NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CCDS24-1362

Chatbot for Education

Interim Report

Submitted by: Chan Zhao Yi

Matriculation number: U2022081F

Supervisor: Dr Shen Zhiqi

Acknowledgements

I would like to express my sincere gratitude to my Final Year Project supervisor, Professor Shen Zhiqi, for the steady guidance, constructive feedback, and patience provided throughout this project.

Abstract

Chatbots have become a promising avenue for enriching education and supporting student's learning experiences across disciplines. By leveraging natural language processing and machine learning, they can engage in human-like conversations such as offering personalised assistance, answering questions and guiding learners through course material. Deploying chatbots in educational contexts offers multiple benefits such as increased accessibility to resources, timely feedback and support, and the ability to adapt to diverse learning styles and preferences.

This project aims to develop a chatbot that serves as a learning companion for students. Rather than focusing on a single textbook, it ingests a curated knowledge base of course materials to accurately answer questions across various topics. In its current form it operates as a retrieval-based FAQ system, grounding responses on relevant passages.

The work reported here investigates the design, development, implementation and evaluation of an educational chatbot tailored to enhance student's understanding and learning outcomes. The final report will extend this evaluation and refine the system based on user feedback.

Table of Contents

List of Figures

1. Introduction

1.1. Background

In recent years, chatbots have become widespread across industries, providing 24/7 assistance, quick responses and personalised services while handling repetitive tasks and reducing costs [1]. Within education, chatbots act as virtual assistants supporting instruction, assessments, data retrieval and admissions processes, they deliver continuous accessibility and enhanced engagement while increasing administrative efficiency [1]. Universities use chatbots to answer questions in real time, guide students through complex processes and reduce staff workload, thereby scaling support without sacrificing personal attention [2]. Chatbots also improve user satisfaction by offering 24/7 availability and can save institutions up to 30% in customer support costs [2]. Finally, they provide immediate, personalised support, streamline administrative tasks and help institutions manage large student populations while maintaining service quality [3].

However, building an educational chatbot comes with its own challenges. The system needs to provide accurate information across a broad range of topics while avoiding mistakes or fabricating answers. At the same time, it should respect data privacy regulations and remember the student's interest over longer conversations. Retrieval-based systems, which rely on curated databases, are generally reliable but can feel limited when students ask unexpected or novel questions [3]. On the other hand, fully generative models can produce more natural and engaging dialogue, but the risk of inaccurate or irrelevant responses is higher [4].

To address these limitations, researchers have proposed hybrid solutions. One widely discussed technique is retrieval-augmented generation (RAG), which combines the factual

accuracy of retrieval with the conversational richness of generative models [4]. This project applies the RAG method to build a chatbot that is intended to be both dependable and engaging for learners. In addition, other factors such as design and ethics play important roles. User friendly design is necessary to ensure that students find the system intuitive and trustworthy. Ethical considerations, including fairness, privacy, and academic integrity, must also be considered for the chatbot to be accepted in a learning environment. Measuring usability and user satisfaction is therefore as important as measuring accuracy, since the overall success of such a system depends on whether both students and teachers view it as a valuable and reliable support tool [5].

1.2. Objective

The goal of this project is to design, implement and evaluate an educational chatbot that assists students and educators by providing fast, accurate and personalised support throughout the learning process. To achieve this, the objectives for this project are:

1. **Implement a retrieval-augmented generation chatbot:** Create a baseline retrieval system that maps user queries to relevant internal knowledge in the knowledge base and use a large language model to construct accurate and correct answers.
2. **Develop a robust knowledge base:** Upload and store course materials, lecture notes, frequently asked questions and supplementary resources and embed them into vector representation for efficient similarity search.
3. **Design an engaging user interface:** Build a responsive web application that allows students to converse with the chatbot naturally via text, manage their interaction history, and test their knowledge with quizzes. Support multiple user profiles so that the chatbot can tailor responses to individual courses and levels.

1.3. Scope

To keep the project focused and achievable within the final year timeline, the following scope has been defined:

- **Domain coverage:** The chatbot will support courses that have been uploaded. The knowledge base will consist of curated materials from these courses and can be expanded.
- **Core functionality:** Deliverables include a web application with a conversational chatbot, a retrieval component used to query external data sources for user queries, and an LLM component that generates responses using retrieved information.
- **Other features:** Implement an administrative permission system to allow the site administrator to assign professor roles. Professors will be able to ask the web application to generate quizzes using the large language model so students can test their knowledge. Additionally, the platform will support passwordless login via one-time passcodes (OTP) and restrict OTP issuance to verified NTU student accounts.

By clearly articulating these boundaries, the project sets realistic expectations for what will be delivered in the final report, while leaving room for future enhancements beyond the scope of the final year project.

2. Literature Review

2.1. Evolution of Chatbots

Early chatbots, such as ELIZA, were rule-based systems that used pattern matching and scripted templates to generate responses [5]. As chatbots became more sophisticated, retrieval-based systems emerged, these systems select answers from a knowledge base rather than generate them, and chatbots can now be classified by knowledge scope (open or closed-domain), purpose, and response mechanism [6]. The latest generation of chatbots are generative systems built on transformer-based large language models that produces fluent, context-aware response [2].

2.2. Vector Store (Embeddings, Chunking, Similarity Search)

Vector store maintains dense vector representations of documents so semantically similar passages can be retrieved at query time [3]. In practice, vector store such as Supabase, Chroma, and Weaviate are commonly used, as they provide seamless integration with frontend frameworks such as NextJS and other popular tools. It also provides methods for finding items that resemble a given query based on their similarity score using cosine, inner product, and Euclidean distance to rank candidates. Text Embeddings from models like Google's Gemini map words, sentences, or code to vectors for semantic search, classification, and clustering. Before indexing, documents are split into overlapping chunks to balance recall and precision, selecting an appropriate chunking strategy is critical for optimal results.

Furthermore, semantic search systems use embedding vectors to link database records with queries and measure their closeness. Similarity metrics such as cosine similarity, dot product or Euclidean distance are used to compare query embeddings with document embeddings so that the vector database can return documents whose embeddings are closest to the query [7].

2.3. Retrieval-Augmented Generation (RAG)

Retrieval Augmented Generation (RAG) combines the strengths of information retrieval systems and large language models (LLMs). It enhances the capabilities of LLMs by allowing them to access and incorporate information from external knowledge sources before generating text. This approach improves the accuracy, relevance, and up-to-date nature of the generated text, especially in scenarios where the LLM's pre-trained knowledge is insufficient or outdated.

In enterprise implementations, RAG is treated as a design pattern that augments a chat completion model by adding an information retrieval step. A search index retrieves relevant documents and provides them as context to the large language model, ensuring that generative outputs are grounded in trusted content and constrained to proprietary knowledge sources [8].

Breakdown of RAG:

- **Indexing:** Data from various sources are converted into embeddings and stored in a vector database to enable efficient similarity search [4], [8].

  This usually involves:

  - **Loading of data:** Ingestion of raw data.
  - **Splitting of data:** Breaking down raw data into smaller and manageable chunks.
  - **Embedding of chunks:** Converting each chunk into a numerical vector representation that captures its semantic meaning.
  - **Storing of chunks:** Storing these chunks along with their embedding in a Vector Store

- **Retrieval:** A search algorithm selects relevant documents from the vector store based on a query [4], [8].
- **Augmentation:** The retrieved documents are combined with the query and provided as context to the LLM via prompt engineering [4], [8].
- **Generation:** The LLM generates an answer using both the query and the contextual documents [4], [8].

Types of RAG:

- **Naïve RAG:** The most basic RAG system, involving simple retrieval of text chunks based on semantic similarity to a query, followed by generation using a language model. Easy to implement and understand, good for simple question-answering tasks.
- **Hybrid RAG:** Combines different retrieval methods to overcome the limitations of a single approach.
- **GraphRAG:** Uses knowledge graphs to represent data and their relationships, enabling more nuanced retrieval and understanding.
- **Agentic RAG:** Employs AI agents to orchestrate the RAG process, including query decomposition, retrieval strategy selection, and result refinement.

