Step 1: The "Skeleton" (Environment & Auth) Goal: Set up a place where code can live and users can log in.

The Work: Initialize a Next.js or React frontend and a Node.js/Express backend.

Authentication: Use Supa base Auth or Clerk. You don't want to code your own login system; it’s a security risk.

Folder Structure: Organize by "Features" (e.g., /chat, /booking, /dashboard) rather than just "Components."

Step 2: The "Brain" (The AI RAG Pipeline) This is the most technical part of your project. You aren't just "talking" to ChatGPT; you are giving it a brain based on your university's data. This is called Retrieval-Augmented Generation (RAG).

The Coding Workflow: PDF Ingestion: Write a script (using langchain or PyPDF) to read your university PDFs.

Chunking: Split that text into small pieces (e.g., 500 words each).

Embedding: Send those pieces to OpenAI’s text-embedding-3-small model. It turns text into a list of numbers (a vector).

Storage: Save those numbers in a Vector Database (like Supabase Vector or Pinecone).

Retrieval Logic: When a student asks a question, the code "searches" the numbers in the database to find the closest match.

Step 3: The "Engine" (Database & Logic) Goal: Coding the "Low" and "Mid" priority items like room bookings.

Database Design: Create tables for Users, Rooms, Bookings, and ServiceRequests.

Conflict Logic: This is a classic coding challenge. You must write a function to check if a room is already booked before saving a new entry.

Example Logic: IF (requested_start_time < existing_end_time AND requested_end_time > existing_start_time) THEN REJECT.

Step 4: The "Face" (Frontend Integration) Goal: Making the code look professional and easy to use.

State Management: Use React Context or Zustand to keep the student's data consistent across pages.

API Fetching: Use React Query (TanStack) to handle data loading, "Loading..." spinners, and error messages when the server is down.

The Chat UI: Build a streaming interface (like ChatGPT) so the response appears word-by-word.