Project Plan: A Modular and Scalable AI Mentor for Computer Science Education

I. Strategic Foundation & Environment Configuration (Week 1: PLAN)

This initial phase establishes the foundational infrastructure and development environment for the project. A meticulously configured and version-controlled environment is paramount to mitigating risks associated with configuration drift, ensuring reproducibility, and enabling a streamlined development workflow across the seven-week timeline. All server-side components will be deployed on the remote Runpod instance, with the local laptop functioning as a thin client for development.

A. Project Scaffolding and Version Control

A unified monorepo structure will be employed to manage both frontend and backend codebases. This approach simplifies dependency management, streamlines the build process, and provides a single source of truth for the entire project under Git version control. The first step is to create the root project directory and initialize it as a Git repository. Subdirectories for the backend and frontend applications will be created immediately to establish the high-level structure.
Executable Instruction:

Shell

```
mkdir ai-mentor-project && cd ai-mentor-project && git init
```

Executable Instruction:

Shell

```
mkdir backend frontend
```

A comprehensive .gitignore file will be established to prevent unnecessary build artifacts, dependencies, and environment-specific files from being committed to the repository.
AI Coder Prompt:
Generate a comprehensive.gitignore file for a monorepo project containing a Python/FastAPI backend and a Node.js/Svelte frontend. Include standard ignores for Python virtual environments (pycache, venv/), Node.js dependencies (node_modules), editor-specific files (.vscode,.idea), operating system files (.DS_Store), and environment configuration files (.env).

B. Remote GPU Environment Setup (Runpod)

The project's hardware constraints—an 8GB RAM local laptop and a 24GB GPU on Runpod—mandate a decoupled architecture. All computationally intensive tasks, including language model inference, embedding generation, and database operations, will be executed on the remote Runpod instance. The local machine will serve exclusively as a development client, connecting via SSH for code editing and terminal access. This separation is critical for the project's feasibility.
A Runpod instance equipped with a 24GB GPU (e.g., an RTX A5000) should be provisioned. A standard Docker image with pre-installed CUDA drivers, Python, and essential build tools (e.g., runpod/pytorch) is recommended to simplify the setup process.
Once the pod is running, the next step is to install the necessary system-level packages for the project's technology stack.
Executable Instruction (On Runpod):

Shell

```
sudo apt-get update && sudo apt-get install -y python3-venv python3-pip nodejs npm git
build-essential cmake
```

Secure Shell (SSH) access should be configured to allow for remote development using tools like Visual Studio Code's Remote-SSH extension, providing a seamless local development experience while leveraging the power of the remote server.

C. Backend Environment Setup (FastAPI)

The backend, built with FastAPI, will serve as the central nervous system of the application, orchestrating interactions between the user interface, the AI agent, and the data stores.
Within the backend directory on the Runpod instance, a Python virtual environment will be created to isolate project dependencies.
Executable Instruction (On Runpod):

Shell

```
cd backend && python3 -m venv venv && source venv/bin/activate
```

All required Python libraries will be installed into this virtual environment. The selection of packages is deliberate: fastapi[all] provides the core framework along with optional dependencies for a complete development experience [1]; llama-cpp-python[server] is crucial as it includes not only the Python bindings for the LLM engine but also the command-line utility to launch an OpenAI-compatible web server [2]; pymilvus provides the client for the vector database; llama-index and langgraph form the core of the RAG and agentic logic; and PyMuPDF is a high-performance PDF parsing library.[3]

Executable Instruction (On Runpod):

Shell

```
pip install "fastapi[all]" "uvicorn[standard]" "python-dotenv" "pymilvus" "llama-index"
"langgraph" "llama-cpp-python[server]" "PyMuPDF"
```

The initial FastAPI application structure will be scaffolded following best practices for modularity and scalability, separating concerns into distinct directories for API endpoints, services, and core configuration.[4]

AI Coder Prompt:

Generate the initial file structure for a FastAPI project within the 'backend' directory. Create a 'main.py' file to serve as the application entry point. Also, create an 'app' subdirectory containing 'init.py' and three further subdirectories: 'api' for endpoint routers, 'core' for configuration, and 'services' for business logic. In 'main.py', instantiate the FastAPI app and include a root health-check endpoint ('/') that returns {"status": "ok"}.

D. Frontend Environment Setup (Svelte)

The frontend will be developed locally on the 8GB RAM laptop. Svelte is selected for its unique compile-time approach, which shifts the bulk of the framework's work from the browser's runtime to the build step.[5] This results in highly optimized, minimal vanilla JavaScript, leading to smaller application bundles and superior performance.[7] For an educational tool, a fluid and responsive user interface is critical for student engagement.

A new SvelteKit project will be scaffolded within the frontend directory.