2.4. Large Language Model (LLM)

Large Language Models are advanced AI systems trained on massive amounts of text data to understand and generate human-like language [3]. They can perform various natural language processing tasks like text classification, question answering, document summarization and text generation [9], [10]. LLMs are based on deep learning architectures, particularly transformer models, which excel at processing and understanding the relationships between words and phrases in text [3].

LLMs have a wide range of applications, including:

- **Text generation:** Creating articles, stories, or other content [9].
- **Information retrieval:** Finding specific information within large text datasets [9].
- **Chatbots and conversational AI:** Building interactive systems that can understand and respond to user queries [9], [10].
- **Sentiment analysis:** Determining the emotional tone of text [9], [10].

Despite their capabilities, LLMs face challenges, including:

- **Computational resources:** Training and running LLMs requires significant computational power and resources [10].
- **Ethical concerns:** LLMs can be misused and may exhibit biases present in their training data [11].
- **Contextual understanding:** While LLMs can generate human-like text, they may still struggle with nuanced understanding of context and complex reasoning [9].

3. System Design & Architecture

This chapter explains the design rationale and architectural choices behind the educational chatbot.

3.1. Overall System Architecture

3.1.1. High-Level Architecture Diagram

The system adopts a layered micro-services architecture comprising a front-end, back-end API, orchestration layer, data layer and large-language models. The front-end is built with Next.js, a React framework that combines static and server-side rendering to improve performance and search-engine optimisation [12]. Styling is handled by TailwindCSS, whose utility-first classes enable rapid prototyping and responsive layouts [13]. The FastAPI back-end exposes asynchronous REST and WebSocket endpoints and automatically generates OpenAPI documentation [14]. Above the API lies an orchestration layer powered by LangChain and PydanticAI. LangChain decomposes the retrieval-augmented generation workflow into modular chains and agents [15], while PydanticAI is a Python agent framework for quickly and confidently building production-grade AI agents and workflows with generative models [16]. The data layer uses Supabase, a PostgreSQL service with pgvector extension, to store relational data, embeddings and object files [17]. External models such as gemini-embedding-001and gpt-4.1 supply the embedding and generative capabilities. Gemini supports more than 100 languages and uses Matryoshka Representation Learning to adjust output dimensions [18], while gpt-4.1 offers enhanced coding, instruction-following and long-context comprehension, including capability to process up to one million tokens in a single input [19], [20].

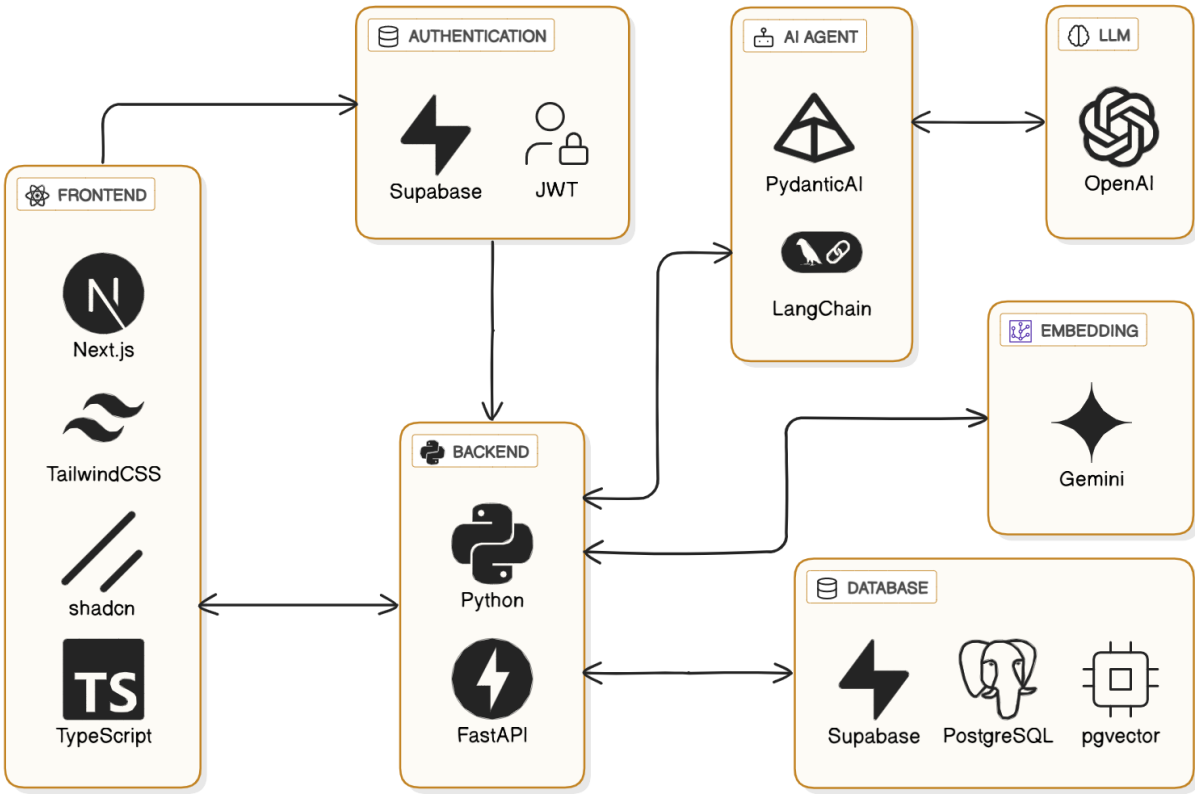

Figure 1: Architecture Diagram of EduChat

3.1.2. Data Flow

When an educator uploads course materials, the system processes the files through four main stages:

1. **Upload:** Educators upload course materials such as PDFs, slides, or text documents through the upload interface. The system checks file type and size, attaches course metadata (e.g., course ID and uploader information), and forwards the files to the backend for ingestion.
2. **Text Extraction:** The backend converts each file into readable text, handling common formats and preserving basic structure such as headings and tables. This ensures the extracted content remains consistent with the source material.

3. **Chunking:** The extracted text is divided into smaller, context-preserving segments (chunks). Each chunk retains metadata referencing its original file and position, allowing the system to retrieve relevant information efficiently during queries.
4. **Embedding:** Each chunk is converted into a vector representation using the embedding model and stored in Supabase with the pgvector extension. This enables fast and accurate similarity searches when students ask questions related to the uploaded content.

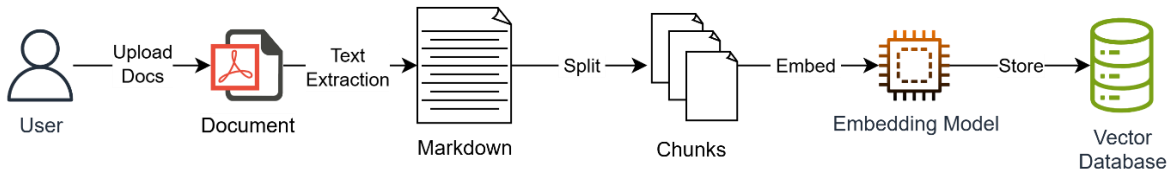

Figure 2: Ingestion Pipeline

When a student submits a message, the system follows a four-stage pipeline:

1. **Client request:** The Next.js client packages the user's query and authentication token and sends it to the FastAPI server via secure REST API. FastAPI performs initial validation using type hints and Pydantic models, ensuring data integrity.
2. **Embedding and retrieval:** The orchestrator generates a dense vector for the query using Gemini embedding. This vector is compared against pre-computed document embeddings stored in Supabase table using cosine similarity. The top-k most similar chunks are returned along with their metadata.
3. **Prompt assembly and generation:** LangChain combines the retrieved data, system instructions and the user's question into a prompt template, applying prompt-engineering techniques to control the output. The final prompt is passed to the AI agent and streams the generated tokens back to the API for real-time response.

4. **Persistence and delivery:** The generated response is validated and saved to the chat_messages table. The API then transmits the response to the client because Supabase provides real-time subscriptions, other components can react to new messages without polling.

