AI Lab Report Agent: Architecture Document

1. Overview

This document outlines the architecture of the AI Lab Report Agent, a web-based application designed to automate the generation of formal engineering and science lab reports. The agent takes a lab manual and a set of experimental observations as input and produces a complete, structured report, significantly reducing the time and effort required for documentation.

2. System Architecture & Interaction Flow

The agent is built on a multi-component pipeline architecture orchestrated by a Flask backend. Each component is a specialized module responsible for a specific task in the report generation process.

The interaction flow is as follows:

1. **User Input:** The user accesses a web interface, uploads a lab manual (e.g., PDF, DOCX), and provides experimental observations in JSON format.
2. **Contextualization (RAG):** The backend extracts the raw text from the manual. This text is processed by a Retrieval-Augmented Generation (RAG) component, which identifies and retrieves the core contextual information, such as the experiment's Aim, Theory, and Procedure.
3. **Code Generation:** The retrieved context and the user's observations are sent to a specialized "Coder Agent." This AI model generates a Python script to perform the necessary calculations for the experiment.
4. **Code Execution:** The generated Python script is executed securely, and its output (the calculated results) is captured.
5. **Report Synthesis:** A "Report Writer Agent" receives the RAG context, user observations, and the calculated results. It synthesizes all this information into a coherent, well-structured, formal lab report.
6. **Output:** The final report is sent back to the user's web browser for display, copying, or downloading.

3. Core Components

3.1. Frontend (Web UI)

- **Technology:** Flask, HTML, CSS, JavaScript
- **Purpose:** Provides a simple and intuitive interface for user interaction. It allows for file uploads, text entry for observations, and displays the final generated report along with usability features like "Copy" and "Download."

3.2. Backend (Flask Application)

- **Technology:** Python, Flask
- **Purpose:** Acts as the central orchestrator of the entire agent. It handles HTTP requests from the frontend, manages the multi-step pipeline, and calls the various components in sequence.

3.3. RAG Component (Contextualizer)

- **Purpose:** To extract relevant theoretical and procedural context from lengthy lab manuals efficiently.
- **Models/Libraries:** LangChain, FAISS (vector store), all-MiniLM-L6-v2 (embedding model).
- **Reasoning:** This approach is highly efficient. Instead of passing a large, multi-page document to an LLM (which is slow and costly), the RAG system quickly pinpoints the most relevant paragraphs. FAISS was chosen for its speed as an in-memory vector store, and all-MiniLM-L6-v2 provides a strong balance of performance and a small footprint for creating embeddings.

3.4. Coder Agent

- **Purpose:** To generate executable Python code for experimental calculations.
- **Model:** Fine-tuned microsoft/Phi-3-mini-4k-instruct (hosted as Barghav777/phi3-lab-report-coder).
- **Deployment:** Accessed via the Hugging Face Inference API.
- **Reasoning:** The Phi-3 model was chosen for its strong coding capabilities and small size. The model was fine-tuned to specialize in the specific domain of lab calculations. Deploying it via the Hugging Face Inference API was a strategic decision to overcome local hardware limitations, eliminate complex dependency issues (like flash-attn on Windows), and ensure fast, reliable performance.
- **Location:** https://huggingface.co/Barghav777/phi3-lab-report-coder
- **Dataset:** Uploaded in github

3.5. Report Writer Agent

- **Purpose:** To synthesize all structured data into a final, human-readable lab report.
- **Model:** Llama3-70b-8192 accessed via the Groq API.
- **Reasoning:** The 70B parameter version of Llama 3 was chosen for its state-of-the-art ability to generate high-quality, coherent, and stylistically appropriate prose. The Groq API was chosen for its industry-leading inference speed, which is critical for providing a responsive user experience.