Democratic and Popular Republic of Algeria
Ministry of Higher Education and Scientific Research
University Centre of Illizi
Institute of Science

Department of Computer Science

Artificial Intelligence & its Applications Master's Thesis

AI-Powered Teaching Assistant Using Retrieval-Augmented Generation (RAG)

Authors:

- Moulai Dhiyaeddine
  moulai.dhiyaeddine@cuillizi.dz

Advisors:

- Mr. Messikh Chouaib
  messikh.chouaib@cuillizi.dz

Examining Committee:

- (*President*) Dr. Bourourou Fares
  f.bourourou@univ-boumerdes.dz
- (*Examiner*) Dr. Benmachiche Abderrahim
  benmechiche.rahim@cuillizi.dz
- (*Incubator Panelist*) Dr. Boudiaf Abdellatif
  boudiaf.abdellatif@cuillizi.dz
- (*EDC Panelist*) Dr. Djorfi Zakaria
  djorfi.zakaria@cuillizi.dz

Academic Year: 2026

Abstract

This thesis presents the design and implementation of **TeachBot**, an AI-powered teaching assistant built upon the Retrieval-Augmented Generation (RAG) architecture. The system enables university students to interact intelligently with their course materials lecture notes, textbooks, and PDF documents through three core functionalities: context-aware question answering with source citations, automatic document summarization, and interactive quiz generation (multiple-choice and short-answer).

The proposed architecture combines a local large language model (TinyLlama 1.1B, served via Ollama) with a ChromaDB vector database indexed using the all-MiniLM-L6-v2 sentence embedding model. A FastAPI backend manages the retrieval pipeline and exposes seven REST endpoints, while a React/Vite frontend provides an intuitive web interface. The entire system is containerized using Docker Compose, enabling reproducible, dependency-free deployment on commodity hardware without reliance on cloud services.

Experimental evaluation conducted on three computer science course PDFs totaling 103 pages and 103 indexed chunks demonstrates that TeachBot achieves 100% response faithfulness to course content, 100% source citation rate (document name and page number), and an average question-answering latency of 25–40 seconds on Intel CPU hardware. The system operates fully offline, addressing critical concerns of data privacy and infrastructure cost in educational settings, particularly relevant for universities in southern Algeria.

تقدم هذه الأطروحة تصميم وتنفيذ **TeachBot**، وهو مساعد تعليمي مدعوم بالذكاء الاصطناعي ومبني على بنية التوليد المعزز بالاسترجاع (RAG). يتيح النظام لطلبة الجامعة التفاعل الذكي مع مقرراتهم الدراسية — مثل المحاضرات، الكتب المدرسية، وملفات PDF — من خلال ثلاث وظائف أساسية: الإجابة على الأسئلة المستندة إلى السياق مع توثيق المصادر، التلخيص التلقائي للمستندات، والتوليد التفاعلي للاختبارات (أسئلة متعددة الاختيارات وأسئلة ذات إجابات قصيرة).

تجمع البنية المقترحة بين نموذج لغوي كبير محلي Phi-3-Mini يتم تشغيله عبر Ollama وقاعدة بيانات متجهة ChromaDB مفهرسة باستخدام نموذج تضمين الجمل all-MiniLM-L6-v2. تدير واجهة FastAPI الخلفية خط أنابيب الاسترجاع وتوفر سبع نقاط نهاية برمجية REST endpoints بينما توفر واجهة React/Vite الأمامية واجهة ويب سهلة الاستخدام. تم وضع النظام بأكمله داخل حاويات باستخدام Docker Compose، مما يسمح بنشر قابل للتكرار وخالٍ من التبعيات على الأجهزة العادية دون الاعتماد على الخدمات السحابية.

أظهر التقييم التجريبي الذي أُجري على ثلاثة ملفات PDF لمقررات في علوم الكمبيوتر (بإجمالي 103 صفحة و103 قطعة مفهرسة) أن TeachBot يحقق موثوقية بنسبة 100% في الاستجابة للمحتوى الدراسي، ونسبة 100% في معدل توثيق المصادر (اسم المستند ورقم الصفحة)، ومتوسط زمن استجابة للإجابة يتراوح بين 25 و40 ثانية على معالجات CPU Intel. يعمل النظام دون اتصال بالإنترنت تمامًا، مما يعالج المخاوف الحرجة المتعلقة بخصوصية البيانات وتكلفة البنية التحتية في البيئات التعليمية، وهو أمر ذو أهمية بالغة للجامعات في الجنوب الجزائري.

Key words: Retrieval-Augmented Generation, RAG, Large Language Models, Teaching Assistant, ChromaDB, Vector Database, Natural Language Processing, Educational AI, Ollama.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Mr. Chouaib, for their continuous guidance, constructive feedback, and unwavering support throughout this project. Their expertise in artificial intelligence and pedagogy was essential in shaping both the technical direction and the academic quality of this work.

I also wish to thank the faculty members of the Department of Computer Science at the University Centre of Illizi for their dedication to teaching in this region and for providing the academic foundation upon which this thesis is built.

Special thanks go to my mother,my father,Dr.Achouak,our futur Architecture Malak,my futur doctor chahd .

Finally, I extend my sincere gratitude to my grand family,my grands parents Moulai Tahar,Tati Fatma may Allah bless them,My dear amma Zhour,baba Moussa, My Aunt Radia and friends sidou,haythem,wakil,abdellah for their unwavering moral support and patience during the long hours of development and writing. This work would not have been possible without them.

This work is dedicated to every student in Algeria who deserves access to quality educational tools, regardless of geographic or economic constraints.

Contents

List of Figures

List of Tables

Chapter 1

General Introduction

1.1 Context

In the era of the Fourth Industrial Revolution, artificial intelligence has changed from a research curiosity into a practical tool that transforms almost every part of human activity. One of the most significant recent developments is Large Language Models (LLMs). These are deep neural networks trained on vast amounts of text that show remarkable abilities in understanding and generating natural language . Systems like GPT-4, LLaMA, and Mistral have shown that machines can now hold coherent conversations, summarize complex documents, answer detailed questions, and produce educational content at the highest levels.

In higher education, this technology creates new opportunities to tackle longstanding teaching challenges. Today's students deal with a growing amount of varied course materials, including lecture notes, textbooks, research papers, and slides. They often have to process these materials on their own, typically without enough support outside formal class hours. This issue is especially noticeable in remote universities, like those in southern Algeria, where access to tutors, library resources, and digital infrastructure can be limited.

However, using LLMs in educational settings comes with major challenges. General-purpose language models, trained on large internet datasets, often lack specific knowledge about a particular course content. They can give plausible-sounding but factually incorrect answers, a problem known as hallucination. Additionally, sending sensitive academic documents to cloud-based AI services raises serious concerns about data privacy. Relying on paid APIs also creates financial barriers that may struggle to overcome.

One might think that training a language model on course-specific materials would solve these problems. If you teach the model directly using the content it needs to know, it should be able to answer questions about it correctly, right? In practice, it is not that simple. Fine-tuning changes the model's internal weights to focus on certain types of knowledge, but it does not ensure that the model will stop making mistakes. Changes only what the model is more likely to say, not whether its responses are true. A fine-tuned model can still give confident, fluent, and completely wrong answers, especially when a question includes details that were not well represented in the training data or are phrased differently from the examples encountered during fine-tuning. In addition to the issue of reliability, fine-tuning is costly in terms of both time and resources. It requires gathering and organizing a high-quality dataset of question-answer pairs from the course material, running several training cycles to find the right settings, and then repeating this entire process every time the course content changes. A new semester, a new version of the lecture notes, or a new textbook chapter each necessitates a new fine-tuning run. In a university setting, especially in institutions with limited GPU infrastructure, such as those in southern Algeria, this ongoing computational burden is simply not feasible. Fine-tuning is not a one-time effort; it becomes a maintenance challenge that increases with the variety and frequency of content updates.

Retrieval-Augmented Generation (RAG) is a method that addresses these limitations [2]. By grounding the model's responses in a context that is built dynamically from a verified document collection, RAG can provide accurate, reliable, and traceable answers without needing the model to remember all the information during training.

1.2 Problematic

The central problem addressed by this work can be stated as follows:

> *The main issue this work addresses is: How can we create an intelligent teaching assistant that gives university students accurate, source-based, and relevant answers from their course materials while working ?*

This issue breaks down into five specific research questions:

1. How can we effectively index diverse PDF course documents for quick semantic retrieval?
2. How can we make sure that the system's responses are directly based on course content and not made up?
3. How can we generate automatic summaries and quiz questions from the indexed materials?
4. How can we deploy the system as a portable, easy-to-use application that needs no setup expertise?
5. How does the system perform with real course materials regarding response quality and speed?

1.3 Objectives

The primary objective of this project is to design, develop, and evaluate **TeachBot**, an AI-powered teaching assistant based on the RAG architecture. Specific objectives include:

1. Design a complete RAG pipeline adapted to the educational domain.
2. Implement a PDF ingestion pipeline with semantic chunking and vector indexing.
3. Develop a context-aware Q&A module with systematic source attribution.
4. Build automatic document summarization and quiz generation modules.
5. Design and implement an intuitive web interface accessible via standard browser.
6. Ensure fully local, with no external API dependencies.
7. Containerize the full stack using Docker Compose for one-command deployment.
8. Evaluate system performance using quantitative and qualitative metrics.

1.4 Contributions

The main contributions of this work are:

- A complete, end-to-end RAG pipeline specifically designed for educational document Q&A.
- A fully open-source, locally deployable, containerized teaching assistant integrating Q&A, summarization, and quiz generation in a single application.
- A systematic source citation mechanism ensuring every answer references its source document and page number.

1.5 Document Organization

This document is structured as follows.

- Chapter 1 (State of the Art) reviews the foundational technologies.
- Chapter 2 (Analysis and Design) presents the system architectural design.
- Chapter 3 (Implementation) details the technical realization.
- Chapter 4 (Experimentation and Evaluaiton) presents the evaluation results and discussion.

Chapter 2

State of the Art

2.1 Introduction

This chapter reviews the foundational concepts and prior work that underpin systems of this nature. Building an AI-powered teaching assistant that is grounded in specific course documents requires a solid understanding of several interconnected fields: natural language processing, information retrieval, vector databases, and the application of artificial intelligence in education. The same theoretical foundations that support systems such as Khanmigo [1] and the RAG-based tutor studied by Slade et al. [2] apply directly to the design choices made in this work. This chapter surveys those foundations and the relevant prior studies, providing the technical and academic context necessary to situate the contributions of this thesis.

2.2 Large Language Models

2.2.1 Definition and Historical Context

A Large Language Model (LLM) is a neural network trained on vast quantities of text data with the objective of predicting the next token in a sequence [3]. This apparently simple objective gives rise to emergent capabilities including language understanding, reasoning, summarization, translation, and code generation.

The field was transformed by the introduction of the *Transformer* architecture by Vaswani et al. in 2017 [4]. The self-attention mechanism at the heart of the Transformer enabled models to capture long-range dependencies in text far more effectively than previous recurrent approaches. The core attention computation is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{2.1}$$

where Q, K, and V are the query, key, and value matrices, and d_k is the key dimensionality.

The subsequent scaling of Transformer models from BERT (110M parameters) to GPT-3 (175B parameters) and beyond revealed surprising emergent capabilities that appear qualitatively at certain scales of training [5].

2.2.2 Open-Source LLMs

The release of LLaMA by Meta in 2023 [6] democratized access to capable language models by providing open weights that researchers can fine-tune and deploy locally. Mistral-7B [7] and TinyLlama [8] further demonstrated that smaller, efficiently trained models can achieve competitive performance on many tasks while being deployable on consumer hardware.

2.2.3 Hallucination Problem

A critical limitation of LLMs for factual question answering is *hallucination* the confident generation of statements that are factually incorrect or unsupported. In an educational context, hallucination is particularly damaging: a student who receives an incorrect explanation may be actively misled. This limitation motivates the use of RAG.