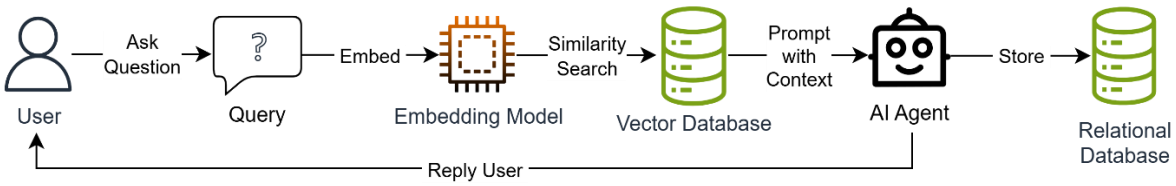

Figure 3: Query Workflow

3.2. Web Application Development

3.2.1. Frontend

The front-end serves as the primary point of interaction for students and instructors. Next.js was selected over alternatives like Angular because it offers server-side rendering, automatic code splitting and a rich plug-in ecosystem, which collectively enhance performance and developer productivity [12]. The interface adopts a light/dark mode using TailwindCSS. Tailwind's utility classes make it straightforward to implement responsive layouts and consistent spacing across pages [13]. Interaction flows are designed to be intuitive: after loading the dashboard, users can start a new chat, review prior conversations or access quizzes. Authentication is handled via one-time passcodes sent to verified university email addresses, eliminating the need for password storage. When a message is submitted, the client establishes a WebSocket connection to receive streaming responses token by token, reducing perceived latency. The UI uses optimistic rendering to display the user's message immediately and then appends the model's reply as it arrives.

3.2.2. Backend

The back-end API is implemented using FastAPI, which provides asynchronous request handling, dependency injection and automatic documentation. Compared with monolithic frameworks like Django or Flask, FastAPI's focus on type-hint-based validation and async support results in higher throughput and scalability [14]. The API exposes endpoints for sending and verifying OTPs, managing user sessions, creating and listing chat sessions, handling file uploads and generating quizzes. A dedicated WebSocket endpoint streams chat responses, ensuring that clients receive each token from gpt-4.1 without waiting for the entire response to finish. The API also schedules background tasks, such as ingesting uploaded course materials into the vector store, to avoid blocking user interactions. Logging and exception handlers capture errors and provide feedback for debugging.

3.2.3. Database & File Storage

Supabase is an open-source alternative to Firebase that uses PostgreSQL as its foundation and supports advanced SQL queries and Atomicity, Consistency, Isolation, and Durability (ACID) transactions [17]. Its authentication service issues JSON Web Tokens (JWTs) and enforces row-level security policies to restrict access at the row level [17]. The database schema includes tables for users, chat sessions, messages, courses, uploaded files, ingested document chunks and quizzes. File uploads are stored in Supabase's object storage and referenced by metadata in the relational tables. When documents are ingested, they are split into overlapping chunks, embedded and inserted into a vectors table with a vector column provided by the pgvector extension. This integration allows similarity search queries to be executed via SQL, enabling filtering by course, user or other metadata. Real-time replication of PostgreSQL rows powers subscription-based updates to the client [17].

The choice of Supabase over a document database like Firestore was deliberate. While Firestore abstracts away schema design, its document model restricts complex joins and cross-collection queries, which are essential for maintaining relationships between courses, users and chat histories. Supabase also avoids vendor lock-in and permits self-hosting [17].

3.2.4. Security Considerations

Securing user data is paramount in an educational context. Authentication is implemented using passwordless one-time passcodes via email. After verifying the code through Supabase Auth, the API issues a JWT that is stored in an HTTP-only, secure cookie. This eliminates the risk of password leaks. Role-based policies are defined in Supabase to ensure that students can only access their own chat logs, while professors can manage their courses and quizzes [17]. All communication between client and server is encrypted via HTTPS, and WebSocket connections are upgraded to secure sockets. FastAPI includes middleware to throttle excessive requests, mitigating denial-of-service attacks, and Pydantic models validate input to prevent malformed or malicious payloads. Finally, prompt templates incorporate guardrails instructing the LLM to avoid sensitive topics and decline to answer questions outside the provided context.

3.3. AI Development

3.3.1. AI Agent

The project utilised a task-oriented AI agent that plans, invokes tools, and returned typed, citation-ready answers grounded in retrieved context. The agent is built using PydanticAI to provide type-check inputs/outputs, structured tool/function calling, automatic retries on validation failures, and run-level instrumentation. It composes with LangChain components for retrieval and prompt templating, ensuring the final response is schema-validated.

3.3.2. Large Language Model (LLM)

OpenAI's gpt-4.1 is adopted as the project's generative engine. It aligns with our goals of accurate responses, predictable operating cost, and smooth integration with the existing stack. In practice, gpt-4.1 offers:

- **Strong reasoning with long prompts:** A large context window comfortably fits the system prompt, retrieved course chunks, and citation scaffolding without truncation, which is essential for RAG-grounded answers.
- **Multimodal:** Supports text, image, and audio.
- **Low cost:** Predictable per-token pricing with cost levers
- **Instructions adherence:** Consistently follows system and developer instructions, resists prompt-injection, and honors tool/function-calling schemas which pairs well with Pydantic for strict, JSON-typed outputs.

3.3.3. Prompt Engineering

Effective prompt design governs how the language model interprets the retrieved context and the user's question. The system uses prompt templates defined in LangChain's documentation. Each template begins with a system message instructing the model to act as a course-aware tutor and to cite the retrieved passages. Retrieved chunks are concatenated with separators and inserted into the prompt, followed by the user's query. Few-shot examples illustrate the expected response format. Guardrails instruct the model to avoid speculation and to admit when information is not available in the provided context.

3.3.4. Vector Database & Retrieval Pipeline

3.3.4.1. Embeddings

Document and query embeddings are computed using gemini-embedding-001. Google reports that the model accepts up to 2048 tokens per input, produces output vectors with configurable dimensions and supports more than 100 languages [18]. It leverages the Matryoshka Representation Learning technique, which allows truncation of the vector to lower dimensions without retraining. Benchmarks like the Massive Text Embedding Benchmark (MTEB) consistently rank Gemini near the top in retrieval tasks. Alternative embedding models such as OpenAI's text-embedding-3-large were considered, but their licensing costs and weaker multilingual coverage made them less suitable.

3.3.4.2. Chunking & Indexing

Prior to embedding, documents are segmented into overlapping chunks to balance semantic coherence and token limits. Chunks of approximately 500–700 words with a 10% overlap ensure that information spanning across boundaries is not lost. During ingestion, each chunk is normalised and stored in the ingested_documents table alongside metadata such as course identifier, file name and page number. An incremental indexing pipeline processes new uploads and updates the vector store. This approach facilitates incremental re-ingestion when a document is updated, avoiding complete re-embedding.

3.3.4.3. Similarity Search

When a query embedding is produced, it is compared against the stored embeddings using cosine similarity. Supabase's pgvector supports efficient approximate nearest-neighbour search and allows queries to filter by metadata such as course or file. A top-k retrieval (typically k = 5) returns the most similar chunks along with their similarity scores. These scores inform the ordering of context in the prompt. In practice, metadata filtering improves precision by narrowing the search space to the relevant course, reducing the risk of retrieving unrelated content.

3.3.4.4. Vector Database

The project stores embeddings directly in the relational database using Supabase pgvector. This choice avoids external dependencies and ensures that embeddings are governed by the same row-level security policies as other data. pgvector supports multiple distance metrics and integrates seamlessly with SQL queries. External vector databases like Pinecone and Weaviate were evaluated but rejected because their managed architectures introduce additional cost and network latency. The integration of embeddings within PostgreSQL also simplifies transaction management and backups. Table 3.1 summarises some alternative technologies considered during the design phase and the rationale for not selecting them.

