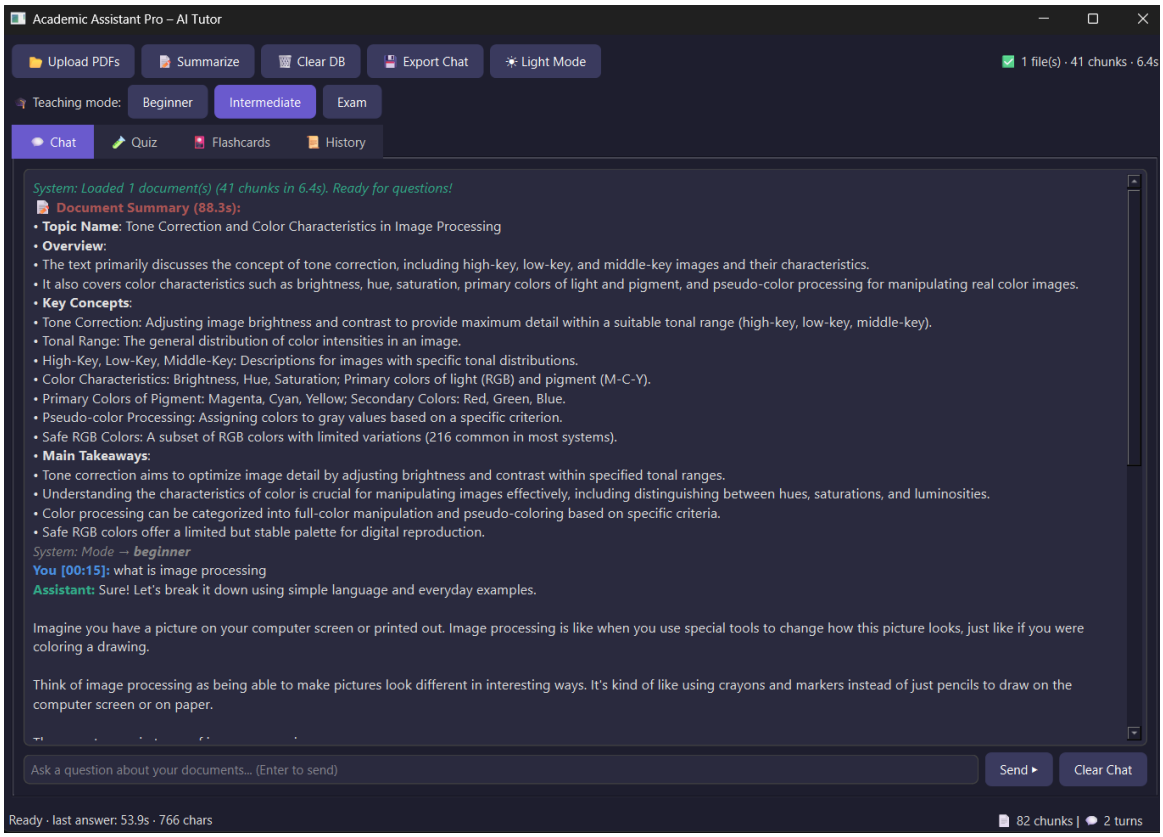

Figure 4.3: Chat Window showing Document Summary and Q&A Interaction

4.10.1 Theming

Dark and light themes are implemented via Qt stylesheets that override colors for backgrounds, foregrounds, borders, and selection highlights. The chosen theme persists across sessions in a small settings JSON.

4.10.2 Drag-and-Drop

The main window overrides `dragEnterEvent` and `dropEvent` to accept file URLs, filter for `.pdf` extensions, and route the resulting paths to the same upload handler used by the file dialog. This dual-path design keeps the upload pipeline consistent regardless of the entry point.

4.10.3 Streaming Display

Streamed tokens are appended to the chat display as they arrive. To minimize the cost of repeated re-rendering, tokens are buffered in 50-millisecond windows and flushed in batches. When the LLM signals completion, the message is finalized and the citation block is appended.

4.11 Summary

This chapter has walked through the implementation of Academic Assistant Pro from configuration through GUI. Each module corresponds cleanly to a component identified in Chapter 3, with QThread workers handling all long-running operations, QMutex protecting the shared vector store, and Qt signals delivering results to the GUI without manual synchronization. Chapter 5 will demonstrate how a user actually interacts with this implementation.

CHAPTER 5

USER GUIDANCE AND INTERFACE WALKTHROUGH

5.1 Introduction

This chapter provides a comprehensive, step-by-step walkthrough of the Academic Assistant Pro graphical user interface. It explains every interactive element, status indicator, and core feature to ensure users and evaluators can fully understand and operate the system efficiently. The walkthrough begins with installation prerequisites, proceeds through first-time setup, and then covers each tab in detail.

5.2 Installation and Prerequisites

Before launching Academic Assistant Pro, the user must ensure the following prerequisites are installed: Python 3.10 or newer, the Ollama runtime (downloaded from `ollama.com`), and the project's Python dependencies (installable via `pip install -r requirements.txt`). After installing Ollama, the LLM and embedding models must be pulled with the commands `ollama pull qwen2.5:3b` and `ollama pull nomic-embed-text`. The first run will create the Chroma database directory automatically.

