DataFlow: An SQL LLM Agent for Intelligent Database Interaction

A MAJOR PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF

**BACHELOR OF TECHNOLOGY**

**(Computer Science & Engineering)**



|  |  |
| --- | --- |
| **Submitted By:**  Utkarsh Prajapati (2101660100061)  Anish Chauhan (2101660100013)  Adarsh (2101660100007)  Anshil Gautam (2101660100017) | **Submitted To:**  **Sh. Sri Nath Dwivedi**  Head of Department  Department of Computer Science & Engineering |

# Abstract

The DataFlow project is an intelligent database assistant that leverages Large Language Models (LLMs) to enable users to interact with SQL databases using natural language. This system, powered by the Google Gemini API, aims to bridge the gap between non-technical users and complex database query languages.

Key features include translating plain English questions into optimized SQL queries, executing these queries against connected MySQL databases, and retrieving results. Furthermore, the system can generate insightful summaries from the fetched data, providing a deeper understanding beyond raw output.

The application features a user-friendly, interactive chat interface built with HTML, Tailwind CSS, and JavaScript, allowing for a seamless user experience. The backend is developed using FastAPI in Python, managing database connections, query execution, and communication with the Gemini API. The system is designed to be schema-aware, dynamically fetching and utilizing the structure of multiple databases (including SQLLLM and StoreDB as examples generated by an auxiliary script) to formulate accurate queries.

Users can also directly execute SQL commands using a specific /run prefix. The project includes a data generation script (gen-data.py) using the Faker library to populate sample databases, facilitating testing and demonstration. This tool enhances data accessibility and empowers users to derive valuable information from their databases without requiring extensive SQL knowledge.

# Acknowledgement

We are highly grateful to the Head of Department (CSE), Dr. Ambedkar Institute of Technology for Divyangjan (AITD), Kanpur, for providing this opportunity to carry out the major project work.

The constant guidance and encouragement received from Sh. Shri Nath Dwivedi, Head of the Computer Science & Engineering Department, AITD, Kanpur, has been of great help in carrying out the project work and is acknowledged with reverential thanks.

We would like to express a deep sense of gratitude and thanks profusely to our project guide, Sh. Shri Nath Dwivedi, Head of the Computer Science & Engineering Department, AITD, Kanpur. Without his wise counsel and able guidance, it would have been challenging to complete the project in this manner.

We express gratitude to other faculty members of the Computer Science & Engineering department of AITD for their intellectual support throughout the course of this work.

Finally, WE are indebted to all whosoever have contributed in this report work.

**Utkarsh Prajapati** (2101660100061)

**Anish Chauhan** (2101660100013)

**Adarsh** (2101660100007)

**Anshil Gautam** (2101660100017)

**List of Figures**

| Figure No. | Figure Description | Page No. |
| --- | --- | --- |
| 3.1 | High-Level System Architecture Diagram for DataFlow | [Page] |
| 3.2 | Data Flow Diagram (DFD) for a User Query | [Page] |
| 3.3 | ER Diagram for SQLLLM Database | [Page] |
| 3.4 | ER Diagram for StoreDB Database | [Page] |
| 5.1 | Main Chat Interface of DataFlow | [Page] |
| 5.2 | Example of Natural Language Query and System Response | [Page] |
| 5.3 | Example of Direct SQL Execution using /run command | [Page] |
| 5.4 | Database Schema Display Panel | [Page] |
| 5.5 | Example of an Error Message Display | [Page] |

**List of Tables**

| Table No. | Table Description | Page No. |
| --- | --- | --- |
| 2.1 | Hardware and Software Requirements | [Page] |
| 3.1 | Structure of employees table in SQLLLM database | [Page] |
| 3.2 | Structure of salaries table in SQLLLM database | [Page] |
| 3.3 | Structure of products table in StoreDB database | [Page] |
| 4.1 | Key Technologies and Libraries Used | [Page] |
| 4.2 | Sample Test Cases for System Validation | [Page] |

**Table of Contents**

[Abstract 2](#_Toc197474297)

[Acknowledgement 3](#_Toc197474298)

[Chapter 1: Introduction 6](#_Toc197474299)

[1.1 Introduction to Project 6](#_Toc197474300)

[1.2 Project Category 6](#_Toc197474301)

[1.3 Objectives 6](#_Toc197474302)

[1.4 Problem Formulation 7](#_Toc197474303)

[1.5 Identification/Reorganization of Need 7](#_Toc197474304)

[1.6 Existing System 7](#_Toc197474305)

[1.7 Proposed System 8](#_Toc197474306)

[1.8 Unique Features of the System 8](#_Toc197474307)

[Chapter 2: Requirement Analysis and System Specification 9](#_Toc197474308)

[2.1 Feasibility Study 9](#_Toc197474309)

[2.2 Software Requirement Specification Document 10](#_Toc197474310)

[2.3 Validation 13](#_Toc197474311)

[2.4 Expected Hurdles 14](#_Toc197474312)

[2.5 SDLC Model to be Used 14](#_Toc197474313)

[Chapter 3: System Design 15](#_Toc197474314)

[3.1 Design Approach 15](#_Toc197474315)

[3.2 Detail Design 15](#_Toc197474316)