Executable Instruction (On local machine):

Shell

```
cd frontend && npm create svelte@latest.
```

During the setup prompts, select a "Skeleton project" with TypeScript support for type safety and improved developer experience. Following the scaffolding, install the necessary Node.js dependencies.

Executable Instruction (On local machine):

Shell

```
npm install
```

E. Database and Inference Engine Setup (Docker)

To ensure consistent and reproducible deployment of stateful services, Docker and Docker Compose will be used to manage the Milvus vector database on the Runpod instance. Milvus is an open-source vector database designed for scalable storage and high-performance similarity searches on embedding vectors, making it an ideal choice for a production-oriented RAG application.[8]

A docker-compose.yml file will be created in the project's root directory on the Runpod instance. This file will define the services for Milvus Standalone and its required dependencies (etcd for metadata storage and MinIO for object storage).

AI Coder Prompt:

Generate a docker-compose.yml file for Milvus Standalone version 2.3.x. The configuration should include three services: 'etcd', 'minio', and 'milvus-standalone'. Ensure that the necessary ports for Milvus (19530 for gRPC, 9091 for HTTP) are exposed. Configure the services to use Docker volumes for data persistence, mapping them to local directories (e.g., './volumes/etcd', './volumes/minio', './volumes/milvus').

With the configuration file in place, the database services can be launched with a single command.

Executable Instruction (On Runpod):

Shell

```
docker-compose up -d
```

This command will pull the required images and start the containers in the background, providing a stable database environment for the duration of the project.

II. System Architecture & Component Specification (Weeks 2-3: SPECS)

This phase translates the high-level project objectives into a detailed technical blueprint. Each component of the system is specified with precision, defining interfaces, data flows, and core logic. This Specs phase is crucial for de-risking the subsequent Build phase by ensuring all architectural decisions are made deliberately and cohesively.

A. The AI Core: Agentic RAG Engine Specification

The heart of the AI mentor is an advanced Retrieval-Augmented Generation (RAG) system built with an agentic workflow. This design moves beyond simple question-answering to incorporate a reasoning loop, enabling the system to self-correct and handle more complex queries.

1. Local LLM Inference Server (llama.cpp)

The system will leverage llama.cpp to run a quantized language model locally on the Runpod GPU. This provides data privacy, eliminates API costs, and offers full control over the inference process.[10]

- **Model Selection:** A high-quality, instruction-tuned model in the GGUF (GPT-Generated Unified Format) is required. A 7-billion parameter model, such as Mistral-7B-Instruct-v0.2 or a comparable Llama-3 variant, offers an excellent balance of performance and resource consumption. A 5-bit quantization (Q5_K_M) provides a good trade-off between model quality and file size. The 24GB VRAM of the Runpod GPU is sufficient to offload all layers of such a model for maximum performance.
  Executable Instruction (On Runpod):
  Shell
  ```
  mkdir -p models && wget https://huggingface.co/TheBloke/Mistral-7B-Instruct-v0.2-GGUF/resolve/main/mistral-7b-instruct-v0.2.q5_k_m.gguf -P./models/
  ```

- **Server Configuration:** The llama.cpp engine will be run as a standalone server process, exposing an OpenAI-compatible API.[2] This architectural choice is fundamental to the system's modularity. It decouples the inference engine from the main application logic, allowing the LLM to be updated, restarted, or even replaced with an entirely different inference server (e.g., Ollama, vLLM) with minimal changes to the backend code—often just a change in a single environment variable specifying the API base URL.

Executable Instruction (Save as start_llm.sh on Runpod):

Bash

```
#!/bin/bash
# Activate the virtual environment if necessary
# source backend/venv/bin/activate

python3 -m llama_cpp.server \
  --model./models/mistral-7b-instruct-v0.2.q5_k_m.gguf \
  --n_gpu_layers -1 \
  --n_ctx 4096 \
  --host 0.0.0.0 \
  --port 8080 \
  --chat_format mistral-instruct
```

This script configures the server to offload all layers to the GPU (--n_gpu_layers -1), sets a context window of 4096 tokens (--n_ctx 4096), and listens for requests on all network interfaces on port 8080.

2. Data Ingestion and Indexing Pipeline (LlamaIndex & Milvus)

The knowledge base of the AI mentor will be built by ingesting a corpus of computer science educational materials (e.g., textbooks, lecture notes, articles) in PDF format. LlamaIndex will orchestrate this entire pipeline.[12]

- **Document Parsing:** The pipeline will utilize PyMuPDF for its high-performance text extraction capabilities, ensuring that content is accurately and efficiently read from the source PDF files.[13]
- **Chunking Strategy:** Documents will be split into smaller, semantically coherent chunks using LlamaIndex's SentenceSplitter. A chunk size of 512 tokens with an overlap of 50 tokens is a robust starting point. This overlap helps preserve context across chunk boundaries, improving the quality of retrieval for queries that span multiple concepts.[15]
- **Embedding Generation:** Text chunks will be converted into numerical vector embeddings. The system will use an OpenAIEmbedding client from LlamaIndex, configured to point to the local llama.cpp server's embedding endpoint. This keeps the entire AI pipeline self-hosted.
- **Vector Storage:** The generated embeddings and their corresponding text chunks will be stored in the Milvus vector database, which was set up via Docker Compose. LlamaIndex's MilvusVectorStore integration will manage the connection and data insertion.[16]