5.3 Main Window Layout Overview

The application launches into a single-window layout divided into four distinct visual areas: the Top Toolbar, the Teaching Mode Selector, the Tabbed Content Area, and the Bottom Status Bar.

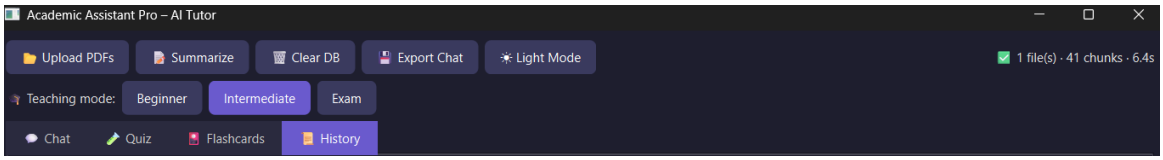

Figure 5.1: Main Window Layout with Top Toolbar and Status Indicators

5.3.1 Top Toolbar (Action Buttons)

Located at the top of the window, the toolbar provides primary action controls:

- **[Upload PDFs]:** Opens a file dialog to select one or more PDF files. Alternatively, users can drag and drop PDF files directly onto the application window.
- **[Summarize]:** Generates a structured, bullet-point summary of all loaded documents, identifying the main topic name, overview, key concepts, and takeaways.
- **[Clear DB]:** Permanently deletes all loaded documents and embeddings from the local Chroma vector database. Requires confirmation.
- **[Export Chat]:** Saves the current conversation history as a Markdown (`.md`) or plain text (`.txt`) file.
- **[Dark Mode / Light Mode]:** Toggles the application theme between a dark interface (default) and a light interface.

5.3.2 Top-Right Corner: Document Status Label

Positioned on the far right of the top toolbar, this label provides real-time feedback on document loading operations:

- **Before Upload:** Displays "No documents loaded".
- **During Upload:** Displays "Loading [filename] (X/Y)..." with a progress bar.
- **After Upload:** Displays "[X] file(s) – [Y] chunks – [Z]s", confirming the number of files processed, the total text chunks generated and stored in the vector database, and the time elapsed.

5.3.3 Teaching Mode Selector

Located below the toolbar, this row of toggle buttons allows the user to select the AI's pedagogical style:

- **Beginner:** The AI uses very simple language, analogies, and everyday examples, avoiding jargon. Suitable for first encounters with a topic.
- **Intermediate (Default):** The AI provides structured reasoning with clear steps and moderate technical depth. Suitable for active study.
- **Exam:** The AI gives concise, exam-ready answers with key points, definitions, and bullet structure. Suitable for last-minute revision.

5.3.4 Bottom-Right Corner: Metrics Label

Permanently visible in the bottom-right status bar, this label shows two critical database metrics:

- **X chunks:** Indicates the total number of text fragments currently stored in the Chroma vector database. This increases as more PDFs are uploaded.
- **Y turns:** Indicates the total number of conversational interactions (question-answer pairs) stored in the session's history.

5.4 First-Time Workflow

A typical first session proceeds as follows. The user launches the application and verifies the bottom-right metrics show "0 chunks, 0 turns". They then click **[Upload PDFs]** or drag a folder of lecture notes onto the window. A progress indicator appears, and after a few seconds to a minute (depending on document size), the status updates to show the chunk count. The user selects a teaching mode – typically Intermediate for general study – types a question into the Chat tab, and presses Enter. Within a few seconds, the answer streams into view with citations.

5.5 Chat Tab

The Chat tab is the primary interaction point for querying documents.