3.4. Cloud Platform & Hosting Strategy

4. Implementation

This section explain how the system described in Section 3 was built, from the front-end flows and API endpoints to ingestion, retrieval, and deployment. The goal is to surface the concrete implementation decisions and verify that each step aligns with the architecture and data flow previously defined.

4.1. Frontend

The front-end serves as the single-entry point for students and instructors. It handles authentication, chat, course file management, and the professor-side quiz tools.

4.1.1. Authentication & Access Control

- **OTP login (Supabase Auth):** The sign-in page triggers a OTP issuance to verified university emails addresses. Upon success, the Supabase JavaScript client maintains the session JWT.
- **Session propagation:** The JWT is attached to all REST and WebSocket API calls to ensure secure and authenticated communication between client and server.
- **Role-aware UI:** Components render professor tools (file upload, quiz) only when the profile has the appropriate role.

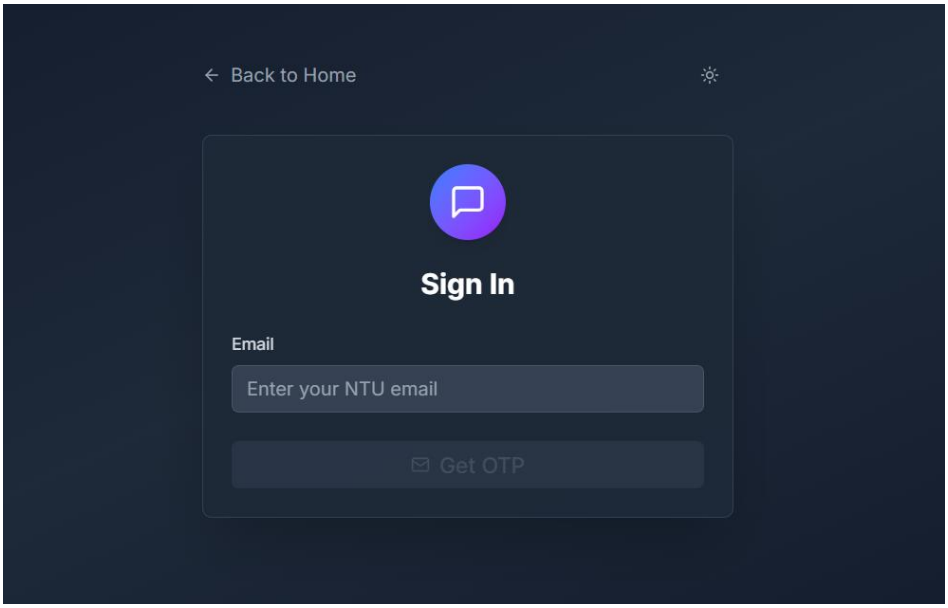

Figure 4: Sign-In Page

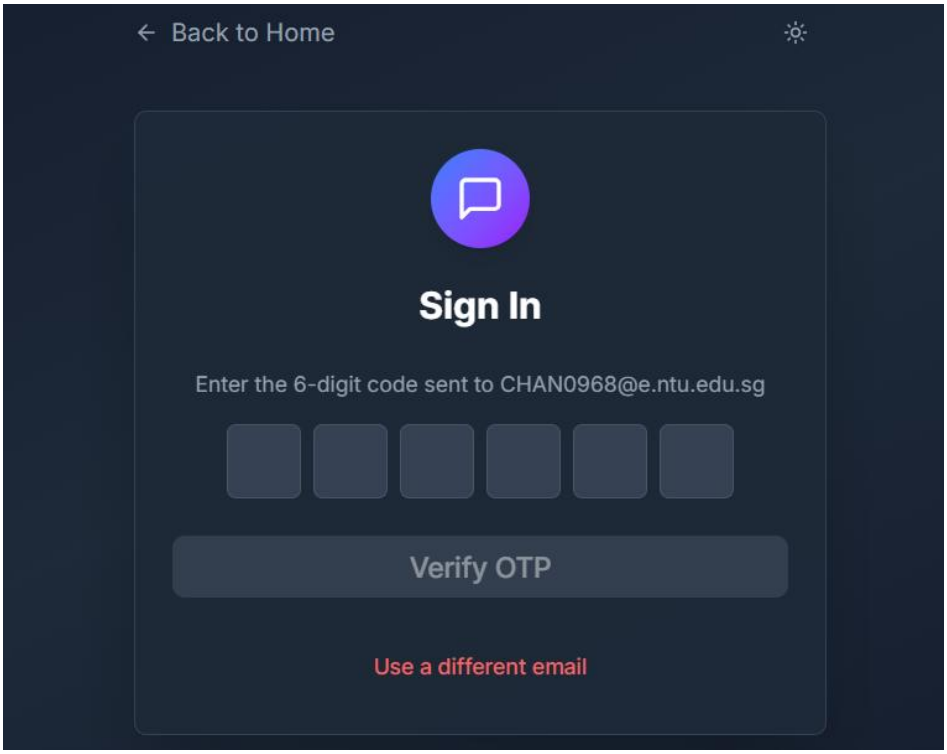

Figure 5: Sign-In Page with OTP Entry

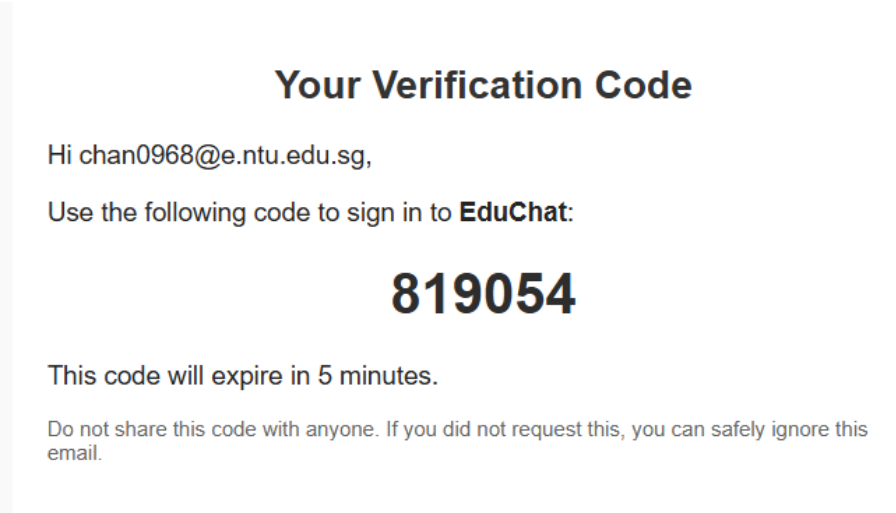

Figure 6: Receiving OTP from EduChat via Supabase Authentication

4.1.2. Chat Experience

- **Streaming UI:** The client establishes a WebSocket (or Server-Sent Events stream) connection to receive tokens incrementally, providing real-time chat feedback as the AI generates responses.
- **Chat list:** Conversation states are stored in Supabase. The sidebar lists recent sessions, allowing users to view or resume prior interactions.
- **Chat history:** Each chat session is timestamped and stored per user profile. Upon reopening a chat, the system retrieves all prior exchanges and context in that conversation, enabling continuity and contextual responses from the chatbot.