  AI Coder Prompt:

Generate a Python script 'ingest.py' that uses the LlamaIndex library to create a data ingestion pipeline. The script must:

1. Accept a directory path containing PDF files as a command-line argument.
2. Use 'SimpleDirectoryReader' to load all documents, explicitly configuring it to use PyMuPDF for file extraction.
3. Initialize an 'OpenAIEmbedding' model client, setting the 'base_url' to 'http://localhost:8080/v1' and 'api_key' to a placeholder string like "not-needed".
4. Initialize a 'MilvusVectorStore' client to connect to a Milvus instance running on 'localhost:19530'.
5. Define a 'ServiceContext' and a 'StorageContext' to link the embedding model and vector store.
6. Use a 'SentenceSplitter' with a chunk size of 512 and an overlap of 50.
7. Create a 'VectorStoreIndex' from the loaded documents, passing the contexts and node parser, to process and store the data in Milvus.
8. Include logging to print progress messages during the ingestion process.

3. Agentic Workflow (LangGraph)

The selection of LangGraph is the central architectural improvement over a standard RAG pipeline. It facilitates the creation of a stateful, cyclical graph, enabling the AI to perform complex reasoning and self-correction.[17] This structure transforms the system from a simple information retriever into a more robust agent.

- **State Definition:** The agent's memory and workflow status will be managed in a TypedDict state object. This object will persist across the nodes of the graph, tracking the conversation history, the original question, retrieved documents, and the generated response.[19]

  AI Coder Prompt:

  In a new Python file 'backend/app/services/agent_graph.py', define a LangGraph state class named 'AgentState' using 'typing.TypedDict'. The state must include the following keys: 'question' (str), 'documents' (List[str]), 'generation' (str), and 'messages' (Annotated[list, add_messages]) to maintain the conversational history.
- **Nodes and Edges:** The workflow will be defined as a graph of nodes (functions) and edges (conditional transitions).
  1. **retrieve Node:** This node is the entry point for a user query. It uses a LlamaIndex query engine, configured as a tool, to search the Milvus vector store for documents relevant to the question in the current state.[21] The retrieved documents are added to the state.
  2. **grade_documents Node:** After retrieval, this node is called. It makes an LLM call to evaluate the relevance of the retrieved documents to the original question. This is a critical self-reflection step that prevents irrelevant information from polluting the final answer.[23] The node's output determines the next transition.

3. **rewrite_query Node:** If the grade_documents node determines the retrieved context is poor, the workflow transitions here. This node uses the LLM to rephrase the original question, aiming to improve retrieval results. The state is updated with the new query, and the graph loops back to the retrieve node.
4. **generate Node:** If the documents are graded as relevant, the workflow proceeds to this node. It synthesizes the final answer by sending the original question and the validated documents to the LLM in a carefully crafted prompt.
5. **Conditional Edges:** The logic connecting these nodes is crucial. The edge from grade_documents will branch: if relevance is high, it proceeds to generate; if relevance is low, it proceeds to rewrite_query. This loop (retrieve -> grade_documents -> rewrite_query -> retrieve) is the core of the agentic behavior.

This cyclical, self-correcting architecture is a significant enhancement. A standard RAG pipeline is linear; if the initial retrieval is poor, the final output will be poor. This agentic design introduces resilience and a rudimentary form of problem-solving, allowing the system to "try again" when it fails, which is essential for a tool that aims to mentor rather than just answer.

B. The Backend API: FastAPI Service Layer Specification

The FastAPI application will expose a set of well-defined endpoints for the Svelte frontend to consume. Pydantic models will be used for rigorous data validation and serialization, ensuring a robust API contract.[24]

- **CORS Configuration:** Cross-Origin Resource Sharing (CORS) middleware must be configured to permit requests from the Svelte development server (typically running on http://localhost:5173) to the backend server (running on http://localhost:8000).[25]
- **API Endpoints:** The API contract defines all possible interactions between the frontend and backend. Establishing this contract early allows for parallel development. The frontend team can build UI components against this specification using mock data, while the backend team implements the corresponding logic.