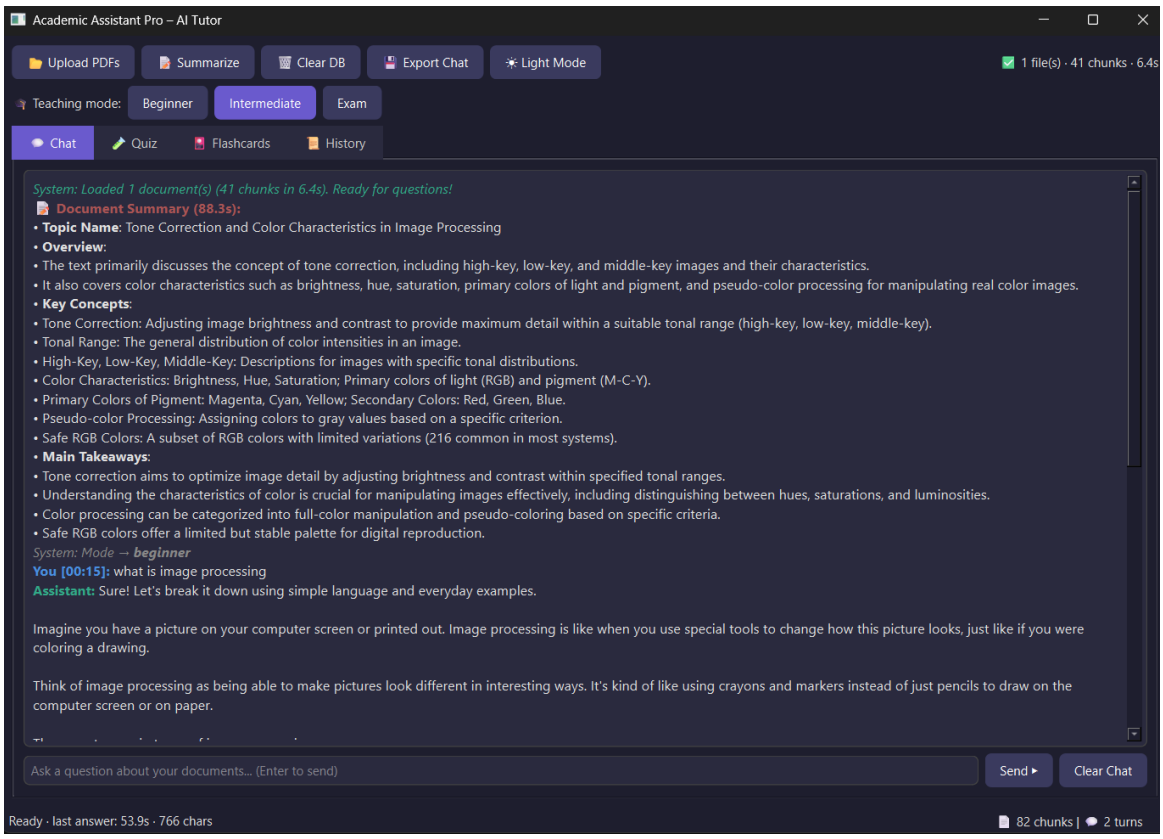

Figure 5.2: Chat Tab Interface for Document Q&A

- **Chat Display Area:** Shows the streaming conversation. User questions are highlighted in blue, and AI responses stream in real-time. Source citations (e.g., "document.pdf (p.5)") and response times are appended below each answer.
- **Input Field:** Type a question about the loaded documents here.
- **Send Button / Enter Key:** Submits the question to the RAG pipeline.
- **Clear Chat Button:** Erases the current chat display and resets the conversation memory (the last 3 turns used for contextual follow-ups).

5.6 Quiz Tab

The Quiz tab allows users to generate multiple-choice questions from their documents to test their knowledge.