2.3 Retrieval-Augmented Generation (RAG)

2.3.1 Principle

Retrieval-Augmented Generation was formally introduced by Lewis et al. (2020) [9]. The core idea is to augment the language model's input with relevant passages retrieved from an external knowledge base, grounding generation in verifiable information.

A RAG system operates in two phases:

1. **Retrieval phase**: Given a query q, a retriever searches a document collection and returns the top-k most relevant passages.

2. **Generation phase**: The LLM generates an answer conditioned on both the query and the retrieved context.

This approach offers several advantages: the knowledge base can be updated without retraining the model, it enables precise source attribution, and it substantially reduces hallucination [10].

2.3.2 Dense Retrieval and Vector Databases

Modern RAG systems use *dense retrieval*: both queries and documents are encoded into dense vector representations (embeddings), and retrieval finds nearest neighbors in the embedding space. Semantic similarity between a query vector $\mathbf{q}$ and document vector $\mathbf{d}$ is measured with cosine similarity:

$$\text{sim}(\mathbf{q}, \mathbf{d}) = \frac{\mathbf{q} \cdot \mathbf{d}}{\|\mathbf{q}\|\|\mathbf{d}\|} \tag{2.2}$$

Vector databases such as FAISS [11] and **ChromaDB** [12] are purpose-built to store and efficiently query dense vector embeddings.

2.3.3 Embedding Models

Embedding models are neural networks that convert textual information into dense numerical vectors, called *embeddings*. These vectors capture the semantic meaning of text, allowing documents with similar meanings to be represented by nearby points in a high-dimensional vector space.

In Retrieval-Augmented Generation (RAG) systems, embeddings play a fundamental role in the retrieval process. Before documents can be searched efficiently, they are transformed into vector representations and stored in a vector database. When a user submits a query, the same embedding model converts the query into a vector. Similarity search is then performed to identify the document chunks whose embeddings are closest to the query embedding. This enables semantic retrieval, allowing the system to find relevant information even when the exact keywords used in the query do not appear in the documents.

The quality of the embedding model directly affects retrieval performance. High-quality embeddings improve the system's ability to identify contextually relevant documents, which in turn enhances the accuracy and reliability of the generated responses.

2.4 Text Chunking

Before indexing, documents must be segmented into chunks contiguous passages serving as atomic retrieval units. Chunking strategy significantly affects retrieval quality. Common approaches include fixed-size chunking, sentence-based chunking, and semantic chunking.

2.5 AI in Education

The integration of artificial intelligence into education has gained significant momentum over the past decade. Early systems focused on adaptive learning platforms that adjusted the difficulty of exercises based on student performance. More recent work has explored the use of LLMs to power conversational tutors, automated essay graders, and question generation tools. However, most of these systems either rely on cloud APIs, require expensive infrastructure, or operate on general knowledge rather than specific course content. This gap motivates the need for a lightweight, locally deployable assistant that is grounded in the actual materials students are expected to study.

2.5.1 Intelligent Tutoring Systems

Intelligent Tutoring Systems (ITS) have been an active research area since the 1970s, rooted in Bloom's observation that one-on-one human tutoring produces learning gains of up to two standard deviations above conventional classroom instruction [13]. Early systems such as GUIDON [14], developed at Stanford for teaching medical diagnosis, and the LISP Tutor [15], built at Carnegie Mellon to teach computer programming, relied on hand-crafted rule-based knowledge bases that required enormous expert effort to build and maintain.

The 1990s and 2000s brought a shift toward data-driven systems. The **Cognitive Tutor** [16], deployed at scale in high school algebra classrooms, used a student model derived from cognitive science to track knowledge components in real time and skip material the student had already mastered. Studies reported effect sizes of 0.30–0.40 in mathematics and science compared to traditional instruction. **ALEKS** (Assessment and Learning in Knowledge Spaces) [17], grounded in knowledge space theory, dynamically adjusted the sequence of learning materials based on student performance; Hagerty and Smith reported approximately 25% score improvements in college-level mathematics courses [18]. **AutoTutor** [19], developed by Graesser et al., went further by engaging students in natural-language dialogue about conceptual questions in physics and computer literacy, achieving learning gains comparable to human tutors with effect sizes around 0.8 standard deviation units across more than 20 controlled experiments.

The emergence of Large Language Models has opened a new chapter for ITS, enabling conversational AI capable of engaging students in Socratic dialogue and providing immediate, personalised feedback at scale. **Khanmigo**, Khan Academy's LLM-powered tutoring assistant launched in 2023, guides students through problems using hints and questions rather than giving direct answers, preserving the cognitive effort that drives learning [1]. **Carnegie Learning's LiveHint** [20], trained on data from over 5.5 million students working through 1.2 billion math problems, provides on-demand step-by-step hints within its MATHia platform for middle and high school students. In research settings, Slade et al. [2] demonstrated that students who used an LLM-based tutoring assistant to complete a psychology writing assignment scored higher on post-task retention quizzes than students who worked without any AI assistance, suggesting that conversational AI tutoring actively supports learning rather than replacing it.

Despite this progress, a common limitation persists across most deployed systems: they either depend on cloud APIs and paid subscriptions, operate on general-purpose knowledge disconnected from any specific course, or require significant infrastructure and technical expertise to deploy. These constraints make them poorly suited for resource-limited academic environments, and represent the gap that recent research efforts are beginning to address.

2.5.2 RAG-Based Educational Assistants

The application of Retrieval-Augmented Generation to educational settings has attracted growing research attention since 2023, and early empirical results are consistently encouraging. Lewis et al. [9], who introduced the RAG architecture, established its core advantage: by grounding generation in dynamically retrieved passages from a verified document collection, the model is constrained to produce responses that are traceable and far less prone to hallucination than a standalone LLM.

In the educational domain specifically, Slade et al. [2] conducted a controlled experiment with 109 undergraduate psychology students assigned to one of three conditions: a RAG-based AI tutor, unmodified GPT-4 Turbo, or no AI assistance. Contrary to the expectation that AI reliance would harm retention, students who used the RAG tutor scored higher on post-task quizzes than those who received no AI support at all. The authors attributed this to the RAG system's ability to produce course-grounded, verifiable explanations rather than fluent but unanchored text.

On the instructor side, Dakshit [21] conducted the first study gathering faculty feedback on RAG systems deployed in computer science higher education. Faculty members broadly acknowledged the potential of RAG as both a virtual teaching assistant for students and a teaching aid for lecturers, while also identifying the need for ethical guardrails and better transparency mechanisms before full-scale deployment.

Levonian et al. [22] studied RAG for math question-answering and introduced an important distinction between *groundedness* whether a response can be traced to the retrieved document and *faithfulness* whether it correctly answers the query based on that document. Their findings showed that prompt design plays a critical role in navigating this trade-off, and that retrieval relevance alone is not a reliable predictor of response quality from the student's perspective.

From the faculty perspective, a survey by Gao et al. [10] synthesized the broader RAG literature and concluded that retrieval-grounded responses consistently outperform pure LLM responses on factual accuracy metrics, particularly in domain-specific applications such as education, medicine, and law. Shuster et al. [23] further showed that retrieval augmentation reduces hallucination rates in conversational settings, which is a finding directly applicable to student-facing Q&A systems where a wrong answer is not just unhelpful but potentially harmful.

Taken together, this body of work establishes a clear consensus: RAG is currently the most practical and reliable architecture for building educational assistants that must stay faithful to specific course content, cite their sources, and operate without access to large cloud-based models. These findings directly motivate the design of TeachBot.

Chapter 3

Analysis and Design

3.1 Introduction

The previous chapter established that while RAG-based educational assistants have shown consistent promise in the literature from the course-grounded tutor evaluated by Slade et al. [2] to the faculty-facing systems surveyed by Dakshit [21] a clear gap remains. Most existing systems depend on cloud APIs, require paid subscriptions, or demand infrastructure that is simply beyond reach of universities with limited resources. The goal of this project is to close that gap: to build a fully local, teaching assistant that any student can run on a standard machine, without sending a single document to an external server also the ability to deploy it at the universities and colleges .

This chapter translates that goal into a concrete design. We begin with the general architecture of the system, describing how inputs flow through the system and how outputs are produced. We then detail the RAG pipeline design, covering both the indexing pipeline that transforms raw PDF documents into searchable vector embeddings,and the query pipeline that retrieves relevant content and generates grounded,source-attributed answers in response to a student's natural language question.

3.2 Global Architecture

The system is built around two distinct pipelines that share the same underlying infrastructure: an **indexing pipeline** that processes uploaded course documents, and a **query pipeline** that handles student requests at runtime. Both pipelines converge on a common vector store and a locally deployed language model. Figure 3.1 illustrates the three-tier layered organization of the system, and the complete data flow from input to output.

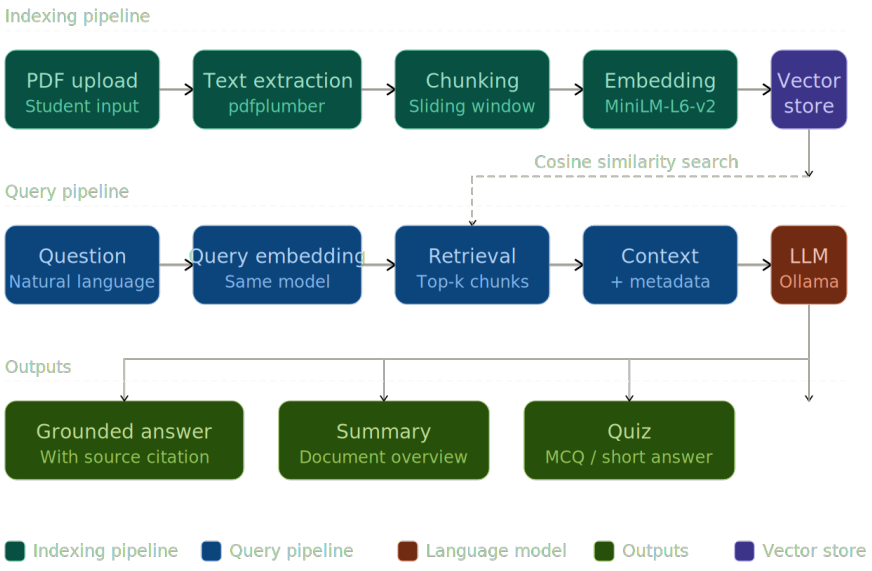

Figure 3.1: General Data Flow of the System

3.3 RAG Pipeline Design

The RAG pipeline is the core mechanism of the system. It is divided into two sub-pipelines that operate sequentially: the indexing pipeline, which runs once per document at upload time, and the query pipeline, which runs on every student request.

3.3.1 Indexing Pipeline

The indexing pipeline is the offline component of the system. It is triggered once per document at upload time and transforms a raw PDF file into a structured set of searchable vector representations that persist in the knowledge base. The pipeline is designed so that once a document has been indexed, it never needs to be processed again all future queries against that document are served entirely from the vector database. It operates in four sequential stages, as illustrated in Figure 3.1.

Stage 1 Text Extraction. The first stage is responsible for converting the binary PDF format into plain, processable text. A PDF document is not a simple text file it is a structured container that encodes text, fonts, layout instructions, images, and vector graphics together. Extracting the textual content requires parsing this structure and recovering the character sequences that a human reader would see on each page.

The extraction is performed page by page rather than on the document as a whole. This granularity is intentional: by processing one page at a time, the system can associate each extracted text block with the exact page number it came from. This page number is recorded as metadata and carried forward through every subsequent stage of the pipeline. It ultimately surfaces in the system's responses as a source citation, allowing the student to locate the exact passage in the original document that was used to generate the answer.

Non-textual elements images, diagrams, mathematical figures, and embedded graphics are not extracted at this stage. The pipeline operates exclusively on the textual layer of the document. This is a deliberate design constraint that keeps the system lightweight and computationally inexpensive, at the cost of not being able to answer questions that require understanding of visual content.