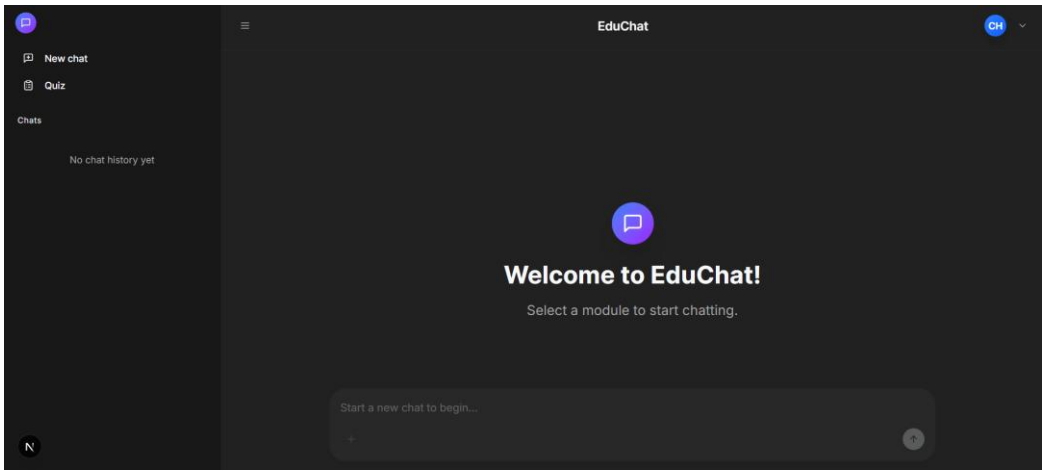

Figure 7: Chat interface after login

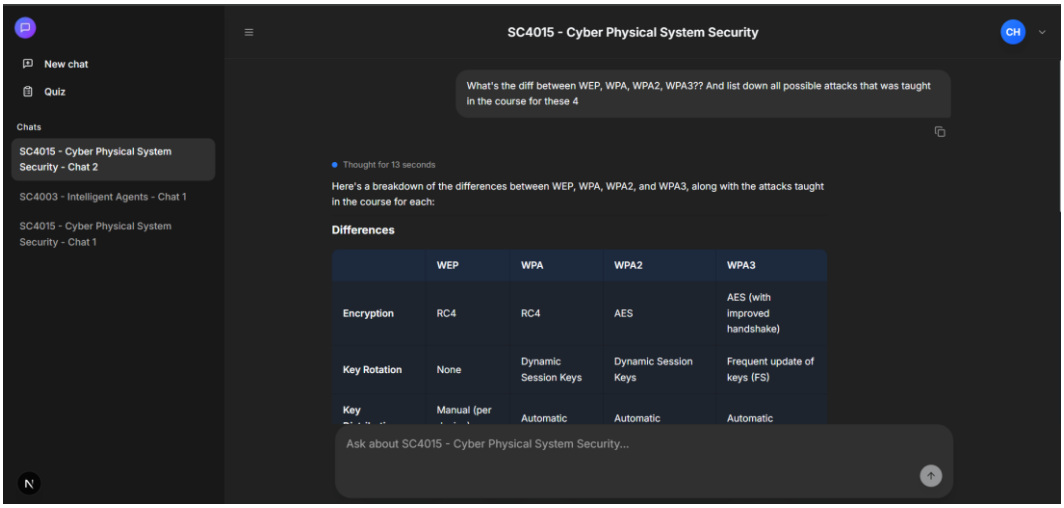

Figure 8: Chat session window

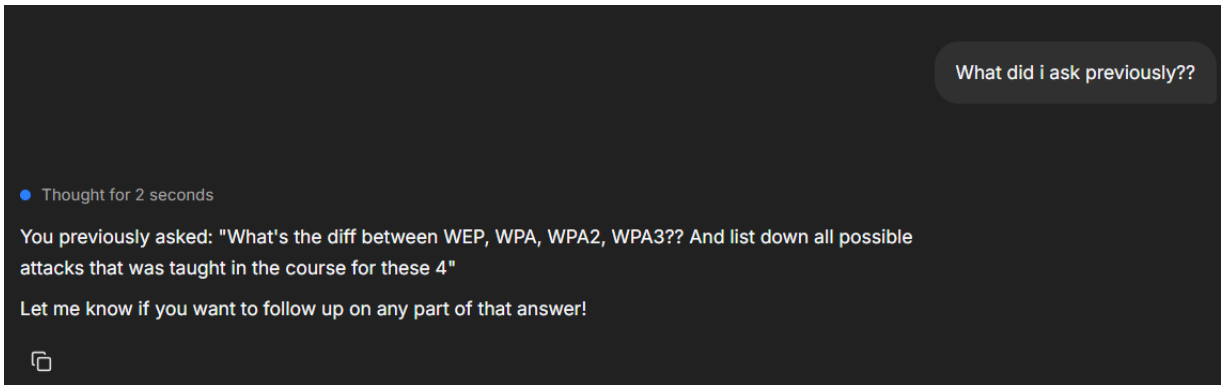

Figure 9: Chatbot retaining context from previous messages

4.1.3. Professor Tools

- **Course dashboard:** Professors can create and manage course modules. Each module serves as a container for uploaded readings, lecture notes, or reference PDFs.
- **File uploads:** The front-end validates file type and size before sending it to the backend ingestion endpoint. Uploaded files are stored with metadata (course ID, uploader ID, timestamp).
- **Quiz generation:** Professors can generate quizzes based on uploaded materials. The front-end sends a structured request to the API; responses are validated using a Pydantic schema before being displayed as formatted quiz items.