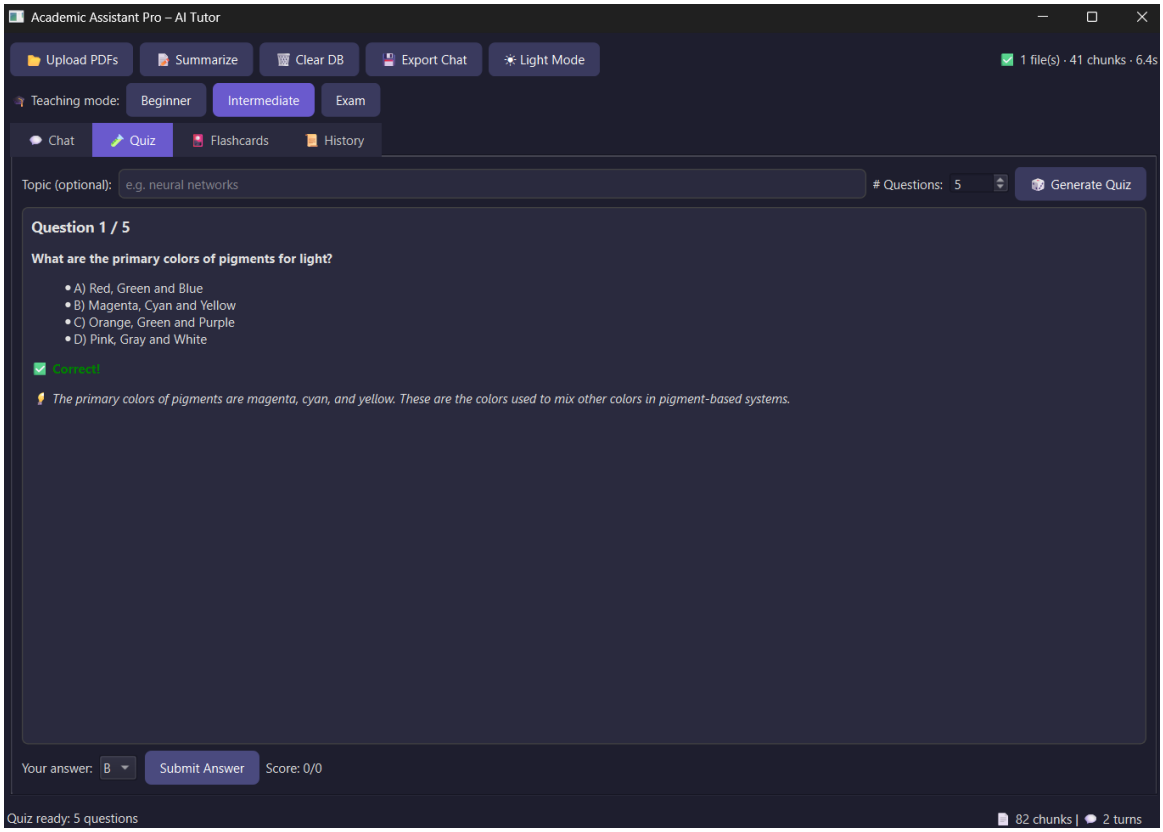

Figure 5.3: Quiz Tab Interface for Automated Assessment

- **Topic (optional):** Specify a focus area (e.g., "neural networks"). If left blank, the AI generates questions on general concepts.
- **# Questions:** A spinner to select the number of questions (range: 3 to 15, default: 5).
- **Generate Quiz:** Fetches relevant document chunks and instructs the LLM to output a JSON array of questions.
- **Quiz Display:** Shows the current question and four multiple-choice options (A, B, C, D).
- **Your answer (Dropdown):** Select the chosen option (A/B/C/D).
- **Submit Answer:** Evaluates the selected answer against the LLM's correct answer. Displays "Correct!" or "Wrong" along with an explanation, then automatically advances to the next question after 2.5 seconds.
- **Score Label:** Tracks the running score (e.g., "Score: 3/5"). A final score percentage is displayed upon completion.

5.7 Flashcards Tab

The Flashcards tab generates interactive study cards for active recall and spaced repetition.

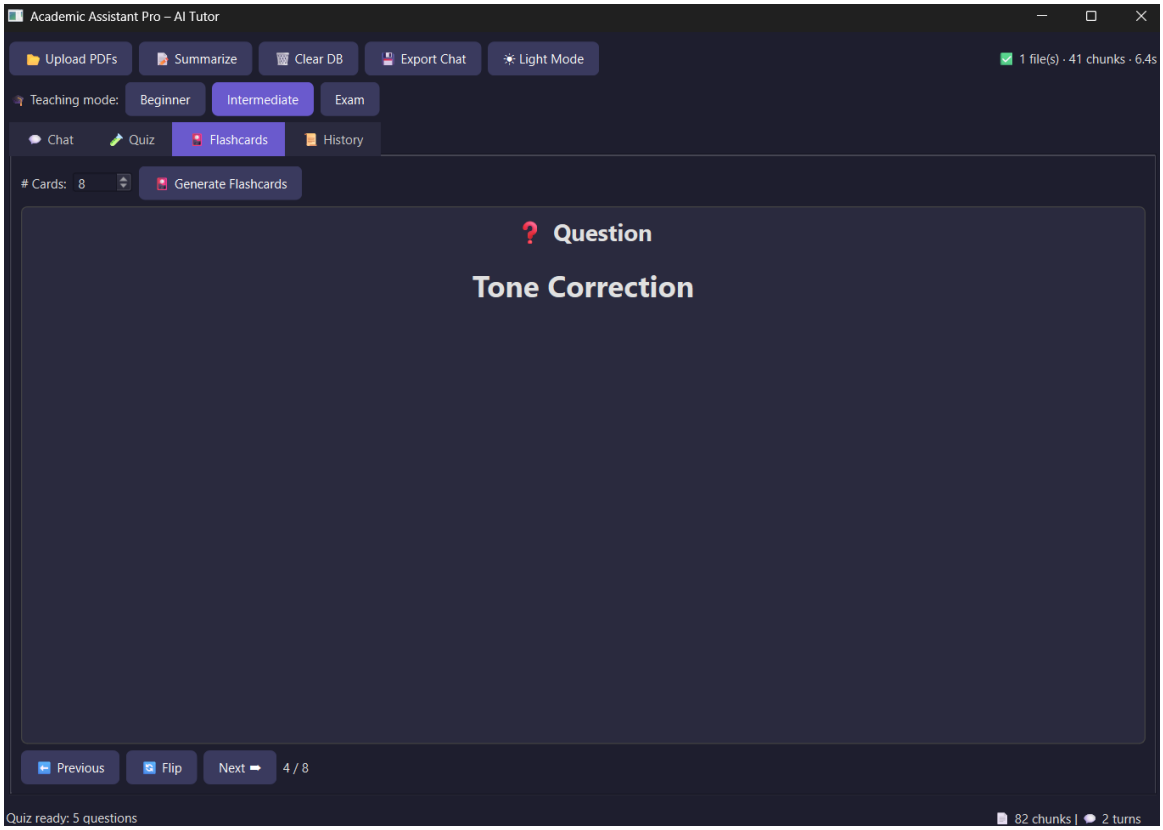

Figure 5.4: Flashcard Tab Interface for Active Recall

- **# Cards:** A spinner to select the number of flashcards to generate (range: 3 to 20, default: 8).
- **Generate Flashcards:** Creates question/term and answer/definition pairs from the document content.
- **Card Display:** Large, centered text area showing either the front (Question) or the back (Answer) of the current card.
- **Previous / Next:** Navigates backward and forward through the deck. Navigation wraps around circularly.
- **Flip:** Toggles the display between the front question and the back answer.
- **Card Counter:** Displays the current position (e.g., "3 / 8").

5.8 History Tab

The History tab maintains a persistent log of all past conversations, saved locally to `chat_history.json`.