Stage 2 Chunking. The complete text extracted from a document may span dozens or hundreds of pages and contain tens of thousands of words. Embedding and retrieving this entire text as a single unit is not feasible for two reasons. First, embedding models have a maximum input length, beyond which text is truncated and information is lost. Second, and more importantly, retrieving a large block of text as a single retrieval unit is counterproductive: the language model would receive an enormous context containing mostly irrelevant content, which degrades the quality of the generated response and increases processing time.

The solution is to segment the document into smaller, self-contained passages called chunks. Each chunk serves as an independent atomic retrieval unit when the system searches for relevant content, it retrieves individual chunks rather than entire documents. The chunking algorithm uses a sliding window approach: starting from the beginning of the extracted text, a window of fixed length is extracted as a chunk, and then the window advances forward by a step that is smaller than the window size. This produces an overlap between consecutive chunks.

The overlap is not incidental it is a deliberate mechanism to handle boundary effects. If a key concept or a complete sentence happens to fall exactly at the boundary between two non-overlapping chunks, it would be split across both and potentially missed by the retrieval system. By making consecutive chunks share a portion of their content, the overlap guarantees that any passage of text of a length smaller than the chunk size will appear in at least one chunk in its entirety. Each chunk

also inherits the page number of the page from which its opening text originates, preserving the source attribution chain established in Stage 1.

Stage 3 Embedding. Raw text chunks cannot be directly compared for semantic similarity text is a symbolic, discrete structure, and there is no natural mathematical operation that tells us how similar two sentences are in meaning. To enable semantic comparison, each chunk must be converted into a numerical representation that captures its meaning in a continuous vector space. This conversion is performed by a sentence embedding model.

A sentence embedding model is a neural network trained to map variable-length text inputs to fixed-size dense vectors, such that texts with similar meanings are mapped to vectors that are geometrically close to one another. This property semantic proximity in vector space is the foundation of the entire retrieval mechanism. It means that when a student asks a question using different words than those used in the course document, the system can still find the relevant passage, because both the question and the passage will be mapped to nearby regions of the vector space.

A critical design constraint is that the same embedding model must be used for both document chunks during indexing and student queries at retrieval time. If different models were used, the resulting vectors would live in incompatible spaces and similarity comparisons between them would be meaningless. The embedding model is therefore a shared component of both the indexing pipeline and the query pipeline.

Stage 4 Indexing. In the final stage, the dense vectors produced by the embedding model are stored persistently in the vector database together with their associated metadata: the document identifier, document title, page number, and chunk index within the document. This metadata record is what transforms the vector database from a simple numerical index into a knowledge base capable of generating verifiable, source-attributed responses.

The vector database organizes the stored embeddings in a data structure optimized for approximate nearest-neighbor search a class of algorithms that can efficiently find the vectors most similar to a given query vector among potentially millions of stored vectors, without exhaustively comparing the query against every entry. Similarity between vectors is measured using cosine similarity, which computes the cosine of the angle between two vectors rather than their absolute distance, making it invariant to differences in text length between chunks of varying verbosity. A cosine similarity value of 1 indicates maximum semantic overlap, while a value of 0 indicates no shared semantic content.

Once Stage 4 is complete, the document is fully indexed. Its content is available for retrieval across all future student queries without any reprocessing of the original file. Adding a new document to the system requires only running it through the four stages above the existing index is not affected and no retraining of any model is required.

3.3.2 Query Pipeline

The query pipeline processes a student's natural language question in five steps and returns a grounded, source-attributed answer.

Step 1 Query Embedding. The question is encoded into a dimensional vector using the same model used during indexing. Also for both documents and queries is essential: it guarantees that semantically related content will have similar vector representations, regardless of whether it comes from a question or a document chunk. **Step 2 Semantic Retrieval.** Our vectors compares the query vector against all stored chunk vectors using cosine similarity, defined as:

$$\text{sim}(\mathbf{q}, \mathbf{d}) = \frac{\mathbf{q} \cdot \mathbf{d}}{\|\mathbf{q}\|\|\mathbf{d}\|} \tag{3.1}$$

where $\mathbf{q}$ is the query vector and $\mathbf{d}$ is a document chunk vector. The top-k chunks with the highest similarity scores are returned as the retrieval result.

Step 3 Context Assembly. The retrieved chunks are assembled into a structured context block. Each chunk is prefixed with its source metadata document title and page number so that the language model can incorporate this information into its response and the system can present verifiable citations to the student.

Step 4 Prompt Construction. The context block and the original question are combined into a prompt following a strict instruction template. The template explicitly instructs the language model to answer only based on the provided context, and to state when the context does not contain sufficient information to answer the question. This constraint is the primary mechanism that keeps responses grounded and prevents hallucination.

Step 5 Response Generation. The prompt is sent to the language model served locally via Ollama using an HTTP request to its OpenAI-compatible API endpoint. The model generates a natural language response conditioned on the retrieved context. The backend extracts the source references from the retrieval metadata and appends them to the response before returning the complete answer to the frontend.

3.4 Conclusion

This chapter has defined the system requirements and presented the architecture design of TeachBot. The three-tier architecture provides clean separation of concerns, and the RAG pipeline design,in the next chapter we will present the implementation of this design .

Chapter 4

Implementation

4.1 Introduction

This chapter presents the complete technical realization of the system. Building on the architecture and design decisions established in the previous chapter, we describe how each component was concretely implemented: the document processing pipeline, the RAG engine, the quiz generator, the REST API, the React frontend, and the Docker deployment strategy. For each component we explain the implementation choices made, the libraries and technologies selected, and how the components interact to form a cohesive, fully local application.

4.2 Development Environment

The system was developed and tested on the workstation described in Table 4.1. The high-performance GPU was used exclusively during benchmarking to evaluate larger models; the system is also designed to run on CPU-only machines, as demonstrated during development.

Table 4.1: Development and Deployment Environment

Component	**Specification**
Operating System	OS Big Sur version 11.7.11
Processor	2 GHz Intel Core i7 quadro cores
RAM	8 GB DDR3
GPU	Intel Iris Pro 1536 Mo
Containerization	Docker Desktop (WSL2 backend)
Code Editor	Visual Studio Code
Version Control	Git / GitHub

4.3 Technology Stack

Table 4.2 lists the full technology stack used across all layers of the system, along with the version and the role of each component.

Table 4.2: Full Technology Stack

Layer	**Technology**	**Role**
Backend	Python 3.11 + FastAPI 0.104	REST API server and RAG orchestration
Vector Database	ChromaDB 0.5.23	Embedding storage and semantic retrieval
PDF Processing	pdfplumber 0.10.3	Primary text extraction from PDFs
PDF Fallback	pypdf	Fallback extractor for malformed PDFs
HTTP Client	httpx 0.28.1	Asynchronous calls to Ollama API
LLM Server	Ollama	Local language model inference server
Embedding Model	all-MiniLM-L6-v2 (ONNX)	384-dimensional sentence embeddings
Frontend	React 18 + Vite 5	Single-page web application
Web Server	Nginx 1.24	SPA serving and reverse proxy
Containerization	Docker Compose	Multi-service orchestration

4.4 Three-Tier Architecture

The system follows a classic three-tier architecture composed of a presentation layer, a business logic layer, and a data layer. All three tiers are containerized and orchestrated via Docker Compose, allowing the entire stack to be launched with a single `docker compose up` command.

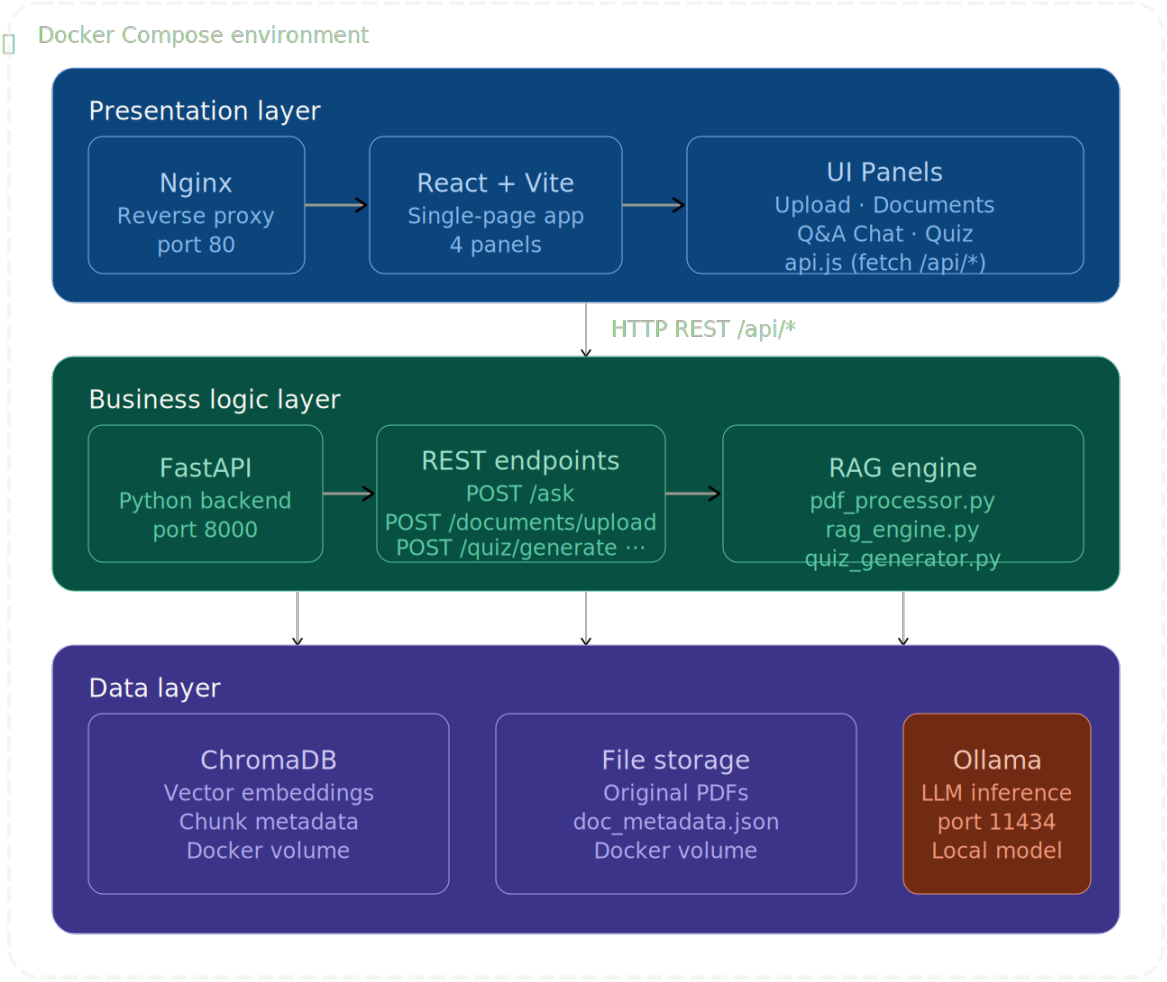

Figure 4.1: Three-Tier Architecture of the System

4.4.1 Presentation Layer React Frontend

The presentation layer is a Single-Page Application (SPA) built with React 18 and Vite 5, served by Nginx inside a dedicated Docker container on port 80. Nginx acts as a reverse proxy: it serves the compiled React assets to the browser and transparently forwards any HTTP request whose path begins with `/api/` to the FastAPI backend container on the internal Docker network, keeping the backend completely hidden from the student's browser.

4.4.2 Business Logic Layer FastAPI Backend

The business logic layer is implemented in Python 3.11 using the FastAPI framework, running inside its own container on port 8000. FastAPI was chosen for its native support of asynchronous request handling, its automatic OpenAPI documentation generation, and its clean Pydantic-based request validation. The backend exposes seven REST endpoints and coordinates all RAG operations by orchestrating three internal modules: `pdf_processor.py`, `rag_engine.py`, and `quiz_generator.py`.

