AI Study Assistant: Backend Project Description

1. Project Overview

The AI Study Assistant is a high-performance, local-first backend system designed to combat **information overload** for students. Our primary goal is to transform static, dense educational documents (like PDF textbooks) into dynamic, interactive knowledge bases. By leveraging Retrieval-Augmented Generation (RAG) and specialized Natural Language Processing (NLP) models, we enable students to find precise answers, generate summaries, and create self-assessment materials instantly, improving study efficiency and focus.

The Problem Addressed

Traditional study methods, particularly searching through large PDF documents for specific facts, are time-consuming and inefficient. The AI Study Assistant provides a centralized, intelligent service to automate these tasks, allowing students to focus on learning and comprehension.

2. Core Capabilities and Features

The backend powers three primary study tools:

2.1. RAG (Retrieval-Augmented Generation)

- **Function:** Finds precise, grounded answers to user questions based *only* on the uploaded documents.
- **Mechanism:** Uses a local FAISS vector index for high-speed similarity search, ensuring the LLM response is anchored directly to the source material.

2.2. Hierarchical Summarization

- **Model: BART-large-CNN** (Trained for abstractive summarization).
- **Strategy:** Implements "Hierarchical Synthesis." For lengthy documents (e.g., full chapters), the system first chunks the text, summarizes each chunk individually, and then synthesizes those smaller summaries into a final, cohesive overview.

2.3. Automated Q&A Generation

- **Model: FLAN-T5** (Fine-tuned for instructional tasks).
- **Function:** Parses sections of the document text and automatically generates a set of potential exam questions and their correct, ground-truth answers. This facilitates self-assessment and active recall study techniques.

3. Technology Stack

Core Backend & Orchestration

Component	Purpose	Details
FastAPI	Web Framework	High-performance, asynchronous Python framework for robust API endpoints.
Python 3.10+	Language	Used for type-safe, maintainable code.
LangChain	Orchestration	Manages the complex pipelines between text processing, vector stores, and LLM calls.

Inference & Machine Learning

Model/Tool	Category	Use Case
Ollama	Local LLM Server	Primary provider for local inference (e.g., Llama 3/Mistral) for privacy and speed.
Google Gemini	Cloud LLM Fallback	Cloud API integration for generation when the local server is unavailable or overloaded.
FAISS	Vector Store	Efficient local indexing and sub-millisecond similarity search.
sentence-transformers	Embedding Model	Used to convert document chunks into vectors for indexing.

4. "Local-First" Philosophy

We designed the system with a strong "Local-First" priority, meaning the core operations—ingestion, indexing, retrieval, and generation—can occur primarily on the user's device or local network.

- **Data Privacy:** Uploaded study documents (PDFs) are chunked and indexed locally; they **never** leave the user's machine, satisfying stringent privacy requirements.
- **Zero Cost:** Users avoid recurring per-token API charges for standard RAG and Q&A operations.
- **Offline Access:** The system remains fully functional even without an active internet connection (using local FAISS and Ollama).
- **Low Latency:** Immediate context retrieval from the local FAISS index ensures fast response times.

5. System Architecture and Pipeline

The system is built as a sequential processing pipeline:

1. **Ingestion (/upload-pdf):** The PDF is received, raw text is extracted, and the text is split into **semantic chunks** with overlap to maintain context across boundaries.
2. **Indexing (src/rag_retriever):** These chunks are embedded (vectorized) using a sentence-transformer model. The resulting vectors are stored in a **FAISS index**, which is then serialized and saved locally.
3. **Retrieval (/ask):** A user query is received, vectorized, and used to query the FAISS index to fetch the most relevant *top-k* chunks (context).
4. **Synthesis (improved_llm_reader.py):** The context and the original query are passed to the **Provider-Aware LLM Reader**, which uses its primary (Ollama) or fallback (Gemini) provider to generate a grounded, natural language answer.

6. API Specification

Method	Endpoint	Description
POST	/upload-pdf	Ingests a PDF file, triggering chunking and indexing processes.
POST	/ask	Executes the RAG pipeline to answer a user's question.
POST	/summarize	Executes the BART hierarchical summarization process on the document content.
POST	/generate-qa	Triggers the FLAN-T5 model to generate a set of questions and answers.
GET	/health	Returns the current status of core components (FAISS index and LLM providers).

7. Future Roadmap

We plan to enhance the system with the following capabilities:

- **Multi-Modal Support:** Implementing Optical Character Recognition (OCR) to process scanned PDFs and textbooks containing image-based text.
- **Knowledge Graphs:** Integrating graph-based RAG methods to model and query semantic relationships between concepts across different documents or chapters more effectively.
- **User Interface:** Developing a dedicated React/Next.js frontend application to provide a user-friendly interface for all API functionalities.
- **Fine-Tuning:** Training a small LoRA adapter on local, academic data to specialize the LLM's tone and style for textbook-specific responses.