[3.3 System Design using various Structured Analysis and Design Tools 17](#_Toc197474317)

[Chapter 4: Implementation, Testing, and Maintenance 22](#_Toc197474318)

[4.1 Introduction to Languages, IDE’s, Tools and Technologies used for Implementation 22](#_Toc197474319)

[4.2 Coding Standards of Language Used \* Python (sql\_assistant.py, gen-data.py): 23](#_Toc197474320)

[4.3 Project Scheduling 23](#_Toc197474321)

[4.4 Testing Techniques and Test Plans 25](#_Toc197474322)

[Chapter 5: Results and Discussions 27](#_Toc197474323)

[5.1 User Interface Representation 27](#_Toc197474324)

[Chapter 6: Conclusion and Future Scope 31](#_Toc197474325)

[Conclusion 31](#_Toc197474326)

[Future Scope 31](#_Toc197474327)

[References/Bibliography 33](#_Toc197474328)

[Appendix 34](#_Toc197474329)

[Appendix A: Development Environment 34](#_Toc197474330)

[Appendix B: Key Code Snippets 34](#_Toc197474331)

# Chapter 1: Introduction

## Introduction to Project

DataFlow is an innovative software application designed to simplify interaction with SQL databases. In an era where data is paramount, the ability to access and understand database information quickly and efficiently is crucial. However, traditional methods often require proficiency in SQL (Structured Query Language), posing a barrier for many potential users. DataFlow addresses this challenge by employing Large Language Models (LLMs), specifically Google’s Gemini API, to translate natural language questions into executable SQL queries.

The project provides an intelligent interface that not only fetches data but also offers insights derived from the query results. It features a web-based chat interface where users can type their queries in plain English or execute SQL commands directly. The backend, built with Python and FastAPI, handles the core logic, including communication with the LLM, database operations via MySQL Connector, and schema discovery. The system is capable of interacting with multiple databases and dynamically adapts its query generation based on the available schema information. An auxiliary Python script (gen-data.py) is included to create and populate sample databases (SQLLLM and StoreDB) with realistic data using the Faker library, facilitating demonstration and testing of DataFlow’s capabilities.

## Project Category

This project falls under the category of **Application or System Development**, with a strong emphasis on being an **Internet-based** application due to its web interface. It also incorporates elements of **Research-based** development in its exploration of LLM integration for database querying.

## Objectives

The primary objectives of the DataFlow project are:

* To enable users to interact with SQL databases using natural language.
* To automatically generate optimized SQL queries from user questions via an LLM.
* To retrieve data by executing the generated SQL queries against MySQL databases.
* To provide users with meaningful insights generated by an LLM based on the fetched data.
* To offer a facility for users to execute direct SQL commands if needed.
* To build a user-friendly and interactive chat interface for a seamless experience.
* To make the system aware of the database schema (multiple databases, tables, and columns) to formulate accurate and relevant queries.

## Problem Formulation

Accessing information stored in relational databases typically necessitates knowledge of SQL. This requirement creates a significant hurdle for individuals who are not technically proficient in database languages but need to extract information for decision-making, analysis, or operational tasks. Writing correct and efficient SQL queries can be time-consuming and error-prone, even for those with some technical background. The problem, therefore, is to lower this barrier to entry and provide a more intuitive, conversational way to query and understand database content.

## Identification/Reorganization of Need

There is a growing need to democratize data access within organizations and for individual users. While Business Intelligence (BI) tools and other database clients exist, they often still require a degree of technical setup, understanding of data models, or specific query constructs. The DataFlow project identifies the need for a system that: \* Requires minimal technical expertise from the end-user. \* Understands user intent expressed in natural language. \* Automates the complex task of SQL query construction. \* Provides not just data, but also context or insights from that data. \* Can adapt to different database structures dynamically.

## Existing System

Traditional methods for interacting with SQL databases include:

* **SQL Clients:** Tools like MySQL Workbench, DBeaver, or command-line interfaces allow direct SQL query execution but require full SQL proficiency.
* **Business Intelligence (BI) Tools:** Platforms like Tableau or Power BI offer visual query builders and dashboards but often involve a learning curve, data modeling prerequisites, and may not be suitable for ad-hoc, conversational querying.
* **Custom Application Frontends:** Some applications have built-in reporting or search features, but these are typically pre-defined and lack the flexibility of arbitrary natural language querying. These systems generally lack the ability to understand and process queries phrased in natural human language without significant pre-configuration or direct SQL input.

## Proposed System

The proposed system, DataFlow, is an SQL LLM Agent that acts as an intelligent intermediary between the user and SQL databases. Its architecture comprises:

* **Frontend (index.html):** A web-based chat interface allowing users to input natural language questions or direct SQL commands. It displays generated SQL, results, insights, and database schema.
* **Backend (sql\_assistant.py):** A FastAPI application that:
* Receives user requests.
* Fetches and provides database schema to the LLM.
* Communicates with the Google Gemini API to:
* Translate natural language to SQL.
* Generate insights from query results.
* Provide conversational fallback responses.
* Connects to MySQL databases to execute queries.
* Formats and returns responses to the frontend.
* **LLM (Google Gemini):** The core intelligence for understanding language, generating SQL, and deriving insights.
* **Database Layer:** MySQL databases that the system connects to. The project includes gen-data.py to create sample SQLLLM and StoreDB databases.