Table 1: FastAPI Endpoint Specification				
Endpoint	**Method**	**Description**	**Request Body (Pydantic Model)**	**Response Body (Pydantic Model)**
/api/chat	POST	Sends a user message to the agent for a complete, non-streamed response.	ChatMessage(message: str, conversation_id: str)	AgentResponse(response: str, sources: List[str])
/ws/chat/{conversation_id}	WEBSOCKET	Establishes a real-time,	str (user message)	str (streamed tokens)

		bidirectional chat session for streaming responses.		
/api/documents	POST	Uploads a new PDF document to a staging area for future ingestion.	fastapi.UploadFile	Status(status: str, message: str)
/api/ingest	POST	Triggers the ingestion process for all documents in the staging area.	None	Status(status: str, message: str)

- **WebSocket Implementation:** For a highly interactive and responsive user experience, a WebSocket endpoint will be implemented. This allows the backend to stream response tokens from the generate node of the LangGraph agent directly to the client as they are produced by the LLM. This creates the "live typing" effect common in modern AI assistants and is far superior to traditional HTTP polling.[27]

C. The Frontend Interface: Svelte User Experience Specification

The Svelte frontend will provide a clean, intuitive, and reactive interface for interacting with the AI mentor.

- **Component Architecture:** The UI will be broken down into a series of modular, reusable components:
  - ChatView.svelte: The primary component that orchestrates the entire chat interface.
  - MessageList.svelte: Renders the conversation history.
  - Message.svelte: Represents a single chat bubble, styled differently for the user and the AI assistant.
  - ChatInput.svelte: Contains the text input field and the send button.
  - SourceViewer.svelte: A component that can be toggled to display the source documents the AI used to generate its response.
- **State Management:** Svelte's native writable stores will be used for managing the application's state, such as the array of chat messages, the current input value, and the WebSocket connection status. This approach is simple, powerful, and avoids the need for external state management libraries.[6]
- **API Communication:** A dedicated TypeScript module (src/lib/api.ts) will encapsulate all communication with the FastAPI backend. It will provide functions for making fetch requests to the REST endpoints and a class-based service to manage the lifecycle of the WebSocket connection.

- **Real-time Updates:** The WebSocket service will listen for incoming token streams from the server. As tokens arrive, they will be appended to the last message in the messages store. Svelte's reactivity will automatically and efficiently update the DOM, rendering the AI's response in real-time without requiring any manual DOM manipulation.[7]

III. End-to-End MVP Assembly (Weeks 4-5: BUILD)

This two-week phase is dedicated to the implementation of the specifications defined previously. The workflow will consist of a series of machine-executable commands and precise AI coder prompts designed to rapidly construct a functional, end-to-end Minimum Viable Product (MVP).

A. Build Backend: AI Core and API

The backend build focuses on bringing the AI logic and the server endpoints to life.

- 1. Execute Data Ingestion:
  The first step is to populate the Milvus vector database with the prepared educational materials. This is accomplished by running the ingest.py script created during the specification phase. A directory named course_materials should be created in the project root and populated with relevant PDF files.
  Executable Instruction (On Runpod):
  Shell
  ```
  python ingest.py --directory./course_materials/
  ```

- 2. Implement LangGraph Agent:
  The core agentic logic is now implemented using LangGraph. The AI coder prompt below instructs the generation of the complete graph, including its nodes and conditional edges.
  AI Coder Prompt:
  Based on the 'AgentState' defined in 'backend/app/services/agent_graph.py', implement the full LangGraph workflow.
  1. Create a 'retrieve' function. This function should initialize a LlamaIndex 'VectorStoreIndex' from the existing Milvus store and create a query engine. It will then query this engine using the 'question' from the state and update the state's 'documents' key with the retrieved node content.
  2. Create a 'grade_documents' function. This function should take the 'question' and 'documents' from the state, construct a prompt asking the LLM to determine if the documents are relevant for answering the question (respond with 'yes' or 'no'), and return the decision.
  3. Create a 'rewrite_query' function. This function should take the 'question' from

the state, construct a prompt asking the LLM to rephrase it for better retrieval results, and update the state's 'question' key with the new query.

4. Create a 'generate' function. This function should take the 'question' and 'documents', format them into a final prompt for the LLM, and update the state's 'generation' key with the LLM's response.
5. Instantiate a 'StateGraph' with the 'AgentState'.
6. Add the four functions as nodes to the graph: 'retriever', 'grade_documents', 'rewrite_query', and 'generate'.
7. Set 'retriever' as the entry point.
8. Add a conditional edge from 'grade_documents' that routes to 'generate' if the grade is 'yes', and to 'rewrite_query' if the grade is 'no'.
9. Add a standard edge from 'rewrite_query' back to 'retriever'.
10. Add a standard edge from 'generate' to the special 'END' node.
11. Compile the graph and expose it via a single runnable interface.

