📝🚀 Coursepilot

Requirements

0. Introduction:

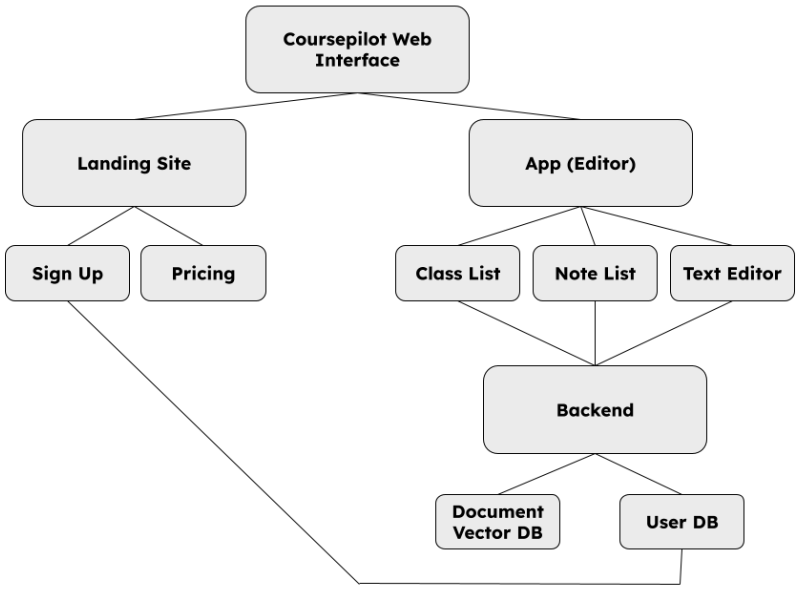
With innovative tools and frameworks emerging surrounding LLMs we wanted to apply some new techniques such as RAG (Retrieval Augmented Generation) to provide value to students by making learning more effective and engaging. Our idea is to embed the semantic meaning of documents, which could be student-taken notes or slides uploaded by a professor, store the documents in a vector database, and use them to provide greater context to generate a better output when the LLM is used. Coursepilot's backend will be the RAG engine, that will parse documents, embed their semantic meaning, and store each document in a MongoDB Atlas database with an id, the text, and its vector embedding. This vector embedding allows us to retrieve relevant documents when querying later. We will also be making our Gemini API calls from the backend and connecting it to the frontend with Flask. The frontend will be mainly built with Next.js, and will handle user authentication with Clerk.

1. Hardware

Coursepilot is a web application, so the application is supported by any modern browser able to render HTML powered by Javascript, with a preference for wide screens.

2. Frontend Requirements

2.0 Diagram



2.1 Notebook

- 2.1.1 Classlist - a list view for organization displaying all classes added by the user
    - Emoji picker - an emoji picker to select icons for different classes
    - Title - a customizable title input for class name
    - Backend - selection updates note in storage
- 2.1.2 Notelist - a list view for organization displaying all notes within the selected class
    - Emoji picker - an emoji picker to select icons for different notes
    - Title - a customizable title input for the note
    - Backend - selection updates note in storage
- 2.1.3 Text Editor - a rich text editor for the selected note
    - The editor and menu bar can be created using the TipTap library
    - Menubar - a menu of buttons to control text options
        - Connection to the text area and its selections
        - Bold - button to bold text
        - Italics - button to italicize text
        - Lists - button to create a numbered/bulleted list

        - Size - button to increase/decrease font size
        - Color - a picker for font color
    - Editor - the main text area
        - Connection to the menu bar and its selections
        - Formatting - markdown support for headings, lists, code blocks
        - AI - a debounced autocomplete
            - Preview - a gray text preview of autocompleted text that can be filled using tab
            - API - request made to Gemini using the context of the user's notes
        - Backend - modification updates note in storage
- 2.1.4 Quizzes - a list view for generated flashcards from notes
    - Flashcard - a front/back Q&A
        - Front - a question to be answered by the back of the card
        - Back - the answer to the question on the front of the card

2.2 Authentication

- 2.2.1 Supports sign up with email, Google, and optionally Microsoft
- 2.2.2 Clerk - Library that provides authentication handling and component library
    - Sign up page - a page/component form for making sign in requests for new users
    - Sign in page - a page/component form for making account creation requests for existing users
    - Sign in button - a component for sign in/sign up page redirecting
    - Sign out button - a component for user sign out handling on the frontend
    - UserID - token used for database organization to store classes and notes per user
    - Access gating - rendering of the app can be conditional based on if a user is authenticated

3. Backend Requirements

3.1 RAG Engine

- 3.1.1 Document chunker - currently implemented as a sentence based chunker but can be extended to perform semantic chunking
- 3.1.2 Document embedder - utilizing the BERT (Bidirectional Encoder Representation for Transformers) language model to capture semantic meaning of text chunks in vector representations

- 3.1.2 MongoDB Atlas Vector database - allows for storage of text data along with its vector representation, which allows for effective querying and retrieval using the KNN algorithm

3.2 Data Pipeline

- 3.2.1 Flask server hosting an API will create a data pipeline from the RAG engine to the frontend
    - “/autocomplete”
        - Takes the context of the current note and returns the suggested autocomplete
    - “/getContext”
        - (optional) Takes the current note and provides metadata for tooltips for related documents previously uploaded

4. Performance Requirements

- 4.1 Autocomplete
    - 4.1.1 - Autocomplete suggestion must be returned in ~500ms and under 1s (1000ms)
        - Ghost text suggestion should be visible to the user less than 1s after the user stops typing
- 4.2 Flashcards
    - 4.2.1 - Flashcard set generation must be completed in under 10s
- 4.3 Quizzes
    - 4.3.1 - Quiz generation must be completed in under 10s
    - 4.3.2 - Quiz validation must be completed in under 1s (1000ms)
        - On quiz completion, users should be able to view their validated results in less than 1 second with short answer responses

5. Environment Requirements

Resource	**Dev**	**Execution**
Windows PC	X	X
AMD Ryzen 5 5600 3.8GHz 6 core	X	
32gb DDR5 RAM, Nvidia 3070 8gb VRAM	X	

Macbook Pro	X	X
Apple M1 Max	X	
32gb RAM	X	

Resource	**Dev**	**Execution**
Next.js	X	X
Flask	X	X
MongoDB	X	X
Heroku		X
Visual Studio Code	X	
Cursor	X	
Clerk Auth	X	X
Google Gemini LLM	X	X
TipTap	X	X