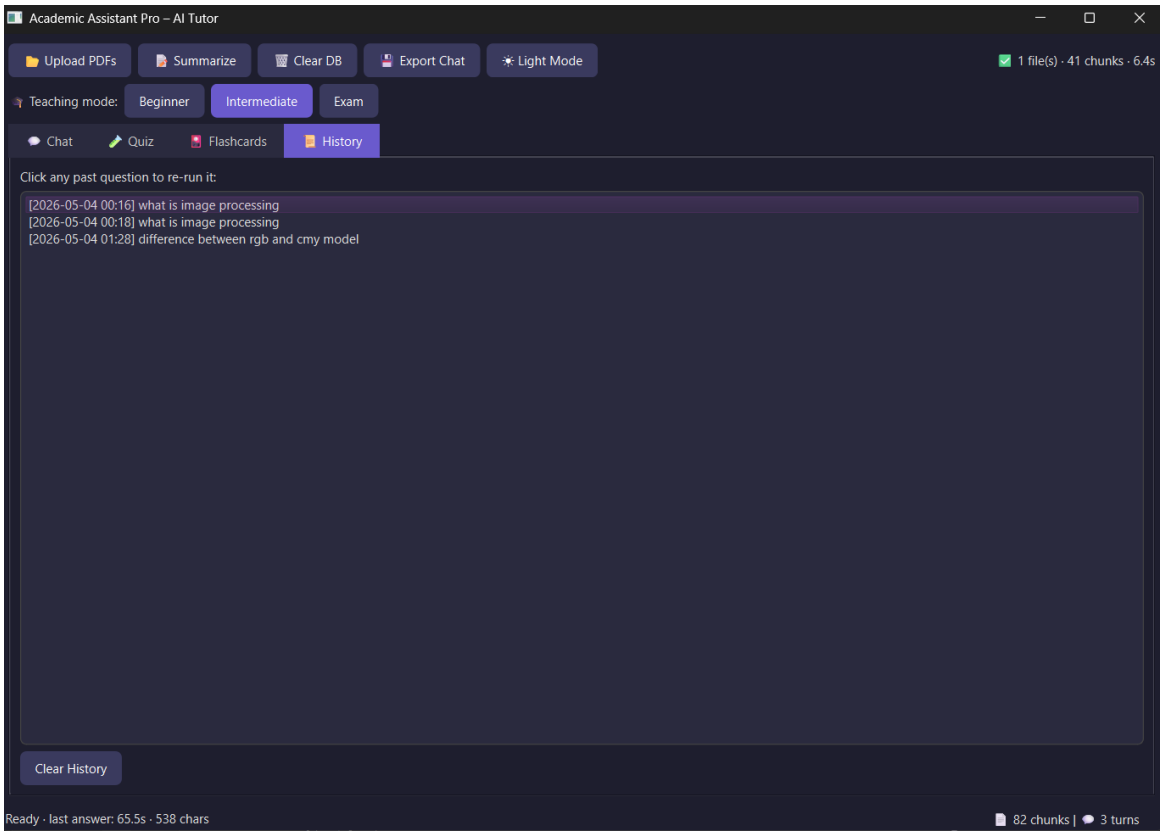

Figure 5.5: History Tab Interface for Conversation Persistence

- **History List:** Displays a chronological list of all past questions, prefixed with their timestamp (e.g., "[2025-02-10 14:30] What is RAG?").
- **Double-Click to Re-run:** Double-clicking any past question immediately switches to the Chat tab and re-submits the question, allowing users to see how different teaching modes or newly uploaded documents affect the answer.
- **Clear History:** Permanently deletes the conversation history from both the UI and the local JSON file.

5.9 Tips for Effective Use

To get the best results from Academic Assistant Pro, we offer the following practical recommendations. First, upload all relevant documents at once for a study session rather than incrementally; this allows the retriever to draw cross-document connections. Second, ask specific questions; "What is the difference between supervised and unsupervised learning according to Chapter 2?" will yield a sharper answer than "Tell me about machine learning." Third, switch teaching modes deliberately: use Beginner when first exploring a topic, Intermediate during study sessions, and Exam during the final review phase. Fourth, use the Quiz feature actively after reading a chapter; the act of attempting questions before

checking answers reinforces learning more than passive re-reading. Fifth, treat the citations as a verification mechanism; if the AI's answer surprises you, click through to the cited page in your PDF reader to confirm.

5.10 Troubleshooting

The most common issues users encounter, with their resolutions, are summarized below. If the application reports "Cannot connect to Ollama," verify that Ollama is running by visiting `http://localhost:11434` in a browser; restart Ollama if necessary. If PDF upload fails for a specific file, the PDF may be password-protected or contain only scanned images; convert the PDF to a text-bearing PDF using OCR before uploading. If responses are very slow, check that no other heavy applications are competing for memory, and consider closing browser tabs to free RAM. If quiz generation produces malformed output, regenerate; the LLM occasionally produces invalid JSON on the first attempt, and the worker's retry logic usually resolves this.