- 3. Implement FastAPI Endpoints:
  With the agent logic compiled, the FastAPI endpoints specified in the API contract are now implemented to expose this functionality to the frontend.
  AI Coder Prompt:
  In a new file 'backend/app/api/chat_router.py', create a FastAPI 'APIRouter'.
  1. Import the compiled LangGraph agent from 'agent_graph.py'.
  2. Implement the '/chat' POST endpoint. It should accept a Pydantic model 'ChatMessage' (containing 'message' and 'conversation_id' fields), invoke the agent with the message, and return a Pydantic model 'AgentResponse' (containing 'response' and 'sources').
  3. Implement the '/ws/chat/{conversation_id}' WebSocket endpoint. Upon connection, it should enter a loop to 'await websocket.receive_text()'. For each received message, it should invoke the agent using its 'stream' method and iterate through the output chunks. Each chunk (token) from the 'generate' node should be sent back to the client via 'await websocket.send_text(token)'. Handle 'WebSocketDisconnect' exceptions gracefully.
  4. In 'main.py', include this new router in the main FastAPI app instance.

B. Build Frontend: Svelte UI

The focus now shifts to the local machine to construct the user-facing application.

- 1. Create Svelte Components:
  The modular UI components are created first.
  AI Coder Prompt:
  Generate the Svelte component code for 'src/lib/components/MessageList.svelte' and 'src/lib/components/Message.svelte'.
  - 'MessageList.svelte' should accept a prop 'messages' (an array of objects, each with 'role' and 'content' properties). It should use an '#each' block to iterate over

    the messages and render a 'Message' component for each.

    - 'Message.svelte' should accept a 'message' object prop. It should use a conditional class directive ('class:user-message={message.role === 'user'}' and 'class:assistant-message={message.role === 'assistant'}') to style the message bubble differently based on the role.

- 2. Implement State Management and API Service:
  The client-side logic for managing state and communicating with the backend is implemented next.
  AI Coder Prompt:
  1. In 'src/lib/stores.ts', create a Svelte 'writable' store initialized with an empty array. This store will hold the chat message objects.
  2. In 'src/lib/api.ts', create a class named 'ChatService'. The constructor should accept a 'conversation_id' and establish a WebSocket connection to ws://<your_runpod_ip>:8000/ws/chat/{conversation_id}. Implement a 'sendMessage(message: string)' method that sends data over the socket. Implement a mechanism (e.g., an 'onMessage' callback) that allows other parts of the application to listen for messages received from the server.
- 3. Assemble the Main Chat View:
  Finally, the components and logic are brought together in the main application view.
  AI Coder Prompt:
  In 'src/routes/+page.svelte', create the main chat interface.
  1. Import the 'messages' store and the 'ChatService'.
  2. In the script section, subscribe to the 'messages' store using the '$' syntax. Instantiate the 'ChatService'.
  3. Configure the 'ChatService' so that when a new message token is received, it updates the 'content' of the last message in the store array. When the server signals the end of a stream, a new empty assistant message should be added to the array in preparation for the next response.
  4. In the markup section, render the 'MessageList' component, passing the '$messages' store value to it.
  5. Render a 'ChatInput' component (a form with a text input and a button). When the form is submitted, it should add a new user message to the store and call the 'sendMessage' method of the 'ChatService'.

C. Integration and End-to-End Testing

With all individual components built, the final step of this phase is to run the entire system and perform a smoke test.

1. **On Runpod:** Ensure the Milvus Docker containers are running (docker-compose ps).
2. **On Runpod:** Start the llama.cpp inference server in a terminal session (./start_llm.sh).
3. **On Runpod:** Start the FastAPI backend server in another terminal session (source venv/bin/activate && uvicorn main:app --host 0.0.0.0 --port 8000 --reload).

4. **On Local Machine:** Start the Svelte development server (npm run dev).
5. Open the Svelte application in a web browser (http://localhost:5173).
6. Send a test message (e.g., "What is a variable in Python?").
7. Verify that the message appears in the UI, a response is streamed back from the AI, and the final message is displayed correctly. This confirms that all components are communicating as expected.

IV. System Evaluation and Initial Deployment (Week 6: TEST & DEPLOY)

Following the successful assembly of the MVP, this week is dedicated to a structured evaluation of its performance and the preparation of a deployable, containerized version of the application. The goal is to move from a development setup to a stable, reproducible artifact.

A. MVP Evaluation Protocol

A systematic evaluation is necessary to objectively measure the quality of the AI mentor's responses and identify areas for improvement.