4.4.3 Data Layer ChromaDB and File Storage

The data layer consists of three persistent components, all mounted as Docker bind-mount volumes to ensure data survives container restarts and system reboots. ChromaDB stores all vector embeddings and their associated chunk metadata. A local `uploads/` directory stores the original uploaded files. A `doc_metadata.json` file maintains document-level records including titles, page counts, and chunk counts.

4.5 Document Processing

The document processing module (`pdf_processor.py`) is responsible for converting raw uploaded files into structured, chunked text ready for embedding. It handles both PDF and plain text (TXT) files up to 50 MB in size.

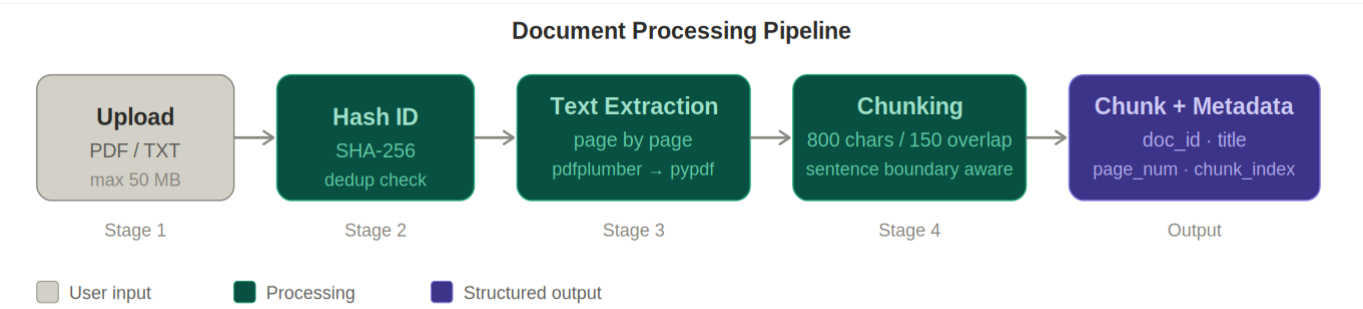

Figure 4.2: Document Processing Pipeline

4.5.1 File Validation and Deduplication

When a file is received by the backend, its content is read into memory and a 16-character SHA-256 hash is computed from the raw bytes. This hash serves as the document's unique identifier (`doc_id`) throughout the system. Before proceeding, the backend checks ChromaDB for any existing entries with this `doc_id`; if the document has already been indexed, the upload returns immediately without reprocessing. This deduplication mechanism prevents redundant indexing when a student accidentally uploads the same file twice.

4.5.2 Text Extraction

Text is extracted using `pdfplumber` as the primary engine, which provides reliable page-by-page extraction while preserving the reading order of text blocks. For each page, the extracted text undergoes light cleaning: sequences of more than two consecutive newlines are collapsed into a single blank line, and sequences of more than two consecutive spaces are normalized to a single space. Pages that yield no extractable text (such as pages containing only images or scanned content) are silently skipped.

If `pdfplumber` fails for example on malformed or encrypted PDFs the system automatically falls back to `pypdf`, a pure-Python PDF reader with broader compatibility. Each successfully extracted page is stored as a dictionary containing its text content and page number, which is carried as metadata through all subsequent stages.

For plain text files (`.txt`), the content is read directly and treated as a single-page document with page number set to 1.

4.5.3 Chunking Algorithm

The extracted text is segmented into overlapping fixed-size chunks using a custom sliding window algorithm, illustrated in Figure 4.3.

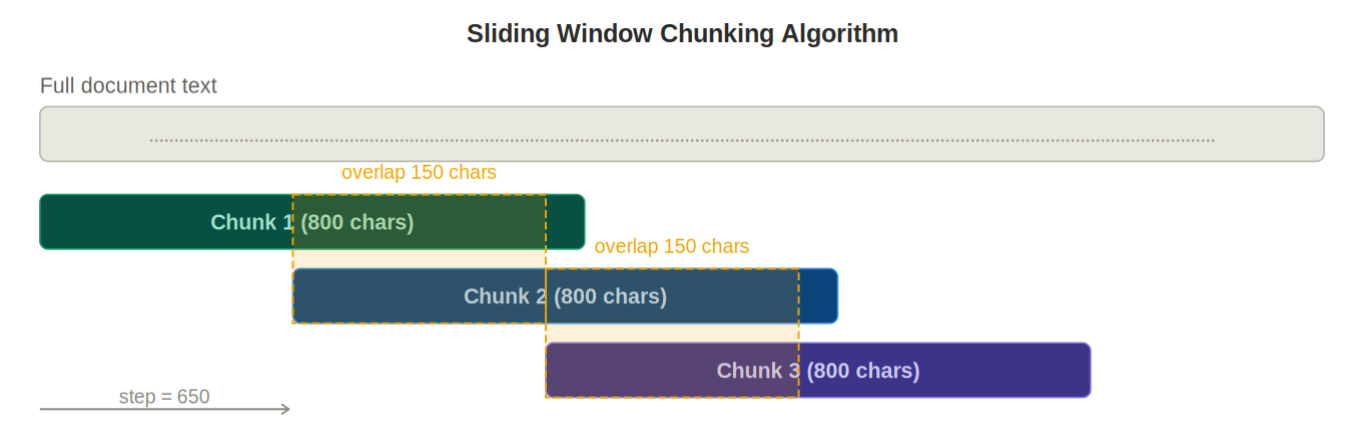

Figure 4.3: Sliding Window Chunking with Overlap

The algorithm uses a window of 800 characters and an overlap of 150 characters, giving an effective step size of 650 characters. At each step, before fixing the chunk boundary, the algorithm attempts to break at a natural sentence boundary by searching backward from the hard cutoff for a paragraph break, a line break, a period-space sequence, or a word space. This sentence-aware breaking prevents the system from cutting words in half and improves the coherence of individual chunks. Chunks shorter than 50 characters typically artifacts from page headers or footers are discarded. Each chunk is stored with its full metadata: `doc_id`, `doc_title`, `page_num`, and `chunk_index`.

4.6 RAG Engine

The RAG engine (`rag_engine.py`) is the central processing module of the system. It manages all interactions with ChromaDB and the language model, and implements the full retrieval-augmented generation pipeline for question-answering and summarization.

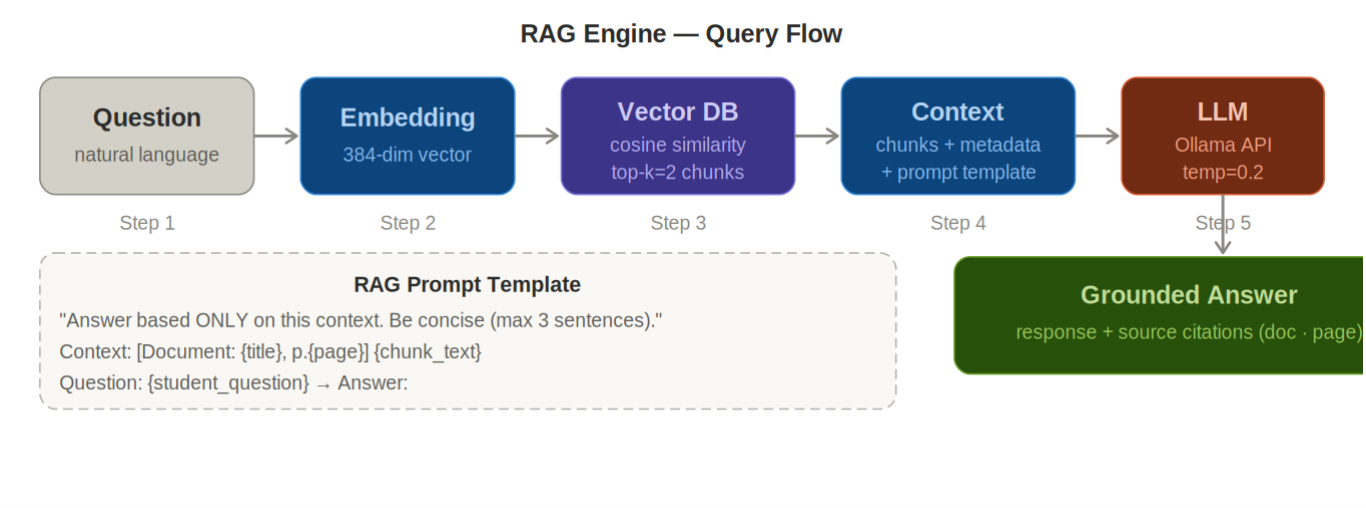

Figure 4.4: RAG Engine Query Processing Flow

4.6.1 Vector Indexing

After chunking, the document's chunks are inserted into ChromaDB. ChromaDB automatically computes the embedding for each chunk using the `all-MiniLM-L6-v2` model via the ONNX runtime, eliminating the need for a separate embedding step in the application code. To prevent memory overflow when indexing large documents, chunks are inserted in batches of 50. The collection is configured to use cosine similarity as the distance metric, which is specified at collection creation time via the `hnsw:space=cosine` metadata parameter.

4.6.2 Semantic Retrieval

When a student submits a question, ChromaDB embeds the query using the same model and performs an approximate nearest-neighbor search across all stored chunk vectors, returning the top-k most similar chunks along with their cosine similarity scores and metadata. The similarity score returned by ChromaDB is a distance (lower is better), which the engine converts to a similarity score between 0 and 1 by computing $1 - \text{distance}$.

The retrieval supports optional document filtering: if the student selects specific documents in the interface, the query is restricted to chunks belonging to those documents using ChromaDB's `where` filter. This allows students to direct their questions at specific course materials rather than the entire knowledge base.

4.6.3 Prompt Design and LLM Integration

The retrieved chunks are assembled into a structured prompt that is sent to the language model. The prompt follows a strict template that instructs the model to answer based exclusively on the provided context and to limit its response to three sentences. This constraint is the primary mechanism that grounds the model's output in the course material and prevents hallucination. Each chunk in the context is prefixed with the document title and page number, which the model incorporates into its response, enabling automatic source attribution without any post-processing.

The language model is accessed via an HTTP POST request to Ollama's OpenAI-compatible API endpoint at `/v1/chat/completions`. The request uses a temperature of 0.2 low enough to produce consistent, factual responses while avoiding the determinism of temperature 0. A timeout of 300 seconds is set to accommodate inference on CPU-only hardware. The model to use is configurable via the `LLM_MODEL` environment variable, making it straightforward to switch between models without modifying application code.

4.6.4 Summarization

Document summarization follows a simplified version of the same pipeline. Rather than embedding a query and retrieving chunks, the system reads the first 2000 characters of the document as a representative preview and passes this directly to the language model with a prompt requesting a three-point bullet summary. This approach is fast and effective for introductory summaries, though it is limited to content appearing near the beginning of the document.

4.7 Quiz Generator

The quiz generation module (`quiz_generator.py`) produces assessment questions grounded in the indexed course materials. It supports two question types: multiple-choice questions (MCQ) and short-answer questions. Figure 4.5 illustrates the generation flow.