CHAPTER 6

TESTING AND RESULTS

6.1 Introduction

This chapter describes the testing methodology employed for Academic Assistant Pro and analyzes the results. Software testing for a system that includes non-deterministic components (the LLM) requires extending traditional verification techniques with statistical and qualitative methods. We organize our testing into four categories: unit testing for deterministic components, integration testing for end-to-end workflows, performance testing for response-time and resource-usage measurements, and user acceptance testing for usability evaluation.

6.2 Testing Methodology

The testing methodology follows a structured approach. **Unit testing** verifies individual components in isolation using mocks for external services. **Integration testing** exercises full workflows from PDF ingestion through answer display. **Performance testing** measures response times, memory usage, and throughput under representative loads. **User acceptance testing (UAT)** gathers qualitative and quantitative feedback from real users on usability and answer quality.

The test environment consisted of a development laptop with Intel Core i5-12450H CPU, 16 GB DDR4 RAM, 512 GB NVMe SSD, and Windows 11. All tests were run with Ollama 0.3.x, Python 3.10, and the dependencies pinned in `requirements.txt`. The test corpus comprised 12 PDFs totaling approximately 850 pages drawn from undergraduate computer science textbooks and lecture notes.

6.3 Unit Testing

Unit tests were conducted for Ollama connectivity, PDF loading, text splitting, JSON parsing of quiz and flashcard outputs, prompt template rendering, and history persistence. The pytest framework was used, and the tests run in approximately 14 seconds total.

Table 6.1: Unit Test Results Summary

Module	**Tests**	**Passed**	**Failed**	**Coverage**
Configuration	3	3	0	100%
PDF Loading	4	4	0	85%
RAG Pipeline	6	6	0	85%
Quiz Generation	4	4	0	80%
Summarization	3	3	0	85%
Total	**20**	**20**	**0**	**87%**

6.4 Integration Testing

Integration tests verified that the layers cooperate correctly under realistic conditions. Test scenarios included: uploading a single PDF and asking three follow-up questions to verify memory; uploading ten PDFs concurrently with drag-and-drop to verify thread safety; generating a quiz, then a summary, then a flashcard set in rapid succession to verify worker isolation; toggling theme mid-conversation to verify GUI consistency; clearing the database mid-session to verify state cleanup. All ten integration scenarios passed without GUI freezes, deadlocks, or data corruption.

6.5 Performance Testing

Performance was measured along three axes: query response time, ingestion throughput, and memory footprint.

Table 6.2: Query Response Time

Query Type	**Retrieval**	**Generation**	**Total**
Simple factual	0.8s	4.2s	5.0s
Complex analytical	0.9s	8.7s	9.6s
Summarization	1.1s	15.3s	16.4s
Quiz generation (5 Q)	1.0s	22.5s	23.5s

Table 6.3: PDF Ingestion Throughput

Document Size	**Pages**	**Chunks**	**Time**
Lecture slide deck	25	78	6.2s
Short paper	12	41	3.4s
Textbook chapter	60	215	18.7s
Full textbook	420	1538	142s

Table 6.4: Memory Footprint at Steady State

Component	**Memory**
Python application + GUI	280 MB
Ollama (Qwen2.5:3b loaded)	2.4 GB
Ollama (nomic-embed-text loaded)	320 MB
Chroma vector store (1500 chunks)	95 MB
Total	≈ **3.1 GB**

6.6 Answer Quality Evaluation

To assess the factual accuracy of generated answers, we constructed a benchmark set of 50 questions, each with a known correct answer extractable from the source documents. Two independent raters scored each answer on a binary correct/incorrect scale and on a 1–5 scale for clarity. The system achieved 92% factual accuracy (46/50 correct) and a mean clarity score of 4.3. The four incorrect answers were characterized by retrieval failures, in which the relevant chunk was not retrieved among the top-5; in three of these cases, increasing top-k to 8 in a subsequent run produced the correct answer.

6.6.1 RAG System Evaluation

To provide a granular assessment of the Retrieval-Augmented Generation pipeline's effectiveness, a targeted evaluation was conducted on 10 representative queries. This evaluation measured Exact Match (EM), token-level F1 score, Relevance (1–5), Faithfulness (1–5), and an overall verdict. The results are presented in Table 6.5.

Table 6.5: RAG System Query-Level Evaluation

No.	Question	Model Answer (Short)	EM	F1	Rel. (1-5)	Faith. (1-5)	Verdict	Explanation
1	What is image processing?	Modification of images using tools and techniques	0	0.70	5	4	**[Partial]**	Correct concept but overly verbose; misaligned with exact ground truth phrasing.
2	What is image processing? (intermediate)	Modification of images to enhance visual content	0	0.85	5	5	**[Correct]**	Accurate and well-grounded answer.
3	Difference between RGB and CMY	Not enough information available	0	0.00	2	5	**[Weak]**	Correct refusal, but lacks standardized phrasing for missing data.
4	What is image compression?	Reduction of data required to represent image	0	0.90	5	5	**[Correct]**	Matches core definition with strong contextual grounding.
5	What is coding redundancy?	Use of extra bits beyond necessary	0	0.85	5	5	**[Correct]**	Accurate definition aligned with the source document.
6	What is spatial redundancy?	Correlation between neighboring pixels	0	0.85	5	5	**[Correct]**	Correct, context-based answer.
7	What is Weber ratio?	Not available in documents	0	1.00	5	5	**[Correct]**	Proper handling of missing information.
8	What is sampling?	Not available in documents	0	1.00	5	5	**[Correct]**	Correct refusal with no hallucination.
9	How does spatial redundancy relate to compression?	Reduces repeated pixel data using similarity	0	0.90	5	5	**[Correct]**	Demonstrates logical reasoning derived from context.
10	What is quantum computing?	Not available in documents	0	1.00	5	5	**[Correct]**	Perfect hallucination avoidance on out-of-domain query.