The workflow involves the user typing a question, the backend fetching the relevant schema, the LLM generating an SQL query, the backend executing the query, the LLM generating insights from the results (if applicable), and the frontend displaying all this information to the user.

## Unique Features of the System

DataFlow offers several unique features:

* **Natural Language to SQL:** Core capability to convert user questions in plain English into executable SQL queries.
* **Multi-Database Schema Awareness:** Dynamically fetches and utilizes schema information from multiple non-system databases to generate contextually accurate queries, including fully qualified table names where necessary.
* **Insight Generation:** Leverages the LLM to provide textual summaries and potential follow-up questions based on query results.
* **Direct SQL Execution:** Allows users to bypass natural language and run specific SQL commands using the /run prefix.
* **Interactive Chat Interface:** Provides a modern, user-friendly chat UI for interaction.
* **Dual Sample Database Generation:** Includes a script to set up two distinct databases (SQLLLM for employee/salary data and StoreDB for product data) for comprehensive demonstration.
* **Conversational Fallback:** If an SQL query cannot be generated to answer a question based on the schema, the system attempts to provide a polite conversational response.

# Chapter 2: Requirement Analysis and System Specification

## 2.1 Feasibility Study

* **Technical Feasibility:**
  + **Technology Stack:** The project utilizes widely adopted and well-supported technologies: Python for backend logic, FastAPI for web framework, HTML/CSS/JavaScript for frontend, MySQL as the database, and Google Gemini API for LLM capabilities. These are all mature technologies with ample documentation and community support.
  + **Integration:** Integrating these components is technically feasible. FastAPI is designed for building APIs, and Python has robust libraries for MySQL connectivity (mysql-connector-python) and HTTP requests (httpx or requests for Gemini API, though the google-generativeai library handles this).
  + **LLM Capabilities:** The Gemini API is capable of code generation (including SQL) and text summarization, making it suitable for the project’s core tasks.
  + **Expertise:** The development requires knowledge of Python, web development fundamentals, SQL, and API integration, which are common skills in software development.
* **Economical Feasibility:**
  + **Software Costs:** The primary software components (Python, FastAPI, MySQL, VS Code) are open-source and free to use.
  + **API Costs:** The Google Gemini API has a free tier, but production use or high-volume requests may incur costs. This should be monitored.
  + **Development Costs:** Primarily the time and effort of the developer(s).
  + **Hardware Costs:** Standard development machine. Deployment would require hosting, which has associated costs (though can start with free/low-cost tiers for small scale).
  + Overall, the project is economically feasible for development and demonstration, with potential operational costs for API usage and hosting if scaled.
* **Operational Feasibility:**
  + **User-Friendliness:** The chat interface is designed to be intuitive, minimizing the learning curve for non-technical users.
  + **Deployment:** The FastAPI application can be deployed using standard Python web server gateways like Uvicorn.
  + **Maintenance:**
    - Updating dependencies (requirements.txt).
    - Managing API keys securely (.env file).
    - Monitoring LLM performance and potential changes in API.
    - Database backups and maintenance (standard DBA tasks, outside the scope of the app itself but relevant for the underlying data).
  + The system is operationally feasible, with standard maintenance practices for web applications.

## 2.2 Software Requirement Specification Document

* **Data Requirement:**
  + User input: Natural language questions, direct SQL commands.
  + Database schema: Information about databases, tables, columns, and their types.
  + Database content: The actual data stored in the tables.
  + Configuration: Database credentials, API keys (stored in .env).
* **Functional Requirements:**
  + FR1: The system shall accept natural language queries from the user through a chat interface.
  + FR2: The system shall fetch and display the schema of accessible non-system MySQL databases.
  + FR3: The system shall use the Google Gemini API to translate natural language queries into SQL queries, considering the database schema.
  + FR4: The system shall execute the generated SQL queries (primarily SELECT) against the appropriate MySQL database.
  + FR5: The system shall display the results of SELECT queries in a tabular format.
  + FR6: The system shall use the Google Gemini API to generate textual insights from the query results.
  + FR7: The system shall allow users to execute direct SQL commands (including DML/DDL) using a /run prefix.
  + FR8: The system shall provide appropriate feedback or error messages for invalid inputs, query execution failures, or when a natural language query cannot be answered.
  + FR9: The system shall provide a conversational response if an SQL query cannot be generated from a natural language question based on the available schema.
  + FR10: The system shall include a script (gen-data.py) to create and populate two sample databases (SQLLLM, StoreDB) for demonstration.
* **Non-Functional Requirements:**
  + **Performance Requirement (NFR1):**
    - Schema fetching should complete within a few seconds.
    - LLM-based SQL generation and insight generation should ideally respond within 5-15 seconds, subject to API latency.
    - SQL query execution time depends on query complexity and database size but should be reasonable for typical ad-hoc queries. The system limits fetched rows to 100 for display.
  + **Dependability/Reliability Requirement (NFR2):**
    - The system should handle common errors gracefully (e.g., database connection issues, API errors, invalid SQL) and provide informative messages.
    - The backend should use connection pooling (get\_db\_connection hints at this) to manage database connections efficiently.
    - The system should be available during operational hours (for a deployed version).
  + **Maintainability Requirement (NFR3):**
    - The Python code should be modular (e.g., separate functions for DB interaction, LLM calls, schema fetching in sql\_assistant.py).
    - Configuration (DB credentials, API keys) must be externalized (using .env file).
    - Code should be well-commented.
    - Dependencies should be managed via requirements.txt.
  + **Security Requirement (NFR4):**
    - API keys must be stored securely and not hardcoded (handled by .env and .gitignore).
    - **SQL Injection Awareness:** The system executes LLM-generated SQL and user-provided SQL via /run. While the LLM is prompted to generate safe SQL, and direct execution is a feature, this presents a security risk. In a production environment, strict validation, sanitization, or an allow-list/block-list approach for SQL commands would be crucial. Database user permissions should be restricted (e.g., read-only for natural language queries if possible). The current project acknowledges this risk in code comments.
  + **Usability Requirement (Look and feel) (NFR5):**
    - The user interface shall be intuitive and easy to navigate.
    - Chat history, generated SQL, results, and insights should be clearly presented.
    - The schema display should be easy to understand.
    - Loading indicators should be used during long operations.
    - The interface should be responsive (Tailwind CSS helps).
* **Table 2.1: Hardware and Software Requirements**