- **Objective:** To quantitatively and qualitatively assess the MVP's retrieval accuracy, generation quality, and overall correctness.
- **Methodology:**
    1. **Question Bank Creation:** A curated set of 20 questions will be developed, derived directly from the content of the ingested computer science documents. This ensures the evaluation is grounded in the system's knowledge base. The questions will be categorized to test different capabilities:
        - **Factual Recall (10 questions):** e.g., "What is the Big O notation for Quicksort's worst-case performance?"
        - **Conceptual Explanation (5 questions):** e.g., "Explain the principle of encapsulation in object-oriented programming."
        - **Comparative Analysis (3 questions):** e.g., "Compare and contrast TCP and UDP."
        - **Code Generation (2 questions):** e.g., "Provide a simple Python function to demonstrate recursion by calculating a factorial."
    2. **Evaluation Criteria:** Each response generated by the AI mentor will be scored by a human evaluator on a 1-5 scale (1=Poor, 5=Excellent) against the following criteria:
        - **Answer Correctness:** How factually accurate is the information provided in the generated response?
        - **Context Relevance:** How relevant were the source documents retrieved by

    the RAG system to the user's question? (This can be inspected via logs or by adding a debug view to the UI).
    - **Clarity and Coherence:** Is the response well-structured, easy to understand, and grammatically correct?
    - **Hallucination Check (Binary):** Does the response contain any information that is factually incorrect or not supported by the retrieved context? (Yes/No).
- **Execution and Analysis:** Each of the 20 questions will be posed to the system. The generated answer, retrieved sources, and scores for each criterion will be recorded in a spreadsheet. The aggregated results will provide clear, data-driven insights into the system's strengths and weaknesses. For example, consistently low scores on "Context Relevance" would suggest that the document chunking strategy or the embedding model needs refinement. High rates of hallucination might indicate that the generation prompt needs to be more restrictive, instructing the model to adhere strictly to the provided context.

B. Iterative Refinement

Based on the analysis of the evaluation data, targeted improvements will be implemented. This is a focused effort to address the most significant issues identified.

- **Example Refinement (Prompt Engineering):** If the evaluation reveals that the AI's tone is too generic and not sufficiently "mentor-like," the system prompt for the generate node in the LangGraph workflow will be refined. Effective prompt engineering is crucial for shaping the AI's persona and ensuring its output aligns with pedagogical goals.[29]
  AI Coder Prompt:
  In 'backend/app/services/agent_graph.py', locate the prompt template used by the 'generate' node. Modify the system message within this prompt to be: "You are an expert Computer Science mentor. Your role is to help students understand complex topics. Explain concepts clearly and concisely. When a student asks a question, first provide a direct answer, then offer a simple analogy or example to aid understanding. Base your answers strictly on the provided context documents. Cite the sources you used by referencing their filenames at the end of your response."
  After implementing changes, a subset of the question bank will be re-run to verify that the refinement had the intended effect.

C. Containerization for Deployment

The final step of this phase is to containerize the entire application stack. This process encapsulates each service and its dependencies into a portable, isolated container, creating a single, declarative deployment artifact using Docker Compose. This is the cornerstone of achieving a modular and scalable system.

1. **Backend Dockerfile:** A Dockerfile will be created in the backend directory. It will use a Python base image, copy the application code, install the dependencies from requirements.txt, and define the command to run the Uvicorn server.
2. **Frontend Dockerfile:** A Dockerfile will be created in the frontend directory. It will utilize a multi-stage build. The first stage will use a Node.js image to build the SvelteKit application into static assets. The second, final stage will use a lightweight Nginx image to copy in the built assets and serve them efficiently.
3. **Update Docker Compose:** The root docker-compose.yml file will be updated to include the new backend and frontend services. It will also be modified to include a service definition for the llama.cpp server, using a pre-built image or a custom one if necessary, ensuring the entire application can be launched with a single docker-compose up command.

This containerized setup ensures that the application can be deployed consistently across different environments, from a local machine to a cloud server, eliminating common deployment issues and providing a clear path for future scaling and management.

V. Critical Analysis and Strategic Roadmap (Week 7: REVIEW)

The final week of the project is dedicated to a comprehensive review of the plan and its execution, as well as the strategic planning of future development phases. This reflective process is essential for understanding the system's limitations and charting a course for its evolution from an MVP to a mature educational tool.

A. Critical Review of the Project Plan

A candid assessment of the project plan reveals potential bottlenecks, risks, and areas for improvement in future iterations.

- **Potential Bottlenecks:**
  - **Data Ingestion Throughput:** The process of parsing, chunking, and generating embeddings for a large corpus of documents (e.g., dozens of textbooks) is computationally intensive and time-consuming. The current plan executes this as a synchronous script, which could block development. For larger datasets, this process should be re-architected as an asynchronous background job.
  - **Agentic Workflow Latency:** The multi-step reasoning loop of the LangGraph agent (retrieve -> grade -> potentially rewrite -> generate) introduces more LLM calls than a simple RAG chain. This will result in higher end-to-end latency. The evaluation phase must rigorously measure the time from query submission to the