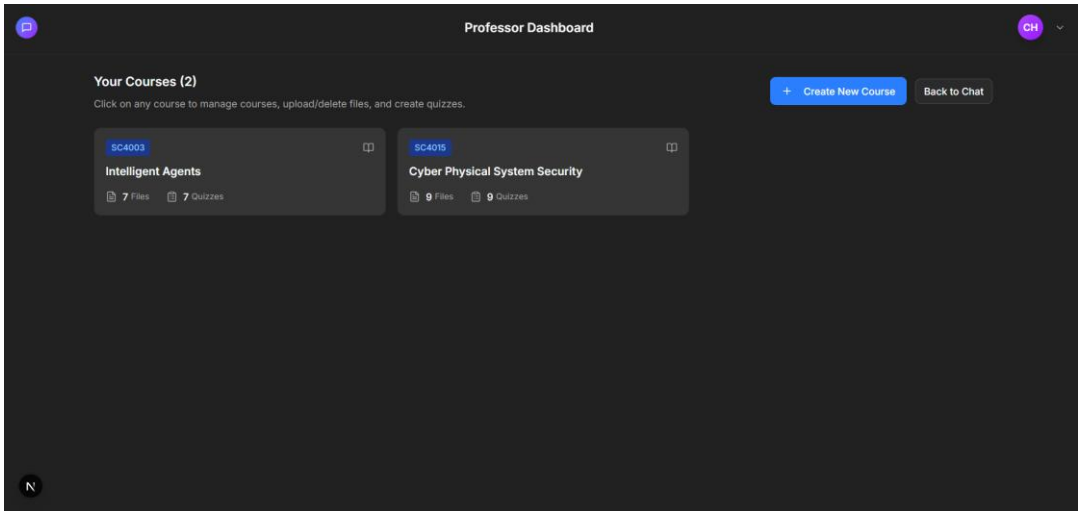

Figure 10: Course Dashboard for Professors to Manage Modules

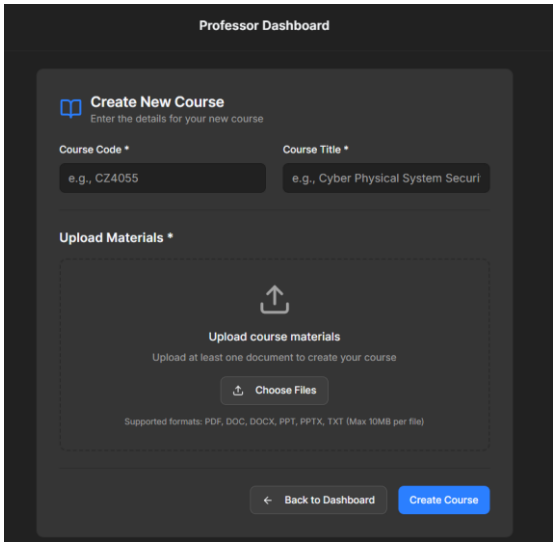

Figure 11: Uploading Files Dashboard for Educators

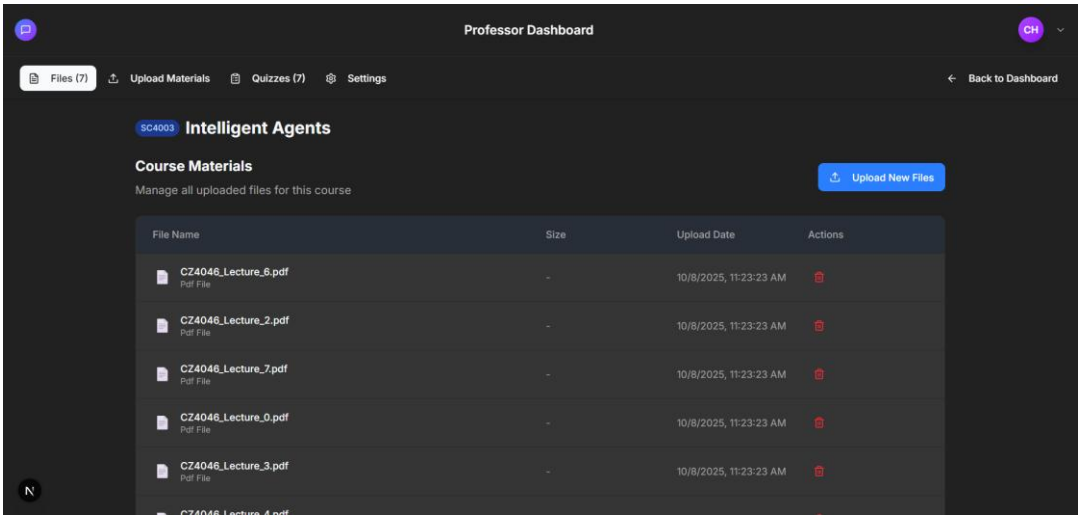

Figure 12: File Management Interface for Uploaded Course Materials

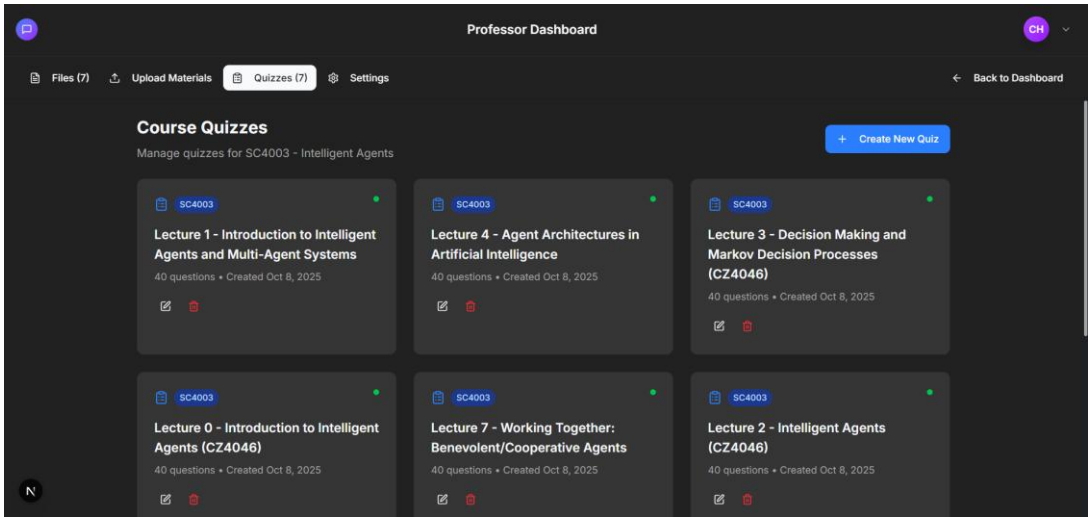

Figure 13: Quiz Management Dashboard

4.1.4. Other features

- **Light & dark mode:** Users can toggle between light and dark themes. The mode preference is stored locally, ensuring the selected theme persists across sessions.
- **Quiz Mode:** Students can attempt quizzes generated by the LLM or manually designed by their educators. The quiz interface provides instant feedback and tracks progress over time.
- **Responsive Design:** The interface is built using TailwindCSS and shadcn components to ensure a seamless experience across desktops, tablets, and mobile devices.