|  |  |
| --- | --- |
| Category | Requirement |
| Hardware (Development/Standalone) |  |
| Processor | Standard Dual-Core processor or higher |
| RAM | 4GB or higher (8GB+ recommended) |
| Storage | 500MB free disk space (for project files, Python, MySQL) |
| Software (Development) |  |
| OS | Windows, macOS, or Linux |
| Python | Version 3.9+ (as implied by FastAPI and type hinting usage) |
| MySQL Server | Version 5.7+ or 8.0+ |
| Web Browser | Chrome, Firefox, Edge, Safari (modern, JavaScript-enabled) |
| Code Editor | VS Code, PyCharm, or any preferred IDE/text editor |
| Git | For version control |
| Software (Runtime/User) |  |
| Web Browser | Modern JavaScript-enabled browser |
| Internet Access | Required for LLM API communication and accessing the web application |
| Key Libraries | Listed in requirements.txt (FastAPI, Uvicorn, google-generativeai, mysql-connector-python, etc.) |

## 2.3 Validation

The system will be validated through various testing phases:

* **Natural Language Query Testing:**
* Feeding a diverse set of questions to the system targeting different tables, columns, and databases.
* Verifying the accuracy of the generated SQL.
* Checking if the results fetched match manual query execution.
* Testing queries that should result in the “Cannot answer” fallback.
* **Direct SQL Execution Testing:**
* Using the /run command with valid SELECT, SHOW, and (cautiously) DML/DDL statements (if DB permissions allow).
* Testing with syntactically incorrect SQL to verify error handling.
* **Schema Display Testing:** Ensuring all relevant databases, tables, and columns are displayed correctly.
* **Insight Generation Testing:** Evaluating the relevance and coherence of insights provided by the LLM.
* **Error Handling Testing:** Simulating DB connection failures, API errors, or providing malformed input to check system robustness.
* **User Interface Testing:** Ensuring all UI elements work as expected, the interface is responsive, and information is displayed clearly.

## 2.4 Expected Hurdles

* **LLM Accuracy and Consistency:** LLMs can sometimes generate incorrect or non-optimal SQL, or their responses might vary. Prompt engineering is key.
* **Handling Ambiguity:** Natural language is inherently ambiguous. The LLM might misinterpret complex or poorly phrased user questions.
* **API Rate Limits/Costs:** Heavy usage of the Gemini API might hit rate limits or incur significant costs if not managed.
* **Security of LLM-Generated Code:** Executing SQL generated by an LLM inherently carries risks if not properly sandboxed or validated.
* **Scalability:** For a large number of concurrent users, the performance of the LLM API and database connections needs to be considered. FastAPI’s asynchronous nature helps.
* **Schema Complexity:** Very large or complex database schemas might challenge the LLM’s ability to generate correct queries or fit within context window limits for the prompt.
* **Maintaining Context:** For multi-turn conversations, maintaining context effectively can be challenging with current LLM interaction patterns.

## 2.5 SDLC Model to be Used

An **Iterative and Incremental development model** (often associated with Agile methodologies) is most suitable for this project.

* **Iterative:** The system can be built in cycles, starting with core functionality (e.g., NL to SQL for one table) and gradually adding features (multi-database support, insights, UI improvements).
* **Incremental:** Each iteration delivers a working increment of the software.
* **Flexibility:** This approach allows for feedback incorporation and adaptation, which is crucial when working with rapidly evolving LLM technologies and complex NLP tasks. Development of prompts for the LLM often requires experimentation and refinement.

