Project Documentation: AI Data Analyst

Version: 1.0
Date: August 16, 2025

1. Executive Summary

The AI Data Analyst is a full-stack web application designed to democratize data analysis. It empowers non-technical users to upload their own datasets (CSV, Excel, etc.) and gain insights by asking questions in plain, natural language. The system leverages a Large Language Model (LLM) to translate user queries into executable SQL, which is then run against a secure, private database. The application features a multi-user environment with secure authentication, persistent chat histories for each dataset, dynamic chart generation, a saveable chart gallery, and a core focus on transparency by showing users the exact query and data used to generate each answer.

2. Core Idea & Vision

The project's vision was to eliminate the barrier of coding for data analysis. Many individuals and small businesses have valuable data but lack the technical skills to query and visualize it. The core idea was to create a "ChatGPT-like" experience for structured data, where the conversation itself is the interface for analysis.

A key architectural principle was **privacy and security**. Instead of sending entire user datasets to an external AI service, we designed a system where only the data's *structure* (i.e., column names) is used by the AI to generate a SQL query. This query is then executed locally on our backend, ensuring the user's raw data remains secure and private.

3. System Architecture

The application is built on a modern, robust technology stack chosen for performance, scalability, and ease of development.

3.1. Technology Stack

- **Backend Framework: FastAPI** - Chosen for its high performance, asynchronous capabilities, automatic API documentation (Swagger UI), and Pydantic data validation.
- **Database: MySQL / MariaDB** - A reliable, open-source relational database for storing user data, dataset metadata, and chat histories.
- **AI & Orchestration: LangChain** - Used to create the Text-to-SQL pipeline, manage prompts, and interact with the LLM.
- **LLM Provider: OpenRouter** - Acts as a gateway to various Large Language Models, allowing for flexibility and access to powerful free-tier models like openai/gpt-oss-20b.
- **Data Handling: Pandas** - For robustly reading various file formats (.csv, .xlsx) and handling data manipulation before database insertion.
- **Database Interaction (ORM): SQLAlchemy** - The standard for Python database

interaction, providing both an ORM for structured data and a core engine for executing raw SQL queries.
- **Frontend: Vanilla HTML, CSS, and JavaScript** - A single-page application (SPA) approach was used for simplicity and direct control.
  - **Styling: Tailwind CSS** - For rapid, modern, and responsive UI development.
  - **Visualization: Chart.js** - A powerful and easy-to-use library for rendering dynamic charts.

3.2. Backend Architecture

The backend follows a modular structure for maintainability:

- /app/main.py: The entry point of the application, responsible for creating the FastAPI app and initializing database tables.
- /app/api/: Contains all API endpoint logic, versioned for future updates.
- /app/core/: Handles configuration (config.py) and security logic (security.py).
- /app/db/: Manages the database connection and session (database.py).
- /app/models/: Defines the SQLAlchemy ORM models for all database tables.
- /app/schemas/: Defines the Pydantic models for API data validation.
- /app/services/: Contains the core business logic, separating the AI (ai_service.py) and file processing (file_handler.py) from the API endpoints.

3.3. Database Schema

Four primary tables support the application's features:

1. users: Stores user credentials securely (hashed passwords).
2. datasets: Tracks uploaded files, linking each file to its owner (user_id) and its unique data table name.
3. chat_messages: Stores the full conversation history for every dataset, enabling contextual follow-up questions.
4. saved_charts: Stores the JSON data and user-provided labels for charts saved to the gallery, linked to a specific dataset.

4. Challenges Faced & Solutions

The development process involved several significant real-world challenges, each requiring a specific and robust solution.

- **Challenge 1: Large File Uploads & Database Limits**
  - **Problem:** Uploading large datasets (e.g., zomato.csv) resulted in Data too long for column and MySQL server has gone away errors.
  - **Analysis:** We discovered this was a two-part problem. First, Pandas' automatic schema inference was creating VARCHAR columns that were too small. Second, sending the entire DataFrame in one transaction was exceeding the database's max_allowed_packet size.
  - **Solution:** We engineered a robust, two-step file handler. First, we took **manual control of table creation**, building a CREATE TABLE statement that proactively uses

the MEDIUMTEXT type for all string columns. Second, we used the chunksize parameter in Pandas' to_sql function to break the data insertion into smaller, manageable batches, preventing connection timeouts.