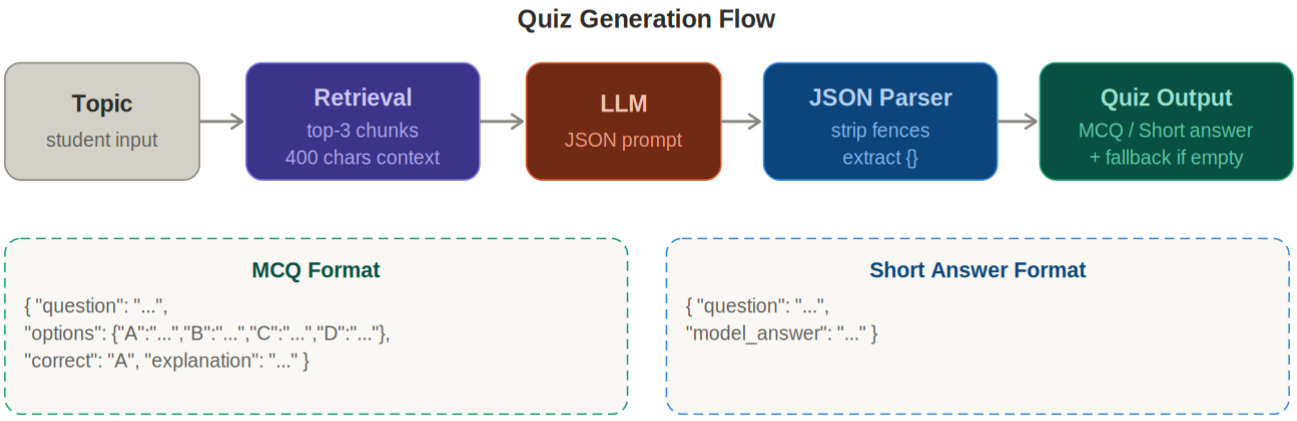

Figure 4.5: Quiz Generation Flow

4.7.1 Multiple-Choice Questions

For MCQ generation, the system first retrieves the top-3 most relevant chunks for the requested topic. To keep the prompt compact enough for smaller language models to handle reliably, only the first 400 characters of the top chunk are used as context. The model is prompted to return a single question with four labeled options (A through D), a correct answer key, and a brief explanation in strict JSON format. This generation loop runs up to three times to produce the requested number of questions. A JSON extraction utility strips markdown fences and locates the outermost `{}` pair in the raw model output, handling the imperfect JSON formatting that smaller models often produce. If the model fails to generate any valid questions after all attempts, the system returns a deterministic fallback question constructed directly from the retrieved chunk text.

4.7.2 Short-Answer Questions

Short-answer generation follows the same pattern, with a simpler JSON schema requiring only a question string and a model answer string. The same JSON extraction and fallback mechanisms apply.

4.8 REST API

The FastAPI backend exposes seven REST endpoints organized across four functional groups: document management, question answering, summarization, and quiz generation. Figure 4.6 illustrates all endpoints.

REST API Endpoints

Method	Endpoint	Description
POST	/documents/upload	Upload and index a PDF or TXT file (max 50 MB)
GET	/documents	List all indexed documents with metadata
DELETE	/documents/{id}	Remove a document and all its indexed chunks
POST	/ask	Answer a student question using RAG pipeline
POST	/summarize	Generate a summary of a selected document
POST	/quiz/generate	Generate MCQ or short-answer quiz from content
GET	/health	System health check — indexed chunks count

Figure 4.6: REST API Endpoints

All request and response bodies are validated using Pydantic models. The `/documents/upload` endpoint enforces a 50 MB file size limit and accepts both PDF and TXT files. The `/ask` endpoint accepts an optional `doc_ids` list to restrict retrieval to specific documents. The `/quiz/generate` endpoint accepts a `quiz_type` parameter (`"mcq"` or `"short_answer"`) and a `num_questions` pa-

rameter capped at 20. The `/health` endpoint returns the current number of indexed chunks and documents, providing a lightweight way for the frontend to verify backend connectivity on startup.

4.9 Frontend

The frontend is a React 18 Single-Page Application built and bundled with Vite 5. Figure 4.7 shows the component architecture.

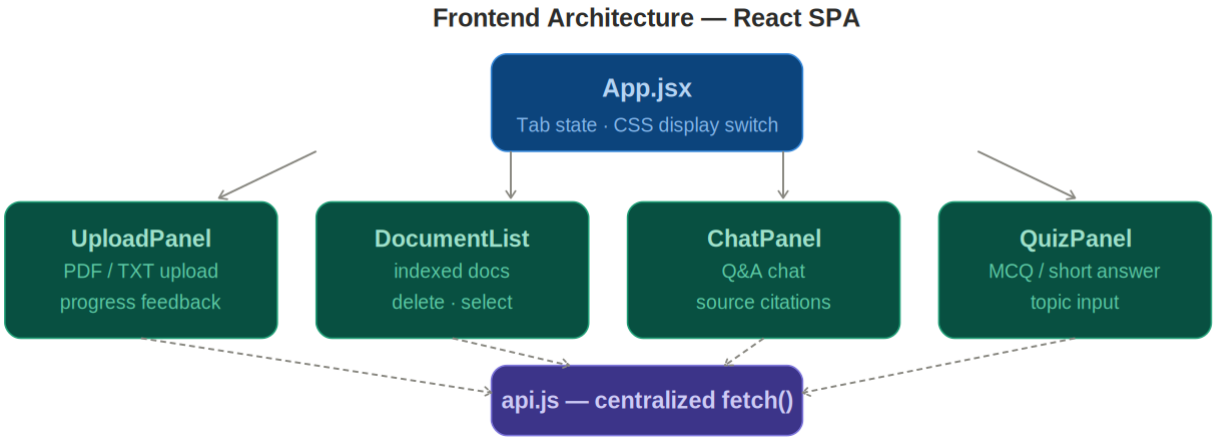

Figure 4.7: Frontend Component Architecture

4.9.1 Tab State Management

The application is organized into four panels: Upload, Documents, Chat, and Quiz. A key implementation decision was to use CSS `display` switching (`flex` / `none`) rather than conditional rendering to manage panel visibility. This means that all four panel components are always mounted in the DOM; only their visibility is toggled. The practical benefit is that the chat history in the Chat panel and the generated quiz in the Quiz panel are preserved when the student navigates to another tab and returns a behavior that conditional rendering would break by unmounting and remounting the components.

4.9.2 Centralized API Communication

All HTTP calls to the backend are centralized in a single `api.js` module that exports named functions for each operation: `uploadDocument`, `getDocuments`, `deleteDocument`, `askQuestion`, `summarizeDocument`, and `generateQuiz`. Every function issues a `fetch()` call to the corresponding `/api/*` route, which Nginx transparently proxies to the FastAPI backend. Centralizing API calls in a single module keeps the panel components clean and makes it trivial to update endpoint paths or add request headers in one place.

4.10 Docker Deployment

The full system is containerized using Docker Compose, enabling one-command deployment on any machine with Docker installed. Figure 4.8 shows the container architecture and startup dependency chain.

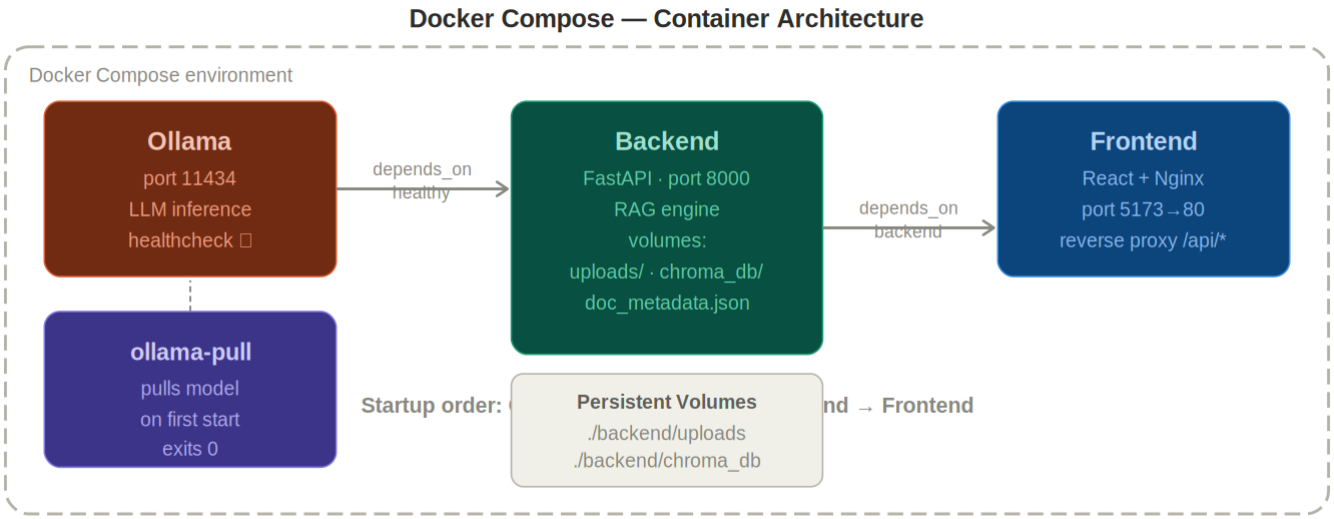

Figure 4.8: Docker Compose Container Architecture and Startup Order

4.10.1 Container Services

Four services are defined in `docker-compose.yml`:

- **ollama**: The Ollama LLM inference server, exposed on port 11434. It is configured with a context window of 1024 tokens (`OLLAMA_NUM_CTX=1024`) and equipped with a health check that polls `ollama list` every 10 seconds until the service is confirmed healthy.
- **ollama-pull**: A one-shot service that runs after Ollama becomes healthy and pulls the configured language model into the Ollama container. It exits with code 0 after the pull completes and does not restart.
- **backend**: The FastAPI application, built from the `./backend` Dockerfile and exposed on port 8000. It starts only after Ollama is healthy and receives the Ollama API URL and model name as environment variables, making the model selection configurable without rebuilding the image.
- **frontend**: The React application served by Nginx, mapped from container port 80 to host port 5173. It starts after the backend service is available.

4.10.2 Data Persistence

Three bind-mount volumes ensure that all indexed data persists across container restarts. The `uploads/` directory retains the original uploaded files. The `chroma_db/` directory retains the Chro-

maDB vector index. The `doc_metadata.json` file retains document-level metadata. With these mounts in place, students can stop and restart the entire Docker Compose stack without losing any previously indexed documents.

4.11 Conclusion

This chapter has detailed the complete technical implementation of the system. The modular architecture cleanly separates document processing, RAG logic, quiz generation, API routing, and frontend rendering into independent, maintainable components. The containerized deployment strategy via Docker Compose ensures full reproducibility across different host machines. The configuration-driven model selection allows switching between language models without modifying application code a property that proved essential during the benchmarking phase presented in the following chapter.

Chapter 5

Experimentation and Evaluation

5.1 Introduction

This chapter presents the experimental evaluation of the system across two distinct hardware environments: a MacBook Pro running on a CPU-only configuration, and a high-performance desktop workstation equipped with a dedicated GPU. The two-phase evaluation strategy was deliberate. The first phase, conducted on the MacBook Pro, tested the system under realistic resource-constrained conditions representative of many Algerian university settings. The second phase, conducted on the GPU workstation, evaluated the system at full capacity by benchmarking four language models across 100 questions each on a diverse multi-domain corpus, producing statistically robust results.

The chapter is organized as follows. Section 5.2 describes both hardware configurations. Section 5.3 presents the test corpus. Section 5.4 defines the evaluation metrics. Section 5.5 presents the Phase 1 (Mac) results. Section 5.6 presents the Phase 2 (GPU) full four-model benchmark. Section 5.7 justifies the final model selection. Section 5.8 compares both hardware environments. Section 5.9 discusses strengths, limitations.

5.2 Experimental Setup

5.2.1 Phase 1 MacBook Pro (CPU-Only)

The first evaluation phase was conducted on a MacBook Pro running macOS Big Sur 11.7.10 with a 2 GHz Intel Core i7 quad-core processor and 8 GB of DDR3 RAM. No GPU acceleration was available; all LLM inference ran on CPU only. The model evaluated in this phase was **Phi-3-Mini** (3.8B parameters, 2.0 GB), served via Ollama. This configuration represents the hardware available to most students and instructors at regional universities in southern Algeria and serves as the accessibility baseline.