# Chapter 3: System Design

## 3.1 Design Approach

The DataFlow system employs a **hybrid design approach**:

* **Service-Oriented Architecture (SOA):** The backend is designed as a set of services exposed via a RESTful API (using FastAPI). The frontend consumes these services.
* **Modular Design:** \* Python code in sql\_assistant.py is organized into distinct functions for database connection, schema fetching, query execution, and LLM interaction. This promotes reusability and maintainability.
* The data generation logic is encapsulated in a separate gen-data.py script.
* **Object-Oriented Programming (OOP):** Pydantic models (ChatRequest) are used for data validation and serialization in the API, which is an OOP concept.
* **Functional Programming Principles:** Many helper functions in the backend operate as pure functions or with minimal side effects where possible, taking inputs and producing outputs.

## 3.2 Detail Design

* **Frontend (index.html):**
  + **Structure:** Single Page Application (SPA) feel, with dynamic content updates using JavaScript.
  + **Technologies:** HTML for structure, Tailwind CSS for styling, vanilla JavaScript for interactivity and API communication (Fetch API). Lucide icons for iconography, marked.js for rendering Markdown in chat messages.
  + **Key Components:**
    - Chat input field and send button.
    - Chat history display area (user and assistant messages).
    - Schema toggle button and display area (dynamically populated).
    - Refresh schema button.
  + **Functionality:**
    - Sends user messages (NLQ or /run commands) to the backend /chat endpoint.
    - Receives responses (generated SQL, results, insights, errors) and renders them appropriately in the chat history.
    - Handles UI state (e.g., loading spinners).
    - Fetches and displays database schema from the /schema endpoint.
* **Backend (sql\_assistant.py):**
  + **Framework:** FastAPI.
  + **API Endpoints:**
    - GET /: Serves the index.html page.
    - GET /schema: Fetches and returns the database schema.
    - POST /chat: Main endpoint for handling user messages, orchestrating LLM calls and database interactions.
  + **Core Modules/Functions:**
    - get\_db\_connection(db\_name: Optional[str]): Establishes MySQL connection (optionally to a specific DB). Manages connection pooling.
    - execute\_sql\_query(query: str): Executes SQL queries, handles SELECT vs. DML/DDL, fetches results and column metadata. Includes error handling and rollback.
    - fetch\_all\_tables\_and\_columns(): Discovers all non-system databases, their tables, and columns.
    - generate\_sql\_with\_gemini(user\_query: str, schema: Dict): Constructs a prompt with schema and user query, calls Gemini API to generate SQL.
    - get\_insights\_with\_gemini(...): Calls Gemini API to generate insights from query results.
    - get\_conversational\_response\_with\_gemini(user\_message: str): Calls Gemini API for a general conversational response as a fallback.
  + **Data Models:** ChatRequest (Pydantic model) for validating incoming chat messages.
  + **Configuration:** Uses python-dotenv to load database credentials and API keys from .env.
* **LLM Interaction (Google Gemini):**
  + The google-generativeai library is used.
  + Prompts are carefully engineered to include:
    - Role definition (e.g., “You are an expert SQL assistant”).
    - Database schema (for SQL generation).
    - User question.
    - Specific instructions on output format (e.g., “Generate only one single SQL statement”, “Format your response clearly using Markdown”).
  + Error responses from the LLM (e.g., “Error: Cannot answer…”) are handled.
* **Data Generation (gen-data.py):**
  + **Purpose:** To create sample databases (SQLLLM, StoreDB) and populate them with data for testing and demonstration.
  + **Libraries:** mysql.connector, Faker, python-dotenv, os, decimal.
  + **Functionality:**
    - Creates databases if they don’t exist.
    - Creates tables: employees and salaries in SQLLLM; products in StoreDB.
    - Generates realistic fake data (names, dates, departments, product details, prices) using Faker.
    - Inserts data into tables. Includes logic for ON DUPLICATE KEY UPDATE for employees and clearing/re-inserting for salaries and products for demo freshness.

## 3.3 System Design using various Structured Analysis and Design Tools

* **Figure 3.1: High-Level System Architecture Diagram for DataFlow** *(This would be a block diagram showing: User <-> Web Browser (Frontend UI) <-> FastAPI Backend (Python) <-> [Google Gemini API, MySQL Databases]. The backend interacts with both the LLM and the Databases.)*
* **Figure 3.2: Data Flow Diagram (DFD) for a User Query (Level 0/1)**
  + **Processes:**
    1. User Submits Query (via UI)
    2. Process User Input (Backend API)
    3. Generate SQL (LLM)
    4. Execute SQL (Database Interface)
    5. Generate Insights (LLM)
    6. Format & Display Response (UI)
  + **Data Stores:**
    - D1: MySQL Databases (SQLLLM, StoreDB)
    - D2: Database Schema (cached or fetched by backend)
  + **External Entities:**
    - E1: User
    - E2: Google Gemini API
  + **Flows:**
    - User Query -> P1
    - Chat Request -> P2
    - Schema + NL Query -> P3 (via E2)
    - Generated SQL -> P2 -> P4
    - Query for Execution -> D1
    - Raw Results -> P4 -> P2
    - Results for Insight -> P5 (via E2)
    - Insights -> P2
    - Formatted Response (SQL, Results, Insights) -> P6 -> E1
* **Data Dictionary (Key Structures):**
  + **ChatRequest:** message: string (user’s input)
  + **SchemaInformation:** Dict[db\_name: string, Dict[table\_name: string, List[column\_name: string]]]
  + **QueryResults:** List[Tuple[Any,...]] (list of rows, each row is a tuple)
  + **ColumnNames:** List[string]
  + **APIResponse (Chat):** type: string ("result", "info", "error"), query: Optional[string], columns: Optional[List[string]], results: Optional[List[Tuple]], insights: Optional[string], content: Optional[string]
* **Flowcharts/UML:**
  + A **Sequence Diagram** for POST /chat would be beneficial, showing interactions: User -> Frontend JS -> FastAPI Backend -> fetch\_schema -> generate\_sql\_with\_gemini -> Gemini API -> execute\_sql\_query -> MySQL -> get\_insights\_with\_gemini -> Gemini API -> Backend -> Frontend JS -> User.
  + An **Activity Diagram** for the handle\_chat function in sql\_assistant.py could illustrate the conditional logic (if /run, if NLQ, error handling, etc.).