    first token of the response to ensure it remains within an acceptable range for a real-time chat application.
  - **GPU Memory Constraints:** While the 24GB GPU is sufficient for the specified 7B model and a 4096-token context, future enhancements such as using a larger model (e.g., 13B or 34B) or significantly increasing the context window could exceed the available VRAM. This represents a hard ceiling on the model's capabilities within the current hardware budget.
- **Identified Risks:**
  - **Quantized Model Quality:** The entire system's performance is fundamentally dependent on the quality of the open-source, quantized LLM. While models like Mistral 7B are highly capable, they may still struggle with nuanced reasoning, exhibit subtle biases, or fail on highly complex, domain-specific queries compared to larger, proprietary models.
  - **Retrieval Efficacy:** The principle of "garbage in, garbage out" applies directly to the RAG system. The effectiveness of the vector search is paramount. A suboptimal chunking strategy, a less capable embedding model, or poorly formatted source documents can lead to the retrieval of irrelevant context, which will degrade the quality of the final answer, even with the agent's self-correction loop.
  - **Dependency Volatility:** The project relies on a stack of rapidly evolving open-source libraries, particularly LlamaIndex and LangGraph. A minor version update in one library could introduce breaking changes or subtle incompatibilities with another. This risk is mitigated in the plan by pinning specific library versions in requirements.txt and package.json, ensuring a stable and reproducible build.

B. Strategic Roadmap for Future Development

The MVP serves as a robust foundation. The following roadmap outlines a strategic vision for evolving the AI mentor into a more sophisticated and pedagogically effective platform.

- **Phase 2: Enhanced Pedagogical Interaction**
  - **Socratic Scaffolding Agent:** Evolve the agent's logic beyond direct question-answering. The graph will be expanded with new nodes and conditional logic to implement scaffolding. For instance, if a student asks for a complete code solution, the agent's new logic would first prompt them to describe their current approach or identify where they are stuck. The agent would then provide targeted hints or explain a relevant concept rather than the full answer, guiding the student toward their own solution.[30]
  - **Metacognitive Feedback Loop:** Introduce a feature where students can submit their code snippets for review. The AI mentor would use a new set of tools (e.g., a static code analyzer) and specialized prompts to provide feedback not just on correctness (syntax errors) but on style, efficiency, and the underlying problem-solving approach. This encourages students to reflect on *how* they solve

problems.

- **Phase 3: Personalization and Advanced Capabilities**
  - **Persistent User Profiles:** Implement a database (e.g., PostgreSQL) to store user data and conversation histories. This would allow the AI mentor to build a profile for each student over time, tracking their strengths, weaknesses, and common misconceptions. Future interactions could then be personalized, with the mentor proactively suggesting topics for review or adjusting the complexity of its explanations based on the student's history.
  - **Multi-Modal Input:** Leverage the capabilities of llama.cpp to serve multi-modal models like LLaVA.[2] The frontend and backend would be enhanced to allow students to upload images, such as screenshots of error messages, diagrams of data structures, or handwritten notes. The mentor could then analyze these images as part of its context for generating a response.
  - **Knowledge Graph Integration:** Augment the existing vector-based retrieval with a knowledge graph. Using LlamaIndex, entities and relationships can be extracted from the source texts and stored in a graph database (e.g., Neo4j or Memgraph).[32] This would enable the agent to answer more complex, multi-hop questions that require understanding the relationships between concepts (e.g., "How does the choice of data structure impact the time complexity of an algorithm?").[22]
- **Phase 4: Platform and Ecosystem Integration**
  - **Learning Management System (LMS) Integration:** Develop plugins or APIs to embed the AI mentor directly within popular LMS platforms like Moodle, Canvas, or Blackboard. This would make the tool seamlessly accessible to students within their existing educational workflow.
  - **Collaborative Learning Environments:** Create "group study" channels where multiple students can interact with the AI mentor in a shared conversational context. The agent could facilitate discussions, pose questions to the group, and summarize key points, fostering a collaborative learning experience.

Works cited

1. An Introduction to Using FastAPI - Refine dev, accessed October 8, 2025, https://refine.dev/blog/introduction-to-fast-api/
2. OpenAI Compatible Web Server - llama-cpp-python, accessed October 8, 2025, https://llama-cpp-python.readthedocs.io/en/latest/server/
3. PyMuPDF, accessed October 8, 2025, https://mupdf.com/pymupdf
4. FastAPI Best Practices and Conventions we used at our startup - GitHub, accessed October 8, 2025, https://github.com/zhanymkanov/fastapi-best-practices
5. Svelte framework: a high-performance front-end framework optimized at compile time | by Tianya School, accessed October 8, 2025, https://tianyaschool.medium.com/svelte-framework-a-high-performance-front-e