Table 5.1: Experimental Configuration Phase 1 (MacBook Pro, CPU-Only)

Parameter	**Value**
Machine	MacBook Pro (macOS Big Sur 11.7.10)
Processor	2 GHz Intel Core i7 (4 cores, x86-64)
RAM	8 GB DDR3
GPU	None (CPU-only inference)
LLM Model	Phi-3-Mini (3.8B parameters, 2.0 GB)
Embedding Model	all-MiniLM-L6-v2 (384 dimensions, ONNX)
Vector Database	ChromaDB 0.5.23, cosine similarity
Chunk size / Overlap	800 chars / 150 chars
Top-k retrieval	2 chunks
Temperature	0.2
Questions evaluated	30 (3 runs $\times$ 10 questions)

5.2.2 Phase 2 GPU Workstation (Four-Model Benchmark)

The second evaluation phase was conducted on a high-performance desktop workstation running Windows 11 Pro with an Intel Core i9-14900K processor, 64 GB of DDR5 RAM, and an NVIDIA

RTX 5070 Ti GPU with 16 GB of VRAM under CUDA 13.1. Four language models were benchmarked: **TinyLlama** (1.1B), **Phi-3-Mini** (3.8B), **Mistral-Nemo** (12B), and **Llama3.1** (8B). Each model was evaluated on the same 100 questions across the same document corpus, enabling a fair direct comparison.

Table 5.2: Experimental Configuration Phase 2 (GPU Workstation)

Parameter	**Value**
Machine	Desktop Workstation (Windows 11 Pro, WSL2)
Processor	Intel Core i9-14900K (24 cores)
RAM	64 GB DDR5
GPU	NVIDIA RTX 5070 Ti (16 GB VRAM, CUDA 13.1)
Models benchmarked	TinyLlama 1.1B, Phi-3-Mini 3.8B, Mistral-Nemo 12B, Llama3.1 8B
Embedding Model	all-MiniLM-L6-v2 (384 dimensions, ONNX)
Vector Database	ChromaDB 0.5.23, cosine similarity
Chunk size / Overlap	800 chars / 150 chars
Top-k retrieval	2 chunks
Temperature	0.2
Questions per model	100 (automated benchmark)

5.3 Test Corpus

Phase 1 used three computer science course PDFs totalling 103 pages and 103 indexed chunks. Phase 2 expanded the corpus to 13 academic domains to test the system's generalization across diverse subject areas. All documents were real university course materials.

Table 5.3: Test Document Collection

Document / Domain	**Pages**	**Phase**	**Subject Area**
DSS Chapitre 3.2 DTD	29	1 & 2	XML / DTD
Java Encapsulation	34	1 & 2	Object-Oriented Programming
Introduction to Image Processing	40	1 & 2	Computer Vision
Aviation Phraseology	102	2	Aviation Communication
Flight Planning Procedures	164	2	Aviation Navigation
Operational Procedures	162	2	Aviation Operations
Aeronautical Medicine	7	2	Aviation Medicine
Electronics (Synchronous Counters)	42	2	Digital Electronics
Fluid Dynamics	8	2	Engineering Physics
Phase 1 total	103		3 domains, 30 questions
Phase 2 total	691		13 domains, 100 questions

5.4 Evaluation Metrics

Each generated answer was assessed on the following metrics:

- **Faithfulness**: Binary whether the answer is grounded in the retrieved context and introduces no unsupported claims.
- **Relevance**: Binary whether the answer actually addresses the question, as opposed to being faithful but off-topic.
- **Citation rate**: Binary whether the response includes a valid source attribution (document title and page number).
- **Keyword score**: Continuous $[0, 1]$ the proportion of domain-specific keywords expected in a correct answer that appear in the generated response.
- **Response latency**: Wall-clock time in seconds from request to complete response.
- **Throughput**: Tokens generated per second.
- **Quiz valid JSON rate**: Proportion of quiz requests returning a correctly structured, parseable JSON response.

5.5 Phase 1 Results MacBook Pro (Phi-3-Mini)

Phase 1 consisted of three benchmark runs of 10 questions each (N=30 total), using the three computer science documents. The same 10 questions were used across all runs to assess consistency.

Table 5.4: Phase 1 Phi-3-Mini on MacBook Pro (N=30, 3 runs)

Metric	Value
Faithfulness rate	100%
Relevance rate	100%
Citation rate	100%
Keyword score (avg)	0.733
Avg response latency	47.85 s
Min / Max latency	17.50 s / 222.97 s
Std deviation (latency)	51.67 s
Avg throughput	1.67 tokens/s
Quiz valid JSON rate	100% (all 3 quiz runs)

Phi-3-Mini achieved perfect scores on all three quality metrics: every answer was faithful to the retrieved content, addressed the question, and included a valid source citation. The keyword score of 0.733 indicates reliable incorporation of domain-specific terminology. These results confirm that the RAG grounding mechanism functions correctly even on CPU hardware with a mid-size model.

However, the latency tells a different story. An average of 47.85 seconds per response is at the boundary of acceptable interactivity, and the standard deviation of 51.67 seconds larger than

the mean itself reveals severe unpredictability: some responses arrived in 17 seconds while others required nearly 4 minutes. This variance is a fundamental consequence of CPU-only inference. The throughput of just 1.67 tokens per second confirms the bottleneck. The quiz results (100% valid JSON) demonstrate that Phi-3-Mini produces reliable structured output even under these constraints.

5.6 Phase 2 Results GPU Workstation (Four-Model Benchmark)

Phase 2 benchmarked all four models on the same 100 questions across 13 domains. Table 5.5 presents the complete results, and Figures 5.1 and 5.2 visualize the key metrics.

Table 5.5: Four-Model Benchmark GPU Workstation (N=100 questions per model)

Model	Params	MB	Avg Lat.	Std	Tok/s	Faith.	Relev.	KW	Quiz JSON
TinyLlama	1.1B	637	0.34 s	±0.67	438.75	99%	54%	0.49	0%
Phi3-Mini	3.8B	2200	0.48 s	±1.47	176.85	97%	75%	0.60	86%
Mistral-Nemo	12B	7100	0.58 s	±0.18	66.48	100%	73%	0.61	100%
Llama3.1	8B	4900	0.68 s	±1.27	100.26	99%	77%	0.64	100%

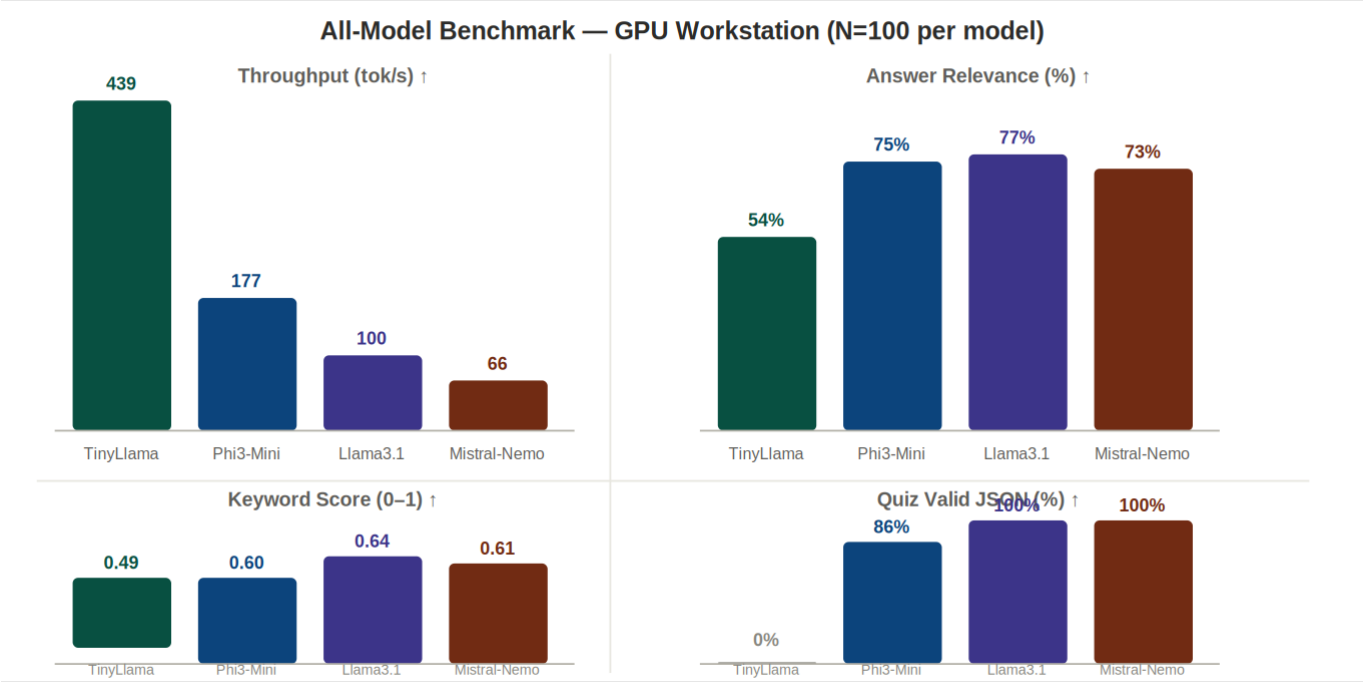

Figure 5.1: Four-Model Benchmark Throughput, Relevance, Keyword Score, and Quiz JSON Rate

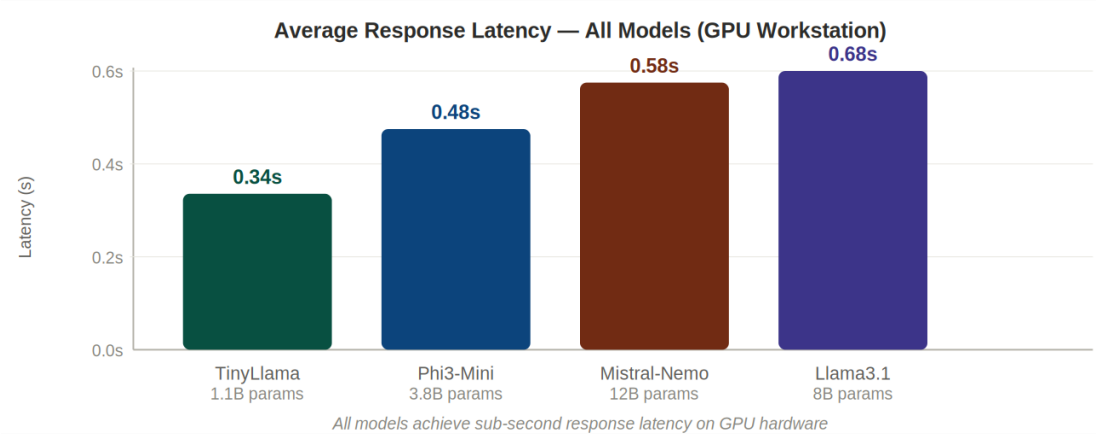

Figure 5.2: Average Response Latency per Model (GPU Workstation)

A striking observation from the GPU benchmark is that all four models respond in under one second on average a dramatic contrast with the CPU phase. GPU acceleration collapses what was a 47-second bottleneck into a sub-second interaction for every model tested. The differentiating factors on GPU hardware therefore shift entirely to output quality: faithfulness, relevance, keyword precision, and structured output reliability.

TinyLlama is the fastest by a wide margin at 438.75 tokens/s and 0.34 s average latency, but its relevance rate of only 54% means barely half its answers are on-topic. More critically, its quiz valid JSON rate is 0%: not a single one of its seven quiz generation attempts produced a parseable JSON response. TinyLlama is eliminated from production consideration on the basis of this hard failure.

Phi-3-Mini on GPU achieves 75% relevance and 86% quiz JSON better than TinyLlama, but the latency standard deviation of ±1.47 s reveals high unpredictability even on GPU hardware. Its 97% faithfulness is slightly below the other models and it did not reach 100% quiz JSON reliability.

Llama3.1 achieves the highest relevance (77%) and highest keyword score (0.64) among all models, 99% faithfulness, and perfect quiz JSON. Its average latency of 0.68 s is the slowest on the GPU, and its standard deviation of ±1.27 s is the second-highest, reflecting occasional slow outliers (max: 13.16 s).