**3.4 User Interface Design**

The UI is designed as a single-page web application with a focus on simplicity and interactivity, resembling a typical chat application.

* **Layout:**
  + Header: Displays application title (“DataFlow”) and a “View/Hide Schema” button.
  + Schema Display Area: A collapsible section below the header to show database schema details. Includes a “Refresh” button.
  + Chat History: The main area where user messages and assistant responses are displayed in chronological order. User messages are aligned to one side, assistant messages to the other, with distinct background colors.
  + Input Area: A text input field at the bottom for users to type messages, and a “Send” button.
  + **Styling:** Tailwind CSS is used for a modern, utility-first approach to styling, ensuring responsiveness.
* **Interactivity:**
  + JavaScript handles sending messages, receiving responses, and dynamically updating the chat history and schema display.
  + Generated SQL is shown in a <pre><code> block.
  + Query results are rendered as HTML tables within a scrollable div.
  + Insights and other informational messages from the assistant are rendered as Markdown (e.g., for formatted lists or code blocks).
  + Loading spinners indicate when the system is processing a request.
* **Key UI Elements from index.html:**
  + chat-history: Div for messages.
  + message-input: Text input for user.
  + chat-form: Form to submit messages.
  + schema-display, schema-content: For showing DB schema.
  + schema-toggle-btn, refresh-schema-btn: Buttons for schema interaction.

**3.5 Database Design**

The DataFlow system is designed to interact with existing MySQL databases and also includes a script (gen-data.py) to set up two sample databases for demonstration purposes: SQLLLM and StoreDB.

* **SQLLLM Database (Employee/Salary Information):**
  + **employees table:**
    - employee\_id: INT, PRIMARY KEY
    - first\_name: VARCHAR(50)
    - last\_name: VARCHAR(50)
    - date\_of\_birth: DATE
    - department: VARCHAR(50)
  + **salaries table:**
    - employee\_id: INT, FOREIGN KEY (references employees.employee\_id)
    - full\_name: VARCHAR(100) (Denormalized, combination of first\_name, last\_name from employees)
    - salary: INT
    - departments: VARCHAR(50) (Denormalized, same as employees.department)
* **StoreDB Database (Product Information):**
  + **products table:**
    - product\_id: INT, AUTO\_INCREMENT, PRIMARY KEY
    - product\_name: VARCHAR(100), NOT NULL
    - category: VARCHAR(50)
    - price: DECIMAL(10, 2)
    - stock\_quantity: INT
    - last\_updated: TIMESTAMP (DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP)
* **Table 3.1: Structure of employees table in SQLLLM database** (As above)
* **Table 3.2: Structure of salaries table in SQLLLM database** (As above)
* **Table 3.3: Structure of products table in StoreDB database** (As above)
* **3.5.1 ER Diagrams**
  + **Figure 3.3: ER Diagram for SQLLLM Database** *(This would show an Employees entity and a Salaries entity. A one-to-one relationship exists from Employees to Salaries based on employee\_id being PK in Employees and FK in Salaries. The gen-data.py script implies a 1:1 relationship as it generates one salary record per employee.)*
  + **Figure 3.4: ER Diagram for StoreDB Database** *(This would show a single Products entity with its attributes.)*
* **3.5.2 Normalization**
  + **SQLLLM.employees**: Appears to be in at least 3NF (assuming employee\_id is the sole determinant for all other attributes).
  + **SQLLLM.salaries**: Contains denormalized data (full\_name, departments) which are derived from the employees table. This is done for simplicity in the data generation script but means it’s not in 2NF or 3NF with respect to these derived attributes if employee\_id is the key. The primary purpose of these tables is to provide sample data for the LLM to query.
  + **StoreDB.products**: Appears to be in at least 3NF (assuming product\_id is the sole determinant).
  + The DataFlow application itself queries the existing structure; it does not perform normalization.
* **3.5.3 Database Manipulation**
  + **DataFlow Application (sql\_assistant.py):**
    - Primarily performs SELECT queries generated by the LLM.
    - Allows SHOW commands.
    - Through the /run command, it can execute other DML (INSERT, UPDATE, DELETE) and DDL (CREATE, ALTER, DROP) statements, subject to the database user’s permissions. Commits and rollbacks are handled.
  + **Data Generation Script (gen-data.py):**
    - Performs CREATE DATABASE, CREATE TABLE.
    - Performs INSERT operations to populate data.
    - Uses ON DUPLICATE KEY UPDATE for the employees table.
    - Uses TRUNCATE TABLE or DELETE for products and salaries before re-inserting data to ensure freshness in demos.
* **3.5.4 Database Connection Controls and Strings**
  + Connection parameters (host, user, password, database name) are managed externally via a .env file and loaded using python-dotenv.
  + The get\_db\_connection function in sql\_assistant.py handles the creation of mysql.connector connections.
  + It includes parameters for connection pooling (pool\_name, pool\_size), which is good practice for managing database connections efficiently in a web application.
  + The auth\_plugin='mysql\_native\_password' is explicitly set.

