Case Study: The "Veltris Intelligent Doc-Bot"

1. Introduction & Context

Scenario:
Veltris has a client in the Technical Services sector. Their field technicians currently struggle to find answers quickly while on-site. They have access to extensive technical documentation, but searching through manual PDFs or Markdown files is inefficient.

The Ask:
You are required to build a Production-Ready RAG (Retrieval-Augmented Generation) System that allows a user to query technical documentation and get accurate, sourced answers via a clean web interface.

Role Expectation:
As a Senior/Mid-level AI Engineer, we are looking beyond just "calling an API." We are evaluating your ability to build a complete end-to-end application: from data ingestion to API design, system architecture, and a user-friendly frontend.

2. The Data

You will use the **HuggingFace Documentation** dataset as a proxy for "Technical Manuals."

- **Dataset:** https://huggingface.co/datasets/m-ric/huggingface_doc
- **Instruction:** You do not need to process the entire dataset. Select a specific subset (e.g., only the transformers or accelerate folder) to simulate a specific product manual.
- **Goal:** Your system must ingest these files and treat them as the "Knowledge Base" for the bot.

3. Enforced Tech Stack

To ensure consistency and evaluate your proficiency with modern standard tools, you **must** use the following stack. Please use the **latest stable versions** (as of Jan 2026).

- **Frontend: Streamlit** (Must include chat interface).
- **Backend API: FastAPI**.
- **Orchestration:** LangChain (Latest Stable) OR LlamaIndex.
- **LLM:** OpenAI (gpt-3.5-turbo/gpt-4o) OR a local LLM via Ollama (e.g., Llama 3) if you do not have API access.
- **Vector Database:** ChromaDB (Local persistence) or FAISS.
- **Deployment:** Docker (Solution must run via docker-compose up).

4. Functional Requirements

A. Data Ingestion Pipeline

- Write a script/service to ingest the documentation subset.
- Implement a robust **chunking strategy**. (Tip: Be prepared to explain why you chose specific chunk sizes and overlap for technical documentation).
- Generate embeddings and store them in the Vector DB.

B. The RAG Backend (FastAPI)

- Create an endpoint /chat that accepts a JSON payload { "query": "..." }.
- **Strict Constraint:** The bot must answer *only* from the context. If the answer is not in the docs, it must reply: "I cannot find the answer in the provided documentation."
- **Source Citation:** The API response must include the source_file (and page_number or section if available) used to generate the answer.

C. The Frontend (Streamlit)

- Build a simple, clean UI.
- **Components:**
  - **Chat Interface:** Use st.chat_input and st.chat_message for a modern look.
  - **Sidebar:** Display which documentation subset is currently loaded/active.
  - **Status Indicators:** Show a spinner while the model is "thinking."

D. MLOps & Engineering Standards

- **Dockerization:** The Backend and Frontend must run as separate services orchestrated by docker-compose.
- **Logging:** Implement structured logging to track queries and retrieval latency.
- **Code Quality:** Use Type Hinting (typing), Pydantic models for validation, and clean module structure.

5. Deliverables

Please provide a link to a **private GitHub repository** (invite: [Insert Your GitHub Username]) containing:

1. **Source Code:** Modular structure (e.g., frontend/, backend/, ingestion/).
2. **requirements.txt / pyproject.toml:** Explicit version pinning is required.
3. **docker-compose.yml**: Must orchestrate the entire stack.
4. **System Diagram:** A visual representation (JPG/PNG/PDF) of your architecture. Show the flow of data from ingestion -> Vector DB -> Retrieval -> LLM -> UI.
5. **README.md**:
   - **Setup Instructions:** How to run the solution.
   - **Architecture Decision Record (ADR):** Briefly explain your choice of Chunking Strategy and Prompt Engineering technique.
6. **Video Walkthrough (Max 5 mins):** A short screen recording showing the system running and a successful QA flow.

6. Evaluation Criteria

Your submission will be graded on a total of 100 points:

- **Architecture & Engineering Quality (40%):**
  - Is the code modular and production-ready?
  - Does docker-compose up work seamlessly?
  - Is the API structured correctly using Pydantic and proper Error Handling?
- **RAG Implementation & Performance (40%):**
  - Does the retrieval logic make sense for technical docs?
  - Does the model handle hallucinations (answering only from context)?
  - Are citations accurate?
- **UI/UX (10%):**
  - Is the Streamlit interface clean and usable?
  - Does it handle loading states and chat history gracefully?
- **Documentation & Diagrams (10%):**
  - Is the System Diagram clear?
  - Are the README and ADR easy to understand?

Time Limit: 48 Hours from receipt of this document.