StudyBuddies Course Helper

Project Description, Use Case Diagram and Requirements

Group 11
Emran Habib
Ishtiaq Chowdhury
Khushi Nanda
Ben Thompson

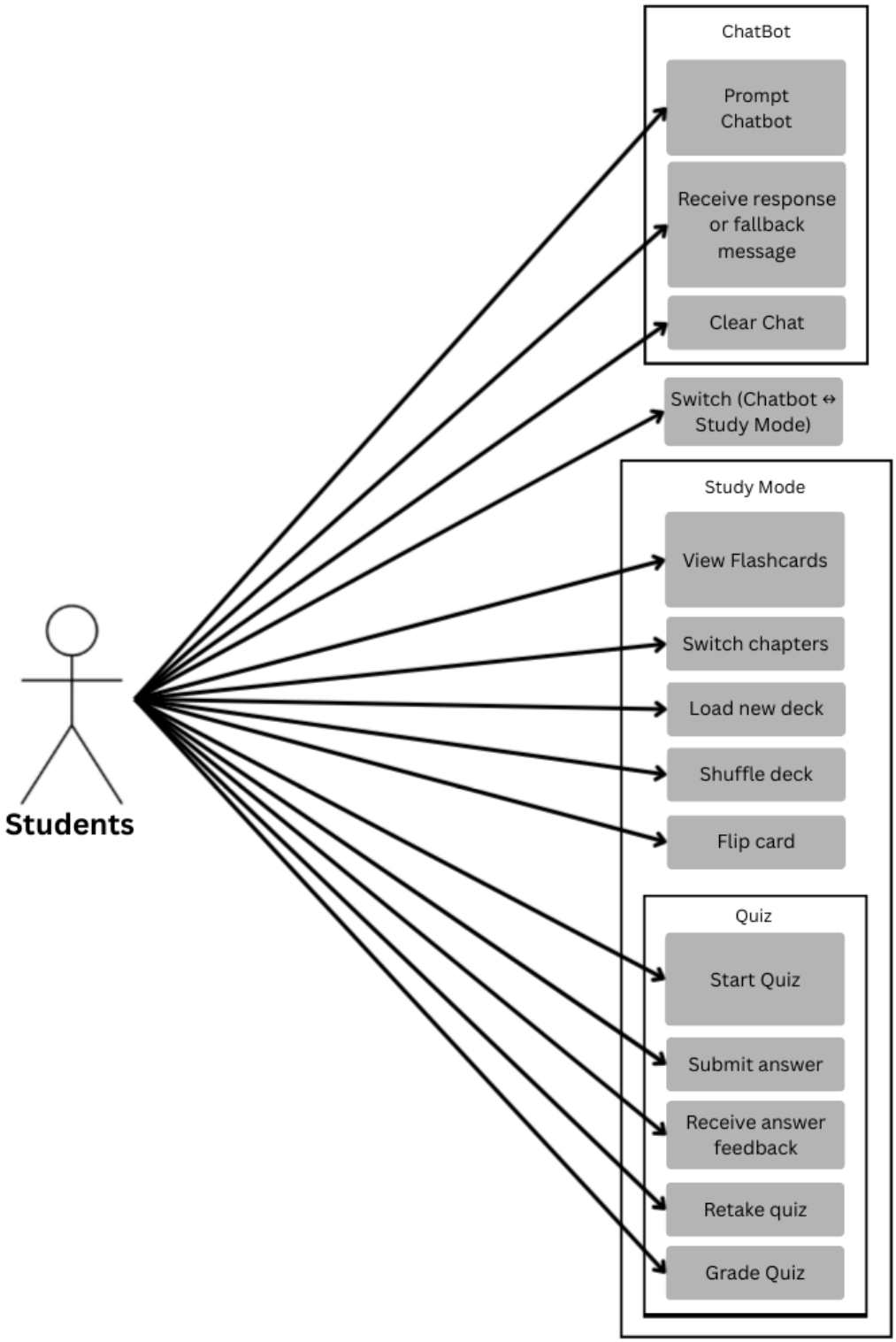
1. Main Components of the System

Component Type	Our Choice	Rationale
Front-End Framework	Streamlit	We chose Streamlit to build the user interface because it is simple to use and works directly with Python. This saves us from having to learn a separate front-end framework like React. With React, we would need to learn JavaScript, JSX, and Node.js, and also set up things like npm packages, routing, and state management, which would add more complexity to the project. Streamlit avoids all of that and lets us quickly build an interactive web app in just Python, where students can access the chatbot and quiz features in their browser.
Programming Language	Python	Python is the main programming language for our project since it has strong support for data processing, database connections, and AI libraries. It also makes it easier to connect different parts of the system, like embeddings, quiz features, and the chatbot, without switching between multiple languages. The majority of our group is already familiar with Python, which makes development faster and smoother, and it ties directly into the rest of our tech stack since tools like Streamlit, PostgreSQL connectors, and the OpenAI API all work well with Python.
Embedding Model API	OpenAI's *text-embedding-3-small* model	We are using OpenAI's embedding model to convert course material (syllabus, PowerPoints, lecture notes) into numerical vectors. These embeddings make it possible to search for answers based on meaning rather than just exact keywords. OpenAI's new embedding model is affordable (around $0.02 per 1M tokens), easy to integrate with vector databases, and all the computation is handled by OpenAI's servers, which means no powerful hardware is required on our end. This makes it ideal for retrieval-augmented generation (RAG) tasks like quiz question generation. We chose the smaller model because it is faster and cheaper to run, which works well for a class project.