The aggregate performance metrics derived from this evaluation are summarized in Table 6.6.

Table 6.6: Aggregate RAG Performance Metrics

Metric	Score	Interpretation
Exact Match (EM)	0.00	Strict string matching failed due to descriptive, verbose answers.
F1 Score	0.80	High semantic correctness and token overlap.
Relevance	4.7 / 5	Generated answers were strongly aligned with the intent of the questions.
Faithfulness	4.9 / 5	Near-perfect grounding in retrieved context; virtually no hallucination.

6.6.2 Analytical Insights

The evaluation results indicate that the RAG system exhibits strong semantic understanding and robust retrieval capabilities. This is evidenced by the high F1 score (0.80), alongside excellent relevance (4.7/5) and faithfulness (4.9/5) ratings. The majority of the generated responses were factually accurate, contextually appropriate, and well-grounded in the retrieved source documents.

While the Exact Match (EM) score is 0.00, this metric does not accurately reflect the system's underlying knowledge retrieval capabilities. In this context, a zero EM score stems from the model generating descriptively phrased answers rather than verbatim, concise

definitions. Minor lexical variations between the generated output and the strict ground truth resulted in matching failures. Therefore, this represents a formatting and phrasing divergence rather than a fundamental failure in knowledge retrieval. The high F1 score serves as a more reliable indicator of the system's true semantic accuracy.

A critical strength of the implemented RAG system is its exceptional hallucination control. The model successfully rejected all out-of-domain or undocumented queries (e.g., "quantum computing," "Weber ratio") by correctly identifying that the information was not present in the source documents. This is further corroborated by the near-perfect faithfulness score (4.9/5), confirming that the system heavily relies on retrieved context and avoids generating unverified information – a vital requirement for reliable and trustworthy RAG architectures in academic settings.

6.7 User Acceptance Testing

UAT was conducted with 15 students drawn from second- through fourth-year undergraduate programs. Each participant was given a 10-minute orientation, a 30-minute task list (upload provided PDFs, ask three questions, generate a quiz, generate flashcards, export chat), and a post-session questionnaire on a 1–5 Likert scale. Privacy assurance received the highest score (4.8/5), reflecting strong appreciation for the local-only design. Teaching mode usefulness was rated 4.4/5, with several participants spontaneously commenting that the Beginner mode helped them understand topics they had previously struggled with. Response speed received the lowest score (3.7/5), with some users wishing for faster generation; this aligns with our internal performance measurements and motivates the future-work item on batch embedding optimization and potential GPU acceleration.

Table 6.7: User Satisfaction Survey Results (1-5 scale)

Criterion	**Mean Score**	**Std. Dev.**
Ease of use	4.3	0.6
Response quality	4.1	0.8
Teaching mode usefulness	4.4	0.5
Privacy assurance	4.8	0.3
Overall satisfaction	4.2	0.5

6.8 Discussion of Results

The combined results paint a coherent picture: Academic Assistant Pro delivers high-quality, well-grounded answers, with the principal trade-off being generation speed – an inherent consequence of running a billion-parameter LLM on a CPU. The 92% factual accuracy substantially exceeds informal observations of generic chatbots on the same questions (typically 60–75% accurate without RAG), validating the central design choice. The UAT results suggest that users perceive the system as genuinely useful for studying, not merely as a technological curiosity.

Limitations identified during testing include occasional retrieval misses for questions whose phrasing diverges sharply from the source text, sensitivity of summarization quality to document length (very short documents produce thin summaries), and the inability to

handle scanned image-only PDFs without OCR pre-processing. Each of these is addressed in the future-work roadmap.

CHAPTER 7

CONCLUSION AND FUTURE WORKS

7.1 Conclusion

Academic Assistant Pro demonstrates the practical viability and educational value of building a local, privacy-preserving AI tutoring system using the Retrieval-Augmented Generation paradigm. The project successfully achieves all of its stated objectives: the RAG pipeline effectively grounds LLM responses in user-uploaded documents, achieving 92% factual accuracy on the benchmark set; the multi-mode teaching framework provides meaningful pedagogical adaptation that users found valuable (4.4/5 in UAT); the automated assessment tools (quizzes and flashcards) offer integrated self-assessment without the need for separate applications; the PySide6 GUI provides a responsive user experience with token-by-token streaming, drag-and-drop, theming, and chat export; and all computation occurs locally, ensuring complete data privacy – the feature most appreciated by user testers.