- **Challenge 2: Database Version Incompatibility**
  - **Problem:** The AI consistently generated advanced SQL with the JSON_TABLE function to query JSON data, which failed on the user's older MariaDB 10.4 environment.
  - **Analysis:** The AI was writing *correct*, modern SQL, but the environment couldn't support it.
  - **Solution:** We implemented **advanced prompt engineering**. We added a specific set of instructions to the AI's prompt, commanding it to generate highly compatible, legacy-style SQL (ANSI SQL-92) and explicitly forbidding the use of modern functions. This forced the AI to generate slower but compatible queries using basic string manipulation, making the feature work on the user's existing setup.
- **Challenge 3: Lack of Conversational Memory**
  - **Problem:** The AI could not answer follow-up questions (e.g., "out of those, how many are legendary?").
  - **Analysis:** The system was stateless; each query was sent to the AI in isolation.
  - **Solution:** We implemented a **conversation history mechanism**. Before sending a new question, the backend now fetches the last 10 messages from the database, formats them into a conversational context, and prepends this history to the user's new question. This gives the AI the "short-term memory" needed to understand context.

5. How to Explain This Project in an Interview

This project is an excellent talking point in an interview. Here’s a structured way to present it:

1. The Elevator Pitch (Start with the "Why")
"I developed a full-stack AI Data Analyst application to solve a common problem: many people have valuable data but lack the coding skills to analyze it. My application provides a secure, multi-user platform where anyone can upload a dataset and get insights simply by asking questions in plain English, just like chatting with an expert."
2. The Technical Deep Dive (Explain the "How")
"For the tech stack, I chose FastAPI for its high performance and automatic documentation. The backend uses SQLAlchemy to interact with a MySQL database that stores user data and dataset metadata.
The core of the application is a Text-to-SQL pipeline I built using LangChain. When a user asks a question, I don't send their private data to the AI. Instead, I securely fetch the table's column names and, combined with the user's question and conversation history, I use an LLM via OpenRouter to generate a compatible SQL query. This query is then executed on my backend, and the results are used to generate a final answer, which can be text or a dynamic chart rendered with Chart.js on the frontend."
3. Showcase a Key Challenge (Demonstrate Problem-Solving)

"One of the most interesting challenges was handling large, real-world file uploads. Initially, files with long text fields would fail because of database column size limits. To solve this, I engineered a robust file handler in Python that takes full manual control. It first reads the file schema, then programmatically constructs and executes a CREATE TABLE statement using the MEDIUMTEXT type for all text columns to ensure sufficient space. Only after the table is securely created does it use Pandas with a chunksize parameter to insert the data in small batches, which also prevents database connection timeouts. This made the upload process resilient and scalable for any type of dataset."
4. The Result & Future
"The result is a fully functional, secure, and user-friendly application that successfully democratizes data analysis. If I were to continue developing it, my next steps would be to add more advanced visualization types, like scatter plots, and implement a feature to export query results to CSV, further empowering the user."

6. Potential Interview Questions

- **"Why FastAPI over Django or Flask?"**
  - *Answer:* "I chose FastAPI primarily for its performance, which is comparable to Node.js and Go, and its native support for asynchronous operations. The automatic data validation with Pydantic and the interactive Swagger UI documentation were huge productivity boosters that allowed me to build and test a robust API very quickly."
- **"How did you handle user authentication and security?"**
  - *Answer:* "I implemented a standard OAuth2 with JWT token-based authentication system. Passwords are never stored in plain text; they are hashed using passlib with bcrypt. Every protected endpoint requires a valid JWT, and for data-specific requests, I implemented an authorization check to ensure a user can only access the datasets they personally uploaded."
- **"Your Text-to-SQL approach is interesting. Why not send the data directly to the AI?"**
  - *Answer:* "Privacy and security were my top priorities. Sending user data to a third-party LLM is a significant security risk. My approach ensures that the user's raw data never leaves my controlled backend environment. By only sending the table schema (the column names), I minimize the data exposure while still giving the AI enough context to generate the necessary SQL query."
- **"What was the most difficult bug you had to solve?"**
  - *Answer:* "The most difficult bug was the MySQL server has gone away error during large file uploads. It was a combination of issues: incorrect data type inference by Pandas and exceeding the database's max_allowed_packet size. The solution required a multi-layered approach: forcing Pandas to read all columns as strings, manually constructing the CREATE TABLE SQL with robust MEDIUMTEXT types, and then using the chunksize parameter to stream the data to the database in manageable batches. This taught me a lot about the intricacies of data engineering pipelines."