Component Type	Our Choice	Rationale
Generation ModelAPI	OpenAI's *gpt-5-nano* text generative model	We are using OpenAI's gpt-5-nano model to generate text because it gives us a fast and cost-efficient way to produce quiz questions, flashcards, and chatbot responses. It fits well with our RAG setup since we only need lightweight generation on top of the embeddings and retrieval, and *gpt-5-nano* keeps costs low while still delivering good enough quality for our project.
Database	PostgreSQL (with Pgvector, a vector embedding extension)	We are using PostgreSQL to store structured data such as the syllabus and lecture slides. The pgvector extension allows us to also store vector embeddings and perform similarity searches. This means we can handle both normal database queries and semantic search in one system, instead of using separate databases.
Database API	psycopg2	We chose to use `psycopg2` to connect Python to our PostgreSQL database with the pgvector extension. This library is lightweight, widely used, and makes it simple to run SQL queries directly from Python. It allows us to insert, update, and query both structured data (like quiz results and flashcards) and vector data (embeddings) in one place. Since our project only needs a straightforward connection and query execution, `psycopg2` is a good fit compared to a heavier option like SQLAlchemy, which adds an ORM layer we don't need for this scope.
Containerization Tool	Docker	We are using Docker to package our application so it runs the same way on any computer. This avoids issues with dependencies and setup, which is especially useful when showing the project to others. It also demonstrates good software engineering practices by making the system portable and reproducible.

Component Type	Our Choice	Rationale
Storage	Local system storage	We are using the local storage of each user's device to handle session-specific data such as chat history and environment variables required to run the application. This design choice means that the current chat history is stored only on the student's own machine, rather than on our server. By avoiding centralized storage of user-specific data, we eliminate the need to implement account systems or authorization layers, which simplifies development and reduces privacy concerns. This approach ensures that students have control over their own data, while keeping the system lightweight and easier to maintain.

2. Use Case Diagram



3. Functional and Non-Functional Requirements

3.1. Functional Requirements

- The system shall answer student questions about the syllabus, deadlines, course policies, and lecture topics.
- The system shall retrieve answers from a course knowledge base stored in a database.
- The system shall handle unknown questions by returning a polite fallback with suggested queries.
- The system shall provide a chat interface for natural-language questions and responses.
- The system shall support a flashcard mode that lets students flip cards and navigate forward/back.
- The system shall allow shuffling and filtering flashcards by tag/topic/unit.
- The system shall support a quiz mode that accepts free-text answers and evaluates them semantically.
- The system shall display immediate feedback in quiz mode, including why an answer was incorrect.
- The system shall maintain session score/progress for the current quiz attempt.
- The system shall store chat history locally in the student's browser and allow the student to clear it.
- The system shall ingest course materials (e.g., syllabus, assignment details) into the knowledge base.
- The system shall prevent generation of reminders or push notifications.
- The system shall avoid displaying external citations or source snippets in responses.
- The system shall operate with a single user role (student) and not expose instructor-only data.
- The system shall not collect or store personally identifiable information or grades.
- The system shall not generate answers unrelated to the course material.
- The system shall not provide full solutions to graded assignments or exams.
- The system shall not store or transmit sensitive student data such as grades, IDs, or personal information.
- The system shall not expose database schema or backend errors directly to the user.
- The system shall not accept inputs larger than a defined limit (e.g., 1,000 characters per query).
- The system shall not allow users to bypass quiz grading by submitting blank or invalid answers.

3.2. Non-Functional Requirements

- The system must respond to chat queries within **30 seconds** on average.
- The quiz grader must evaluate answers within **10 seconds** on average.
- The frontend must be usable on a modern desktop and be **responsive**.
- The system must store current user chats **locally** and provide a **Clear Chat** control.
- The backend must be **containerized** (Docker) and deployable via a single command.
- The database should be **PostgreSQL with pgvector.**
- The retrieval pipeline must constrain the LLM to answer **only from retrieved context** and return a fallback when insufficient.
- Secrets must be managed via environment variables and platform secret stores; no secrets may be committed.
- Logs/analytics must exclude PII and capture basic metrics (request counts, failures, average latency).
- Third-party libraries must use **permissive licenses** suitable for academic use.
- The system must degrade gracefully when the LLM or embedding service is unavailable through showing a friendly error and retry option.

4. Contributions

4.1. Main Components

The main components of our product were determined **collectively** through **group** discussions. Emran and Khushi collaborated to write up the table of component types, our choices, and rationale. Emran, Khushi, and Ben contributed to rearranging the table and finalizing the phrasing of rationale.

4.2. Use Case Diagram

We drafted our use cases and roles in a joint meeting, then we **all** worked together to begin designing the use case UML diagram in Canva. Ishtiaq and Ben completed the diagram by grouping use cases, making format adjustments, and exporting the final design.

4.3. Functional and Non-Functional Requirements

All team members contributed to group discussions surrounding the functional and non-functional requirements. Ishtiaq initially recorded the requirements gathered from the discussions. The **group** then **collaboratively** reviewed the lists of requirements, removing inapplicable ones, adding new ones, and introducing constraints as needed.

4.4 Contributions

Khushi tracked contributions and summarized them in sections. **All** group members reviewed the contributions sections to confirm accuracy.