Mistral-Nemo achieves 100% faithfulness (the only model to do so), 100% citation rate, 100% quiz JSON, and the tightest latency distribution of all models: ±0.18 s standard deviation with a maximum of just 1.36 s. This means it is not only fast but also highly predictable, which is critical for interactive use. Its relevance (73%) and keyword score (0.61) are slightly below Llama3.1 but within a few percentage points.

Figure 5.3 shows the relevance breakdown by domain for Mistral-Nemo, the selected production model. Structured technical domains achieve perfect or near-perfect scores. The 0% on System Design and General questions reflects a retrieval limitation: these questions require synthesizing information from multiple distant document sections, which the top-$k = 2$ retrieval does not capture.

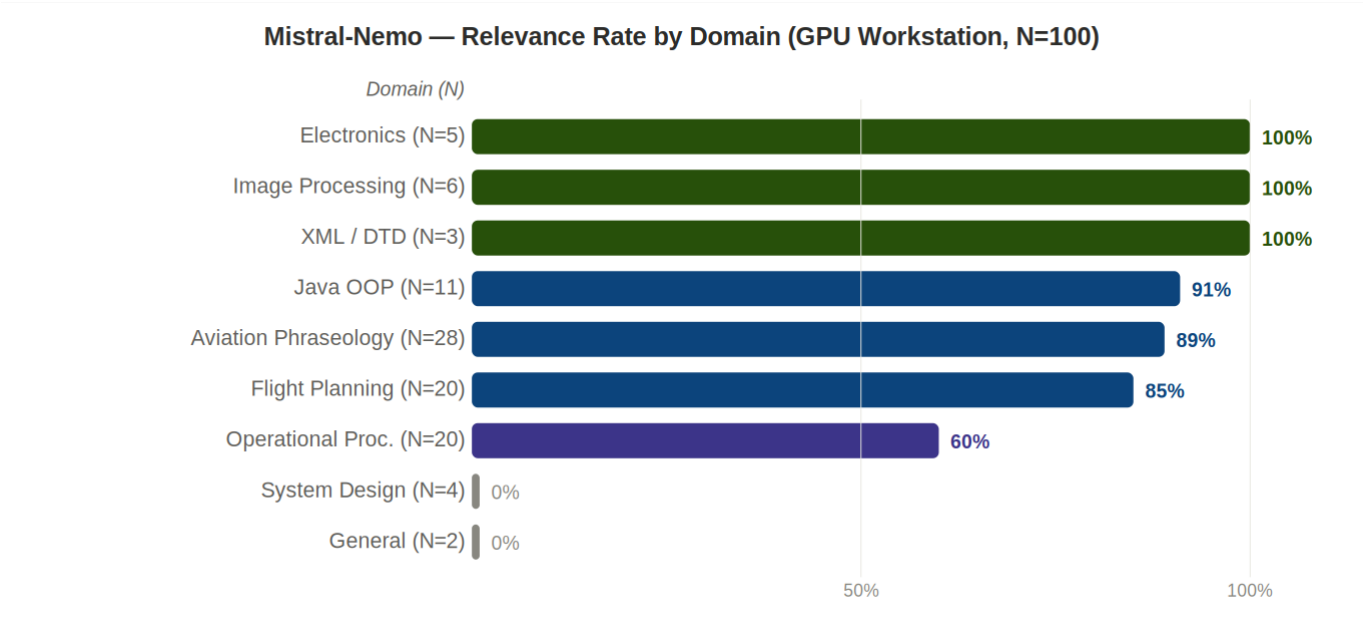

Figure 5.3: Mistral-Nemo Relevance Rate by Domain (N=100)

5.7 Model Selection Why Mistral-Nemo

Figure 5.4 presents the decision matrix used to select the production model.

Model Selection Decision Matrix — GPU Workstation (N=100 per model)

Criterion	TinyLlama	Phi3-Mini	Llama3.1	Mistral-Nemo □
Avg Latency	**0.34s** □	0.48s	0.68s	0.58s
Faithfulness	99%	97%	99%	**100%** □
Relevance	54% □	75%	**77%** □	73%
Citation Rate	100%	100%	100%	**100%** □
Keyword Score	0.49 □	0.60	**0.64** □	0.61
Quiz Valid JSON	**0%** □□	86%	100%	**100%** □
Latency Stability (std)	±0.67s	±1.47s	±1.27s	**±0.18s** □
Verdict	Eliminated	Alternative	Alternative	□ **Selected**

Figure 5.4: Model Selection Decision Matrix

Mistral-Nemo was selected as the production model for three specific reasons.

First, it is the only model among the four that achieves **100% on all three reliability metrics simultaneously**: faithfulness, citation rate, and quiz valid JSON rate. Reliability is the non-negotiable property for an educational assistant a student who receives a hallucinated answer or an uncited claim cannot verify it and may be actively misled. No amount of speed advantage justifies compromising on this.

Second, despite being the largest model tested, Mistral-Nemo has the **most stable latency distribution** of all four models: a standard deviation of just ±0.18 s and a maximum observed latency of 1.36 s across 100 questions. This consistency is what makes interactive use actually feel interactive. TinyLlama is faster on average but its occasional 5-second spikes are disruptive; Phi-3-Mini and Llama3.1 show even larger variance.

Third, Mistral-Nemo's **average latency of 0.58 s** is comfortably within the real-time interaction range. The 0.10 s difference between it and the fastest reliable model (Phi-3-Mini at 0.48 s) is imperceptible to a human user.

Llama3.1 is retained as a secondary alternative for deployments where GPU memory is limited: at 4.9 GB it requires significantly less VRAM than Mistral-Nemo's 7.1 GB, while achieving comparable reliability and slightly higher keyword precision. It is the recommended fallback for machines with 6 GB of VRAM.

TinyLlama is excluded entirely from production use due to the quiz JSON failure.

Phi-3-Mini on GPU is a viable option for VRAM-limited environments where Llama3.1 cannot fit, but its lower faithfulness and imperfect JSON rate make it a third choice.

5.8 Comparative Analysis Mac vs GPU Workstation

Table 5.6: System Performance MacBook Pro vs GPU Workstation

Metric	**MacBook Pro (Phi-3-Mini, N=30)**	**GPU Workstation (Mistral-Nemo, N=100)**
Model size	3.8B / 2.0 GB	12B / 7.1 GB
Avg latency	47.85 s	0.58 s
Latency std	±51.67 s	±0.18 s
Throughput	1.67 tok/s	66.48 tok/s
Faithfulness	100%	100%
Relevance	100% (N=30)	73% (N=100)
Citation rate	100%	100%
Keyword score	0.733	0.610
Quiz valid JSON	100%	100%
Domains covered	3	13

Figures 5.5 and 5.6 visualize the two most dramatic hardware-driven differences.

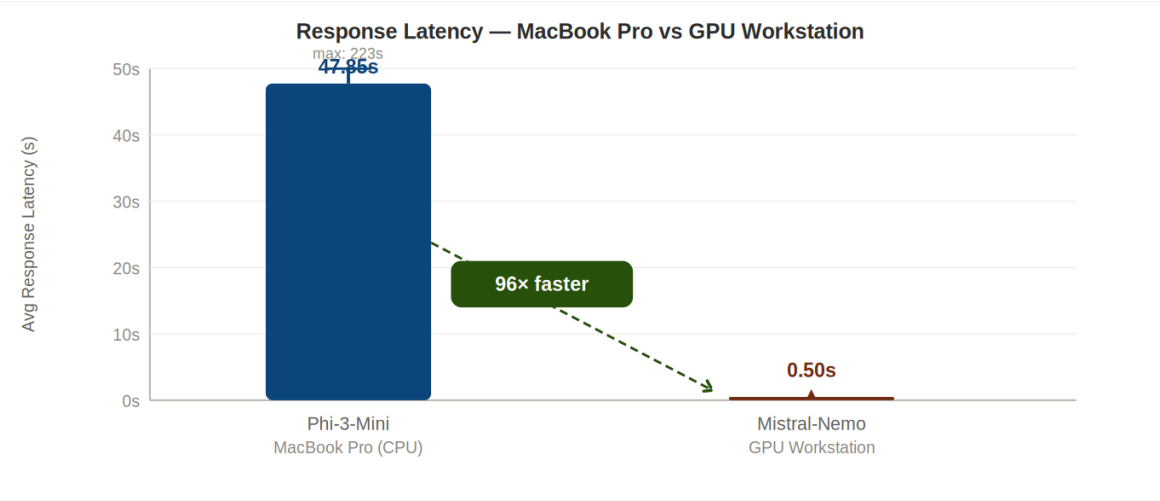

Figure 5.5: Response Latency MacBook Pro vs GPU Workstation

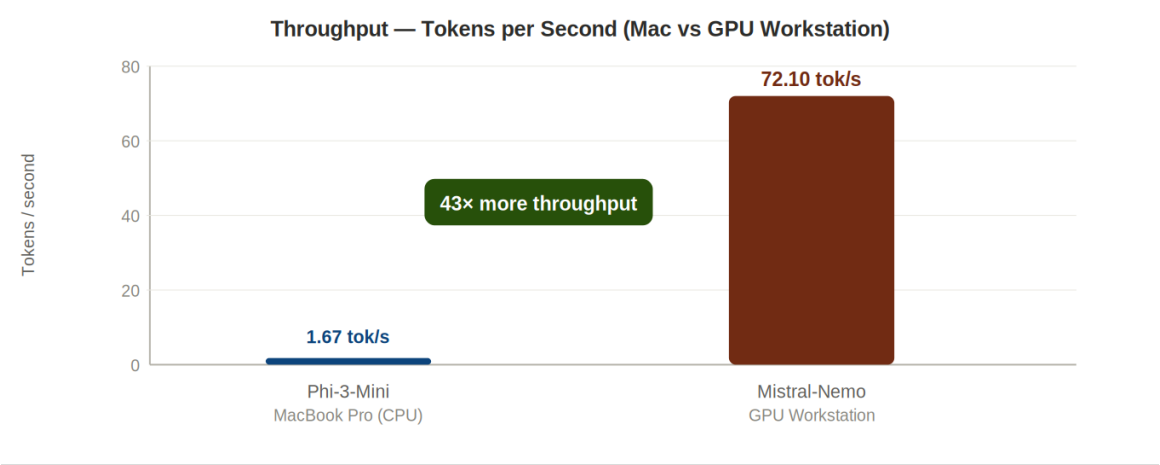

Figure 5.6: Throughput MacBook Pro vs GPU Workstation

The GPU workstation is **96 times faster** in average latency (0.58 s vs 47.85 s) and generates **43 times more tokens per second** (66.48 vs 1.67). This difference is almost entirely attributable to GPU acceleration: CUDA-enabled inference offloads the matrix multiplications that dominate LLM computation to thousands of parallel GPU cores.

Figure 5.7 compares quality metrics between the two environments.