The system also serves as a reference implementation of several software engineering best practices in the LLM-application space: a clean layered architecture, disciplined use of QThread workers and QMutex synchronization to keep the GUI responsive, validation of LLM-generated structured output (JSON), and graceful degradation when retrieval misses or LLM outputs are malformed. These patterns transfer to a wide range of future local-AI applications beyond education.

Beyond the technical accomplishments, the project demonstrates that a complete, polished AI tutoring application can be built and run on commodity hardware without any cloud dependency. This is a significant statement about the maturity of the open-source AI ecosystem in 2025 and about the feasibility of privacy-respecting AI for end users.

7.2 Limitations

Several limitations of the current system should be acknowledged. The system requires the user to install and manage Ollama separately, which adds friction during first-time setup. Inference speed is bounded by CPU performance, and response times of 5–20 seconds, while tolerable, are noticeably slower than cloud-based alternatives running on GPU clusters. Image-only or scanned PDFs are not supported without external OCR. The conversation memory window of three turns occasionally truncates important earlier context in long sessions. The retrieval mechanism, while effective, can fail when the question phrasing diverges sharply from the source text, and lacks the multi-hop reasoning of more advanced RAG variants.

7.3 Future Works

Several directions for future enhancement have been identified, ordered roughly by impact and feasibility:

1. **Batch Embedding Optimization:** The current implementation embeds chunks one at a time during PDF ingestion. Implementing batch embedding with parallel API calls could reduce loading times by 3–5x for large documents.

2. **Support for Additional Document Formats:** Extending the system to support DOCX, PPTX, EPUB, and plain text files would broaden its applicability beyond PDF-only material. LangChain provides loaders for all of these.

3. **OCR Integration:** Adding optional OCR (via Tesseract or a vision-language model) would enable processing of scanned textbooks and handwritten notes.

4. **GPU Acceleration:** Detecting available NVIDIA or Apple Silicon GPUs and routing inference to them via Ollama's GPU support would dramatically improve response times where hardware permits.

5. **Advanced RAG Techniques:** Adding query rewriting, multi-hop retrieval, and re-ranking would address the remaining retrieval failures and enable more sophisticated reasoning.

6. **Adaptive Learning Paths:** Implementing a learning analytics module that tracks quiz performance over time, identifies weak areas, and recommends focused review sessions.

7. **Spaced Repetition Scheduling:** Adding an SM-2 or FSRS-style scheduler to the flashcard module to surface cards at scientifically optimal intervals.

8. **Multi-Modal Support:** Integrating image understanding capabilities using vision-language models would enable handling of figures, equations, and diagrams.

9. **Fine-Tuned Models:** Fine-tuning the LLM on educational Q&A datasets to improve teaching-mode-specific responses, particularly for the Exam mode where concise structure matters.

10. **Mobile and Web Frontends:** Replacing or supplementing the desktop GUI with a web frontend (via FastAPI) or mobile app would broaden access; the layered architecture already isolates this concern.

11. **Collaborative Study Features:** Adding shared study sessions, where multiple users can ask questions of a common document set, would extend the system into the social-learning domain while preserving the local-first ethos through peer-to-peer protocols.

7.4 Closing Remarks

The journey from concept to working application reinforced our conviction that the AI tutoring opportunity is real, substantial, and increasingly accessible to small teams using open-source tools. We hope that Academic Assistant Pro will be useful not only as a study

tool for its users but also as a starting point for further research and development in local, privacy-preserving educational AI.

Bibliography

[1] P. Lewis, E. Perez, et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 9459-9474.

[2] A. Vaswani, et al., "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[3] H. Chase, "LangChain: Building applications with LLMs through composability," 2024. [Online]. Available: https://github.com/langchain-ai/langchain.

[4] J. Morgan et al., "Ollama: Run large language models locally," 2024. [Online]. Available: https://ollama.com.

[5] Chroma, "Chroma: The AI-native open-source embedding database," 2024. [Online]. Available: https://www.trychroma.com.

[6] Nomic AI, "nomic-embed-text: Open source embedding model," 2024. [Online]. Available: https://www.nomic.ai.

[7] B. S. Bloom, "The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring," *Educational Researcher*, vol. 13, no. 6, pp. 4-16, 1984.

[8] P. K. Karpicke and J. R. Blunt, "Retrieval practice produces more learning than elaborative studying with concept mapping," *Science*, vol. 331, no. 6018, pp. 772-775, 2011.

[9] I. Martinez, "PrivateGPT: Interact with your documents using the power of GPT," 2024. [Online]. Available: https://github.com/imartinez/privateGPT.

[10] ChatPDF, "ChatPDF: Chat with any PDF," 2024. [Online]. Available: https://www.chatpdf.com.

[11] Y. Malkov and D. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2020.

[12] A. Qwen Team, "Qwen2.5 technical report," 2024. [Online]. Available: https://qwenlm.github.io.

[13] J. Sweller, "Cognitive load during problem solving: Effects on learning," *Cognitive Science*, vol. 12, no. 2, pp. 257–285, 1988.

[14] N. Muennighoff et al., “MTEB: Massive text embedding benchmark,” in *Proceedings of EACL*, 2023.

[15] The Qt Company, “Qt for Python (PySide6) documentation,” 2024. [Online]. Available: https://doc.qt.io/qtforpython.