**3.6 Methodology**

The operational methodology of DataFlow for a natural language query is as follows:

1. **User Input:** The user types a natural language question into the chat interface (e.g., “Show me all products in the Electronics category”).

2. **Request to Backend:** The frontend JavaScript sends this message to the /chat endpoint of the FastAPI backend.

3. **Schema Fetching (if needed):** The backend calls fetch\_all\_tables\_and\_columns() to get the current schema of all user-accessible databases. This schema is crucial context for the LLM.

4. **SQL Generation:** The backend constructs a detailed prompt containing the user’s question and the fetched database schema. This prompt is sent to the Google Gemini API via generate\_sql\_with\_gemini.

5. **LLM Processes and Responds:** The Gemini API processes the prompt and returns a generated SQL query (e.g., SELECT \* FROM StoreDB.products WHERE category = 'Electronics').

6. **SQL Validation/Error Handling (LLM):** The backend checks if the LLM returned a valid-looking query or an error message (e.g., “Error: Cannot answer question…”). \* If a specific “cannot answer” error, it may fall back to get\_conversational\_response\_with\_gemini. \* If other error, it’s reported to the user.

7. **Query Pre-processing (Handling SHOW TABLES;):** If the query is a plain SHOW TABLES;, the backend checks if there’s a single user database to target and prepends a USE database\_name; statement if so. If multiple databases exist, it prompts the user for clarification.

8. **SQL Execution:** The (potentially modified) SQL query is executed against the MySQL database using execute\_sql\_query. This function handles fetching results and column names for SELECT/SHOW, or committing changes for DML/DDL.

9. **Database Error Handling:** If execute\_sql\_query encounters a database error, it’s caught and reported.

10. **Insight Generation (for SELECT/SHOW results):** If the query was successful and returned results, these results, along with the original query, generated SQL, and column types, are sent to the Gemini API via get\_insights\_with\_gemini to generate a textual analysis.

11. **Response to Frontend:** The backend constructs a JSON response containing the type of message (result, info, error), the generated SQL, column names, query results, and generated insights (if any).

12. **Display to User:** The frontend JavaScript parses this JSON response and updates the chat interface to display the information to the user in a structured way.

For /run commands, steps 3-6 are skipped, and the user-provided SQL is directly sent to step 7 (with the same pre-processing for plain SHOW TABLES;) and then to step 8 for execution.

# Chapter 4: Implementation, Testing, and Maintenance

## 4.1 Introduction to Languages, IDE’s, Tools and Technologies used for Implementation

* **Table 4.1: Key Technologies and Libraries Used**

|  |  |  |
| --- | --- | --- |
| Category | Technology/Tool/Library | Purpose |
| Programming Languages | Python 3.x | Backend logic, data generation, API interaction |
|  | HTML5 | Frontend structure |
|  | CSS3 (Tailwind CSS) | Frontend styling and layout |
|  | JavaScript (ES6+) | Frontend interactivity, API calls |
| Backend Framework | FastAPI | Building efficient and modern RESTful APIs in Python |
|  | Uvicorn | ASGI server for running FastAPI applications |
| Frontend Libraries/Tools | Tailwind CSS | Utility-first CSS framework for rapid UI development |
|  | Lucide Icons | SVG icon library |
|  | Marked.js | Markdown parser for rendering assistant messages |
| Database | MySQL | Relational database management system |
|  | mysql-connector-python | Python driver for connecting to MySQL databases |
| LLM API | Google Gemini API | Natural language understanding, SQL generation, insight generation |
|  | google-generativeai | Official Python SDK for Gemini API |
| Data Generation | Faker | Library for generating fake data |
| Configuration | python-dotenv | Loading environment variables from .env files |
| Version Control | Git | Distributed version control system |
| IDE (Developer Choice) | Visual Studio Code (implied by .vscode) | Code editor with Python, web dev support |
| Package Management | pip | Python package installer |

## 4.2 Coding Standards of Language Used \* Python (sql\_assistant.py, gen-data.py):

* Adherence to PEP 8 style guidelines for code layout, naming conventions, and comments is generally observed. \* Use of type hinting (e.g., query: str, -> Tuple[Optional[List[Tuple]], ...]) enhances code readability and maintainability.
* Modular design with functions for specific tasks.
* Meaningful variable and function names.
* Comments are used to explain complex logic or important sections (e.g., security warnings about SQL execution). \* Error handling using try-except blocks.
* Logging (logging module) is implemented in sql\_assistant.py for tracking events and errors.
* **HTML (index.html):**
  + Semantic HTML5 tags are used where appropriate.
  + Code is well-indented for readability.
* **JavaScript (index.html):**
  + Modern JavaScript (ES6+) syntax.
  + Functions are used to encapsulate logic (e.g., addMessageToChat, fetchSchema, sendMessage).
  + Comments explain function purposes and complex sections.
  + Variables are declared using const and let.
  + Asynchronous operations (async/await) are used for API calls.
* **General:**
  + Consistent naming conventions.
  + .gitignore file is used to exclude unnecessary files/directories from version control (e.g., .env, \_\_pycache\_\_, /SQLLLM directory which might be a temp data dir).