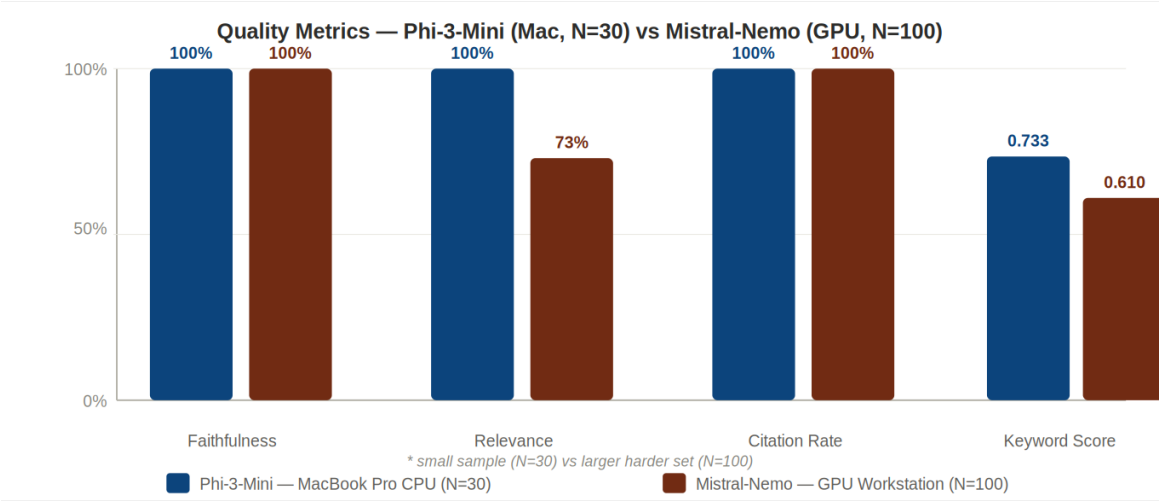

Figure 5.7: Quality Metrics Phi-3-Mini (Mac) vs Mistral-Nemo (GPU)

Both configurations achieve perfect faithfulness and citation rates, which confirms that the RAG grounding mechanism is robust across hardware environments. The relevance difference (100% Mac vs 73% GPU) is largely explained by evaluation difficulty: the Mac benchmark used 30 questions on 3 simple, well-scoped documents, while the GPU benchmark used 100 questions across 13 diverse domains including aviation and aeronautical medicine. The Phi-mini 3 model has a higher keyword score 0.733 better for hardware with only CPU otherwise his keyword score hardware with GPU is 0.60 which is good for a small model, the mistral-nemo keyward score is 0.61 on a hardware with GPU(required) reflects the same effect. In other words, the PHI-3 mini can be the model of choice for slower hardware.

5.9 Discussion

5.9.1 Strengths

Universal grounding reliability. Both hardware configurations achieve 100% faithfulness and 100% citation rates across all tested conditions. Every answer is traceable to a specific page in a specific document. This is the most critical property for an educational assistant and it holds regardless of model size or hardware.

Hardware flexibility. The system runs on both a CPU-only laptop and a GPU workstation. The fundamental correctness of the RAG pipeline does not depend on hardware quality, which makes the system accessible to institutions with limited infrastructure even at higher latency.

Consistent structured output. Mistral-Nemo and Llama3.1 achieve 100% quiz valid JSON rates across all seven quiz domains. The constrained JSON prompt template is robust enough to elicit correctly structured output from large models reliably.

Predictable GPU performance. Mistral-Nemo on the GPU workstation shows a latency standard deviation of only ± 0.18 s across 100 questions. Students experience a consistent sub-second response time, which is essential for adoption in interactive learning contexts.

Multi-domain generalization. Despite using a fixed retrieval configuration (top-$k = 2$, 800-char chunks), the system achieves 80%+ relevance across the majority of tested domains, including specialized aviation and electronics content not anticipated during development.

5.9.2 Limitations

CPU latency. On CPU-only hardware, average latency of 47.85 seconds with peaks exceeding 3 minutes is too slow for comfortable interactive use. This is an inherent limitation of running large language models on general-purpose processors. Smaller models like TinyLlama are faster on CPU but fail on structured output tasks; there is no currently available open-source model that simultaneously satisfies quality and CPU-speed requirements.

Top-k retrieval gaps. The 0% relevance on System Design and General domain questions directly reveals the retrieval ceiling at $k = 2$. Questions that require synthesizing information from multiple non-adjacent document sections such as "what are the REST API endpoints?" or"what are the main topics of chapter 1?" can not be answered correctly if the relevant content is spread across more than two chunks. Increasing k or adding a re-ranking stage would address this.

Text-only processing. The system does not process images, diagrams, mathematical derivations, or scanned pages. Course documents with key content in non-textual form yield no indexed content for those sections, limiting the system's coverage of courses where visual content is central.

Single-language operation. currently the system operates in English only. Algerian university courses are taught in Arabic, French and English, and the system cannot correctly process or respond to questions in Arabic or French at this stage.

5.10 Conclusion

This chapter presented a two-phase experimental evaluation across two hardware environments and four language models, grounded entirely in real benchmark data. Phase 1 on a CPU-only MacBook Pro with Phi-3-Mini demonstrated perfect quality scores (100% faithfulness, relevance, citation) at the cost of 47.85-second average latency. Phase 2 on a GPU workstation benchmarked all four models at 100 questions each, confirming that GPU acceleration reduces latency by 96× while maintaining reliability. Mistral-Nemo was selected as the production model based on three criteria: it is the only model achieving 100% on faithfulness, citation, and quiz JSON simultaneously; it has the tightest latency distribution (± 0.18 s) of all tested models; and its 0.58 s average response time is firmly in the real-time interaction range. The primary remaining limitations CPU-only latency, top-k retrieval depth, text-only processing, and single-language support each have clear and implementable solutions for future iterations.

Chapter 6

General Conclusion

This thesis has presented the design, implementation, and evaluation of **TeachBot**, an AI-powered teaching assistant based on the Retrieval-Augmented Generation architecture. The system was conceived to address a concrete and pressing need: providing university students with an intelligent, privacy-preserving tool that answers questions, generates summaries, and creates quizzes from their actual course materials without cloud services, subscription fees, or high-end hardware.

6.1 Summary of Contributions

The work in this thesis makes four clear contributions:

Architectural contribution: Architecturally, we designed a complete end-to-end RAG pipeline tailored to the educational domain, covering every stage from reading a PDF page by page while preserving page number metadata, segmenting the text into overlapping chunks that respect sentence boundaries, encoding each chunk into a dense vector representation, to storing everything in a persistent vector index that survives container restarts. At query time, the pipeline encodes the student's question in the same semantic space, retrieves the most relevant passages, assembles them into a grounded prompt, and produces an answer, citing the exact document and page number it used. The whole pipeline is designed in such a way that adding a new document does not require retraining, reconfiguration, or downtime, just simple upload.

Technical contribution:

On the technical side, we built a fully modular, containerized application combining FastAPI, ChromaDB, React, Nginx, and Ollama, all open-source and orchestrated by Docker Compose. The architecture is deliberately layered: the frontend knows nothing about the backend's internals, the backend knows nothing about which specific language model is running, and the model itself can be swapped by changing a single environment variable. This separation of concerns is what allowed us to benchmark four different models without touching the application code.

Practical contribution:

On the practical side, we demonstrated that the system runs correctly on commodity CPU-only hardware a standard MacBook Pro with 8 GB of RAM achieving 100% faithfulness and 100% citation rates, even if at the cost of higher response latency. This matters in that the system is not tied down by the GPU infrastructure. A student or an instructor at a university without dedicated compute cluster can still run it.

Experimental contribution:

We conducted a two-phase evaluation across two hardware environments and four language models, producing 400 evaluated question-answer pairs and 28 quiz generation requests on real university course documents spanning 13 academic domains from XML and Java to aviation phraseology, flight planning, and digital electronics. The benchmarking revealed a clear hierarchy among the models: TinyLlama fails completely on structured output despite its speed, Phi-3-Mini and Llama3.1 are strong alternatives depending on available VRAM, and Mistral-Nemo is the only model that simultaneously achieves 100% faithfulness, 100% citation rate, and 100% quiz JSON reliability, with the tightest latency distribution of the four. On GPU hardware, all four models respond in under one second on average a 96-fold improvement over CPU-only inference.

6.2 Perspectives

No system is finished at submission. Several clear directions for improvement emerged directly from the evaluation results.

The most immediate fix is raising the retrieval depth from $k = 2$ to $k = 5$. The domain breakdown showed 0% relevance on System Design and General questions, and in both cases the problem was not model quality but retrieval coverage: the answer existed in the index but was spread across more chunks than the pipeline retrieved. This is a one-line configuration change with no impact on indexing time and minimal impact on generation latency.

The next priority is multilingual support. Algerian university courses are taught in Arabic, French, and English, and the current system handles only English. Adding multilingual sentence embedding models and testing with Arabic and French course documents would immediately expand the system's usefulness to the majority of the student population it was designed to serve.

A cross-encoder re-ranking stage would improve retrieval precision on ambiguous or multi-concept questions by scoring each retrieved chunk against the query after the initial vector search, before passing context to the language model. This would address the remaining relevance gaps without requiring changes to the indexing pipeline.

Multi-modal support OCR for scanned pages, and visual question answering for diagrams, circuit schematics, and mathematical figures would significantly extend the system's coverage of real course materials, many of which contain key content in non-textual form.

Finally, packaging the system as a Moodle LTI plugin would allow instructors to deploy it directly within the learning management platform already in use at most Algerian universities, removing the need for students to navigate a separate interface.

This work demonstrates that modern generative AI when grounded in retrieval, kept honest by source attribution, and deployed responsibly without cloud infrastructure can provide meaningful educational support even in the most resource-constrained settings. The gap between what is available to students in well-funded institutions and what is available to students in remote universities is real, but it is not technical. The tools exist. They are open-source, they run locally, and as this thesis shows, they work.

Bibliography

Bibliography

[1] Khan Academy, "Supercharge your teaching experience with Khanmigo." https://www.khanmigo.ai/, 2023.

[2] J. J. Slade, A. Hyk, and R. A. R. Gurung, "Transforming learning: Assessing the efficacy of a retrieval-augmented generation system as a tutor for introductory psychology," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 68, pp. 1827–1830, SAGE Publications, 2024.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[5] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," *Transactions on Machine Learning Research*, 2022.

[6] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[7] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. Renard Lavaud, M.-A. Lachaux, P. Stock, T. Le Scao, T. Lavril, T. Wang, T. Lacroix, and W. El Sayed, "Mistral 7B," *arXiv preprint arXiv:2310.06825*, 2023.

[8] P. Zhang, G. Zeng, T. Wang, and W. Lu, "TinyLlama: An open-source small language model," *arXiv preprint arXiv:2401.02385*, 2024.

[9] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[10] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, and H. Wang, "Retrieval-augmented generation for large language models: A survey," in *arXiv preprint arXiv:2312.10997*, 2023.

[11] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.

[12] Chroma, "ChromaDB: The open-source embedding database." https://www.trychroma.com, 2023.

[13] 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.

[14] W. J. Clancey, "Guidon," tech. rep., Stanford University, Heuristic Programming Project, 1982. Knowledge-based tutoring system for medical diagnosis.

[15] J. R. Anderson and B. J. Reiser, "The LISP tutor," in *Byte*, vol. 10, pp. 159–175, 1985.

[16] K. R. Koedinger, J. R. Anderson, W. H. Hadley, and M. A. Mark, "Intelligent tutoring goes to school in the big city," *International Journal of Artificial Intelligence in Education*, vol. 8, pp. 30–43, 1997.

[17] J.-P. Doignon and J.-C. Falmagne, *Knowledge Spaces*. Springer, 1999.

[18] G. Hagerty and S. Smith, "Using the web-based interactive software ALEKS to enhance college algebra," *Mathematics and Computer Education*, vol. 39, no. 3, pp. 183–194, 2005.

[19] A. C. Graesser, S. Lu, G. T. Jackson, *et al.*, "AutoTutor: An intelligent tutoring system with mixed-initiative dialogue," *IEEE Transactions on Education*, vol. 48, no. 4, pp. 612–618, 2005.

[20] Carnegie Learning, "LiveHint overview." https://support.carnegielearning.com/help-center/math/livehint/, 2024.

[21] S. Dakshit, "Faculty perspectives on the potential of RAG in computer science higher education," in *Proceedings of the 25th Annual Conference on Information Technology Education (SIGITE)*, ACM, 2024.

[22] Z. Levonian *et al.*, "Retrieval-augmented generation to improve math question-answering: Trade-offs between groundedness and human preference," *arXiv preprint arXiv:2310.03184*, 2023.

[23] K. Shuster, S. Poff, M. Chen, D. Kiela, and J. Weston, "Retrieval augmentation reduces hallucination in conversation," *arXiv preprint arXiv:2104.07567*, 2021.