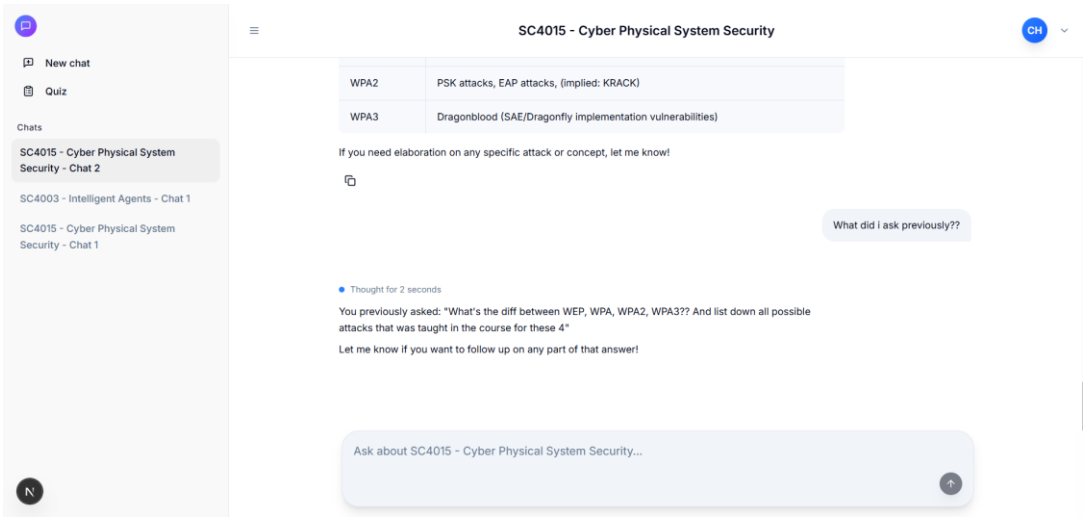

Figure 14: Light Mode Interface

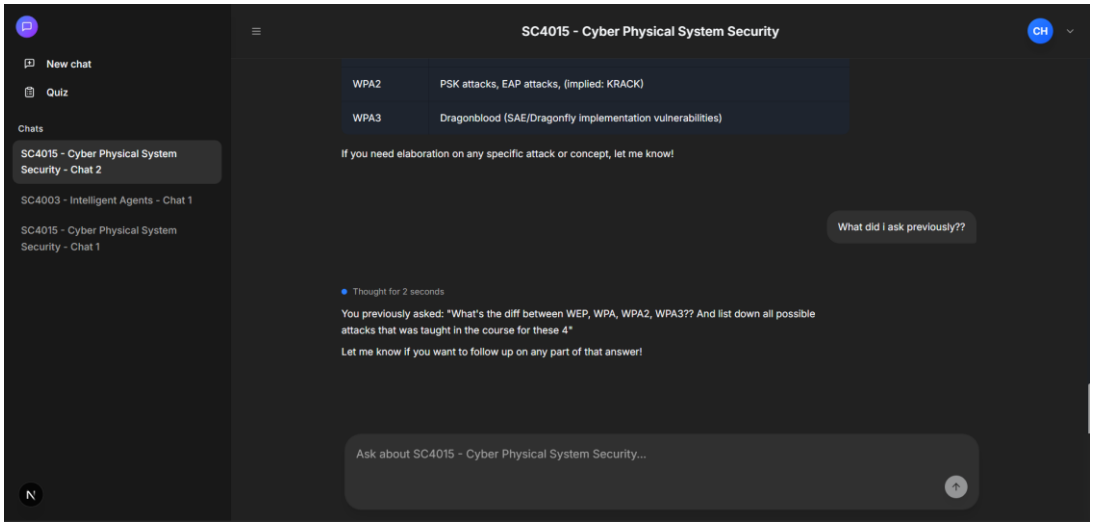

Figure 15: Dark Mode Interface

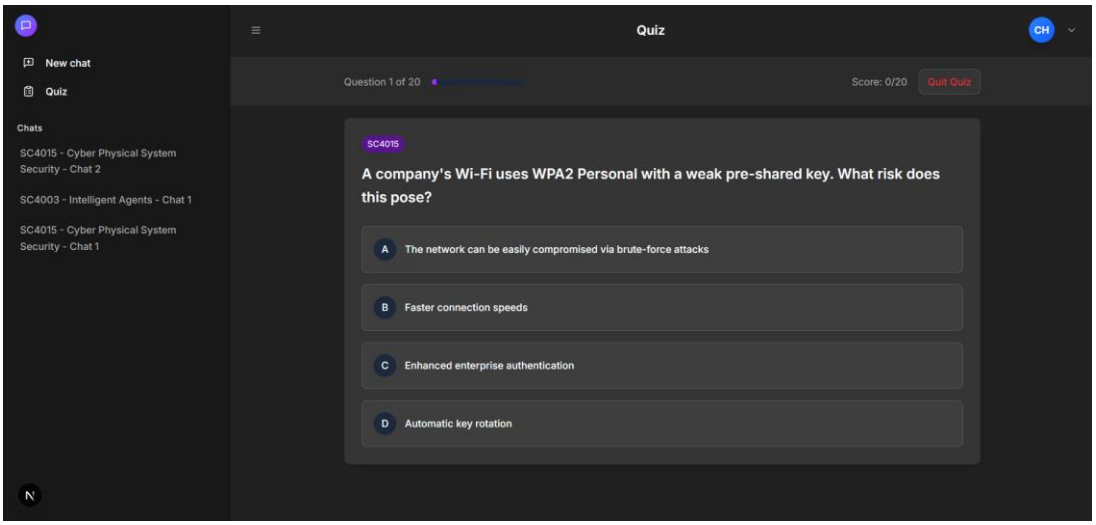

Figure 16: Student Quiz Mode Interface

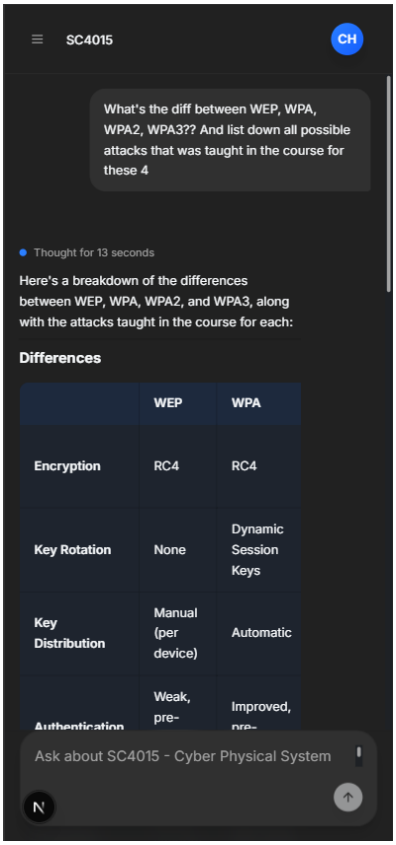

Figure 17: Responsive Layout - Mobile View

4.2. Backend

The backend is implemented using FastAPI to connect the frontend and the database. It exposes two primary workflows, ingestion and query, which together enable the system to store, retrieve, and process course content efficiently.

4.2.1. Endpoints

All API endpoints require a valid Supabase JWT for authentication. Request and response bodies are defined and validated using Pydantic models, ensuring consistent structure and type safety across the system. For endpoints that support real-time responses (such as chat and query), communication is handled through Server-Sent Events (SSE) or WebSocket streams, allowing tokens to be delivered progressively to the client interface.

Health Check & Root

- **GET /:** Health check endpoint to verify backend service is running.

User Management

- **POST /user-management:** Handle user authentication and creation. Creates new users with Student role on first login or returns existing user data.

Chat Management

- **POST /chat/send:** Send a chat message and receive AI-generated response. Persists conversation history and associate messages with user sessions and courses.
- **GET /chat/sessions**: Retrieve all chat sessions for a specific user.
- **GET /chat/sessions/{session_id}/messages**: Retrieve all messages from a specific chat session.
- **DELETE /chat/sessions/{session_id}**: Delete a chat session and all its associated messages.

Course Management

- **POST /courses/create:** Create a new course with files. Processes files synchronously to ensure completion, generates vector embeddings for RAG and automatically generates quizzes.
- **POST /courses/{course_id}/upload:** Upload additional files to an existing course. Processes files, updates vector database, and generates quizzes for new content.
- **POST /courses/{course_id}/delete:** Delete a course and all associated data (files, quizzes, embeddings).
- **POST /courses/{course_id}/files/{course_file_id}/delete:** Delete a specific file from a course.

Quiz Management

- **POST /courses/{course_id}/generate-quiz:** Generate quizzes for existing course.
- **GET /courses/{course_id}/quizzes:** Retrieve all quiz topics and questions for a course.
- **POST /courses/{course_id}/quizzes:** Create a new empty quiz.
- **DELETE /courses/{course_id}/quizzes/{topic_id}:** Delete a quiz and all its questions.
- **GET /courses/{course_id}/quizzes/{topic_id}/details:** Get detailed information about a specific quiz topic including all questions.
- **POST /courses/{course_id}/quizzes/{topic_id}/questions:** Create a new question in a quiz.
- **PUT /courses/{course_id}/quizzes/{topic_id}/questions/{question_id}:** Update an existing quiz question.
- **DELETE /courses/{course_id}/quizzes/{topic_id}/questions/{question_id}:** Delete a specific quiz question.

4.2.2. Models, Validation & Security

- **Pydantic models:** All input and output data are validated using Pydantic schemas, ensuring strict type enforcement and predictable API responses.
- **JWT verification:** Each request carries a Supabase-issued JWT. A FastAPI dependency validates the token, extracts the user identity, and attaches the user context to the request scope.
- **Rate limits & CORS:** Middleware enforces rate limits per user to prevent abuse. CORS settings restrict access exclusively to the front-end domain for security.

4.3. Ingestion Pipeline

The ingestion pipeline converts raw course materials into structured, searchable vector representations. This process enables fast, contextually relevant retrieval during chat interactions.

4.3.1. Parsing & Normalization

- **Conversion to text:** Each uploaded file (PDF/Word/Slides/TXT) is converted into Markdown. Structural elements such as headings and tables are preserved when possible.
- **Cleaning:** The parser removes repetitive headers and footers, fixes line breaks, and trims whitespace to maintain readability.
- **Output:** The result is kept in a single “clean text” string per file.

4.3.2. Chunking Strategy

- **Why chunking:** Large documents exceed model context limits; splitting them improves retrieval precision and latency.
- **How to split**: The text is segmented into overlapping windows of approximately 650-700 tokens with a 100-token overlap, preserving continuity across chunk boundaries.
- **What to save:** Each chunk includes its text, file reference, course linkage & metadata.

4.3.3. Embeddings

- **Model:** gemini-embedding-001
- **Process:** Each chunk is embedded into a high-dimensional space. During querying, similarity searches identify the most contextually relevant chunks.
- **Reliability:** Failed embedding requests are retried briefly. Any persistent errors are logged for administrative review.

4.3.4. Supabase

- **Available Tables:** users, courses, course_files, ingested_documents, chat_sessions, chat_messages, quiz_topics, and quiz_questions.
- **Structure:** Every record uses a UUID as its primary key for global uniqueness.
- **Performance:** Supabase's vector indexing ensures fast top-k retrieval, while selective filtering optimizes both accuracy and prompt efficiency.