nd-framework-optimized-at-compile-time-909b8c4f1fed
6. The Rise of Svelte: Revolutionizing Front-end Development | by Chaewonkong | Medium, accessed October 8, 2025, https://medium.com/@chaewonkong/the-rise-of-svelte-revolutionizing-front-end-development-d6a3a469ff9d
7. Getting started with Svelte - Learn web development - MDN, accessed October 8, 2025, https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Frameworks_libraries/Svelte_getting_started
8. What is Milvus? - IBM, accessed October 8, 2025, https://www.ibm.com/think/topics/milvus
9. Milvus | Open-source Vector Database created by Zilliz, accessed October 8, 2025, https://zilliz.com/what-is-milvus
10. How to Use llama.cpp to Run LLaMA Models Locally - Codecademy, accessed October 8, 2025, https://www.codecademy.com/article/llama-cpp
11. Running OpenAI's server Locally with Llama.cpp | by Tom Odhiambo | Medium, accessed October 8, 2025, https://medium.com/@odhitom09/running-openais-server-locally-with-llama-cpp-5f29e0d955b7
12. Introduction to RAG | LlamaIndex Python Documentation, accessed October 8, 2025, https://developers.llamaindex.ai/python/framework/understanding/rag/
13. Converting PDFs to Images with PyMuPDF: A Complete Guide - Artifex Software, accessed October 8, 2025, https://artifex.com/blog/converting-pdfs-to-images-with-pymupdf-a-complete-guide
14. PyMuPDF is a high performance Python library for data extraction, analysis, conversion & manipulation of PDF (and other) documents. - GitHub, accessed October 8, 2025, https://github.com/pymupdf/PyMuPDF
15. Agentic RAG Application using LlamaIndex — Router Query Engine | by Abdul Samad, accessed October 8, 2025, https://medium.com/@samad19472002/agentic-rag-application-using-llamaindex-router-query-engine-5b3f7b7feb75
16. Milvus is a high-performance, cloud-native vector database built for scalable vector ANN search - GitHub, accessed October 8, 2025, https://github.com/milvus-io/milvus
17. Building Multi-Agent Systems with LangGraph | by Clearwater Analytics Engineering, accessed October 8, 2025, https://medium.com/cwan-engineering/building-multi-agent-systems-with-langgraph-04f90f312b8e
18. LangGraph: Multi-Agent Workflows - LangChain Blog, accessed October 8, 2025, https://blog.langchain.com/langgraph-multi-agent-workflows/
19. Building Multi-Agent Systems with LangGraph: A Step-by-Step Guide | by Sushmita Nandi, accessed October 8, 2025, https://medium.com/@sushmita2310/building-multi-agent-systems-with-langgraph-a-step-by-step-guide-d14088e90f72

20. How to Build LangGraph Agents Hands-On Tutorial - DataCamp, accessed October 8, 2025, https://www.datacamp.com/tutorial/langgraph-agents
21. LlamaIndex vs LangGraph: How are They Different? - ZenML Blog, accessed October 8, 2025, https://www.zenml.io/blog/llamaindex-vs-langgraph
22. Knowledge Graph RAG Query Engine | LlamaIndex Python Documentation, accessed October 8, 2025, https://developers.llamaindex.ai/python/examples/query_engine/knowledge_graph_rag_query_engine/
23. Self-Reflective RAG with LangGraph - LangChain Blog, accessed October 8, 2025, https://blog.langchain.com/agentic-rag-with-langgraph/
24. FastAPI, accessed October 8, 2025, https://fastapi.tiangolo.com/
25. Building a Real-time Dashboard with FastAPI and Svelte | TestDriven.io, accessed October 8, 2025, https://testdriven.io/blog/fastapi-svelte/
26. Preparing FastAPI for Production: A Comprehensive Guide | by Raman Bazhanau | Medium, accessed October 8, 2025, https://medium.com/@ramanbazhanau/preparing-fastapi-for-production-a-comprehensive-guide-d167e693aa2b
27. Simple chat Application using Websockets with FastAPI - GeeksforGeeks, accessed October 8, 2025, https://www.geeksforgeeks.org/python/simple-chat-application-using-websockets-with-fastapi/
28. Building a Real-Time Chat Application with FastAPI and WebSockets - Medium, accessed October 8, 2025, https://medium.com/@palanikalyan27/building-a-real-time-chat-application-with-fastapi-and-websockets-6d72f06916e4
29. Prompt Engineering for Chatbot—Here's How [2025] - Voiceflow, accessed October 8, 2025, https://www.voiceflow.com/blog/prompt-engineering
30. AI Prompt Engineering Tips for Teachers - AVID Open Access, accessed October 8, 2025, https://avidopenaccess.org/resource/ai-prompt-engineering-tips-for-teachers/
31. Prompt Engineering - The Center for Digital Learning and Innovation (CDLI), accessed October 8, 2025, https://cdli.ehe.osu.edu/ai-in-ehe/ai-101/prompt-engineering/
32. How to build multi-agent RAG system with LlamaIndex? - Memgraph, accessed October 8, 2025, https://memgraph.com/blog/multi-agent-rag-system
33. Knowledge Graph Query Engine | LlamaIndex Python Documentation, accessed October 8, 2025, https://developers.llamaindex.ai/python/examples/query_engine/knowledge_graph_query_engine/