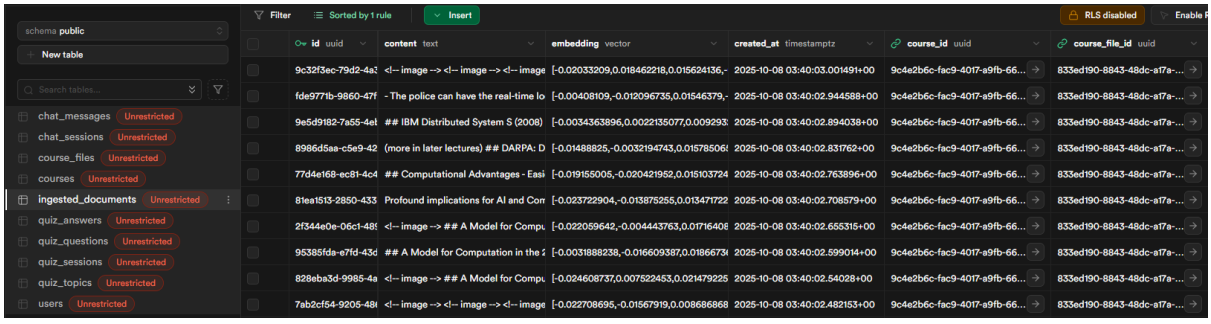

Figure 18: Supabase Table Editor

4.4. Retrieval & Orchestration

At query time, the backend executes a Retrieval-Augmented Generation process that embeds the user's question, retrieves the most relevant content from Supabase, and orchestrates a structured response from the AI model. This ensures the chatbot produces accurate, course-aware answers grounded in uploaded materials.

4.4.1. Retrieval

- **Query embedding:** When a user submits a query, the query is embedded, converting it into a high-dimensional vector representation, using the same Gemini embedding model to ensure vector space consistency.
- **Similarity search:** The query vector is compared against existing document embeddings stored in Supabase's pgvector index using cosine similarity. The most semantically relevant document chunks are then ranked and returned to the application.
- **Top-k value:** The system retrieves the top 5 most relevant chunks (k = 5) to balance precision and prompt context length.

4.4.2. Prompt Construction

- **Template:** The retrieved chunks are combined with the original user query within a course-aware tutor prompt template. The model is instructed to respond as an academic assistant, referencing materials from the retrieved context only.
- **Safety/guardrails:** System-level instructions enforce boundaries, directing the model to politely decline questions outside the retrieved context or unrelated to the uploaded course materials.

4.4.3. Agent & Output Validation

- **Model call:** The PydanticAI agent orchestrates the model call to GPT-4.1, leveraging function-calling capabilities for structured tasks such as quiz generation or contextual explanations.
- **Schema enforcement:** Responses from the model are validated against predefined Pydantic schemas before being stored. This ensures that all AI outputs conform to expected data structures, maintaining type safety and reliability across the application.

4.5. Deployment

5. Conclusion & Future Work

5.1. Conclusion

This interim report presented the problem context, and the system built to date: a course-aware educational assistant that retrieves from instructor-provided materials and answers student queries with citations.

The current implementation delivers an end-to-end workflow:

1. A Next.js front-end with OTP authentication
2. A FastAPI back-end exposing ingestion and chat endpoints
3. A retrieval-augmented generation pipeline orchestrated with LangChain and PydanticAI
4. Supabase for relational data, object storage, and pgvector similarity search. OpenAI's gpt-4o is used generate replies that balances reasoning quality, context capacity,

multimodal readiness, and operational fit with the existing toolchain, while Gemini Embedding is used for multilingual, strong retrieval embeddings.

Key engineering decisions, such as chunking strategy, prompt structure, top-k retrieval, and schema-validated outputs, are implemented to reduce hallucination, preserve grounding, and streamline downstream features such as quizzes and instructor dashboards. The current system currently consists of the final frontend design and supports security measures.

Overall, the platform is functional and aligned with the design in section 3. The remaining work focuses on rigorous evaluation, hardening, and polish to reach production quality and to substantiate learning outcomes with quantitative and qualitative evidence.

5.2. Future Improvements

~~5.2.1. Ingestion~~

~~As I am still in the process of developing the web application, new innovative ideas keep constantly coming in. Therefore, a new and improve ingestion function from the backend which includes AI quiz generation and ingestion of documents will be implemented.~~

~~5.2.2. Chatbot conversation and history~~

~~Similarly, I've set up a new table in my database and will be working on the backend functions to store and the chat history and retrieval method to give a more accurate and efficient response to the user.~~

5.2.3. Permission

This will be done last but setting up a simple admin page to set permission of user between Student and Educator. Therefore, user with Educator role can create new course and upload files.

5.2.4. Deployment

Running locally is not viable for multi-user access and reliability. The next milestone is a managed deployment. I will be considering a cloud provider that supports GPU with low cost.

References

[1] Juwel rana, “21 key benefits of chatbots in education you shouldn’t miss!,” Apr. 2024, [Online]. Available: https://www.revechat.com/blog/benefits-of-chatbots-in-education/

[2] Antonio Nucci, “What Are Large Language Models (LLMs)?” [Online]. Available: https://aisera.com/blog/large-language-models-llms/

[3] Langflow, “Vector Stores.” [Online]. Available: https://docs.langflow.org/components-vector-stores

[4] Hugging Face, “RAG.” [Online]. Available: https://huggingface.co/docs/transformers/en/model_doc/rag

[5] Wikipedia, “ELIZA.” [Online]. Available: https://en.wikipedia.org/wiki/ELIZA

[6] E. Adamopoulou and L. Moussiades, “An Overview of Chatbot Technology,” in *Artificial Intelligence Applications and Innovations*, vol. 584, I. Maglogiannis, L. Iliadis, and E. Pimenidis, Eds., in IFIP Advances in Information and Communication Technology, vol. 584. , Cham: Springer International Publishing, 2020, pp. 373–383. doi: 10.1007/978-3-030-49186-4_31.

[7] Supabase Docs, “Semantic Search.” [Online]. Available: https://supabase.com/docs/guides/ai/semantic-search

[8] Microsoft Learn, “Retrieval Augmented Generation (RAG) in Azure AI Search.” [Online]. Available: https://learn.microsoft.com/en-us/azure/search/retrieval-augmented-generation-overview?tabs=docs

[9] Elastic, “What are large language models (LLMs)?” [Online]. Available: https://www.elastic.co/what-is/large-language-models

[10] SAP, “What is a large language model?” [Online]. Available: https://www.sap.com/resources/what-is-large-language-model

[11] Katalin Wargo and Brier Anderson, “Striking a Balance: Navigating the Ethical Dilemmas of AI in Higher Education,” Dec. 2024, [Online]. Available: https://er.educause.edu/articles/2024/12/striking-a-balance-navigating-the-ethical-dilemmas-of-ai-in-higher-education

[12] TatvaSoft and Itesh Sharma, “Next.js vs Angular: Choose the Right Framework.” [Online]. Available: https://www.tatvasoft.com/outsourcing/2024/04/next-js-vs-angular.html
[13] Strapi and Oluwadamilola Oshungboye, “Bootstrap vs. Tailwind CSS: A Comparison of Top CSS Frameworks.” [Online]. Available: https://strapi.io/blog/bootstrap-vs-tailwind-css-a-comparison-of-top-css-frameworks
[14] JetBrains and Evgenia Verbina, “Which Is the Best Python Web Framework: Django, Flask, or FastAPI?” [Online]. Available: https://blog.jetbrains.com/pycharm/2025/02/django-flask-fastapi/
[15] DatascienceDojo, “What is LangChain? Key Features, Tools, and Use Cases.” [Online]. Available: https://datasciencedojo.com/blog/what-is-langchain/
[16] Samuel Colvin, “Introduction to Pydantic AI.” [Online]. Available: https://ai.pydantic.dev/
[17] Stanley Ulili, “Supabase vs Firebase.” [Online]. Available: https://betterstack.com/community/guides/scaling-nodejs/supabase-vs-firebase/
[18] Min Choi and Janie Zhang, “Gemini Embedding now generally available in the Gemini API.” [Online]. Available: https://developers.googleblog.com/en/gemini-embedding-available-gemini-api/
[19] OpenAI, “Introducing GPT-4.1 in the API.” [Online]. Available: https://openai.com/index/gpt-4-1/
[20] Reuters, “OpenAI launches new GPT-4.1 models with improved coding, long context comprehension,” OpenAI launches new GPT-4.1 models with improved coding, long context comprehension. [Online]. Available: https://www.reuters.com/technology/artificial-intelligence/openai-launches-new-gpt-41-models-with-improved-coding-long-context-2025-04-14/