## 4.3 Project Scheduling

While a formal project scheduling tool (like PERT or GANTT charts) was not explicitly used for this academic project, the development likely followed logical phases:

1. **Phase 1: Conceptualization and Requirement Analysis (Estimate: 1 Week)**

* Defining project scope, objectives, and features.
* Researching LLM capabilities for SQL generation.
* Outlining basic system architecture.

2. **Phase 2: Backend Core Development (Estimate: 3-4 Weeks)**

* Setting up FastAPI application.
* Implementing database connection logic (get\_db\_connection, execute\_sql\_query).
* Developing schema fetching mechanism (fetch\_all\_tables\_and\_columns).

3. **Phase 3: LLM Integration (Estimate: 3-4 Weeks)**

* Integrating with Gemini API (generate\_sql\_with\_gemini, get\_insights\_with\_gemini).
* Extensive prompt engineering and testing for SQL generation accuracy.
* Implementing conversational fallback.

4. **Phase 4: Frontend Development (Estimate: 2-3 Weeks)**

* Designing and implementing the HTML structure.
* Styling with Tailwind CSS.
* Writing JavaScript for chat functionality, API communication, and dynamic updates.

5. **Phase 5: Data Generation Script (Estimate: 1 Week)**

* Developing gen-data.py to create and populate sample databases.

6. **Phase 6: Testing and Refinement (Estimate: 2 Weeks - Ongoing)**

* Unit testing individual functions.
* Integration testing of frontend, backend, LLM, and database.
* User acceptance testing (informal) to gather feedback.
* Debugging and refining prompts and logic.

7. **Phase 7: Documentation (Estimate: 1 Week)**

* Writing README.md.
* Preparing the project report.

This phased approach allows for iterative development and adjustments based on challenges encountered, especially with LLM integration.

## 4.4 Testing Techniques and Test Plans

* **Testing Techniques:**
  + **Manual Testing:** Core method used, involving direct interaction with the web interface, inputting various natural language queries and /run commands, and observing the output.
  + **Black Box Testing:** Testing the system based on its specifications without looking at the internal code structure (from a user’s perspective).
  + **Error Guessing:** Based on experience, anticipating potential error conditions (e.g., empty input, invalid SQL, ambiguous NLQ).
  + **Integration Testing (Informal):** Ensuring that the frontend, backend, LLM, and database components work together correctly. For example, verifying that a natural language query flows through the system to produce the correct SQL, which then fetches the correct data, which is then displayed with insights.
  + **Unit Testing (Conceptual):** While explicit unit test files are not provided, individual functions in sql\_assistant.py (like execute\_sql\_query with mock connections, or prompt generation logic) would be candidates for unit tests in a more extensive setup.
* **Test Plan & Sample Test Cases:**
  + **Table 4.2: Sample Test Cases for System Validation**

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case ID | Description | Expected Input | Expected Output |
| TC\_NLQ\_01 | Simple NLQ - All records from one table | “Show all employees” | Generated SQL: SELECT \* FROM SQLLLM.employees; (or similar), results table, insights |
| TC\_NLQ\_02 | NLQ with WHERE clause (targeting StoreDB) | “List products in Electronics category with price less than 100” | Correct SQL with WHERE clauses, filtered results, insights |
| TC\_NLQ\_03 | NLQ requiring JOIN (implicit) | “What are the salaries of employees in Engineering?” | SQL joining employees and salaries tables, correct results, insights |
| TC\_NLQ\_04 | NLQ that cannot be answered | “What is the weather like today?” | “Error: Cannot answer question with available schema.” or polite conversational fallback |
| TC\_SQL\_01 | Direct SQL - SELECT | /run SELECT product\_name, price FROM StoreDB.products WHERE category = 'Books'; | Results table showing product names and prices for books |
| TC\_SQL\_02 | Direct SQL - SHOW TABLES (single user DB context) | /run SHOW TABLES; (assuming SQLLLM or StoreDB is only user DB) | Prepends USE db\_name;, shows tables for that DB |
| TC\_SQL\_03 | Direct SQL - SHOW TABLES (multiple user DBs) | /run SHOW TABLES; (when both SQLLLM and StoreDB exist) | Error message asking to specify database or use SHOW TABLES FROM db\_name; |
| TC\_SQL\_04 | Direct SQL - Invalid Syntax | /run SELEC \* FROM employees; | SQL Error message from the database |
| TC\_UI\_01 | Schema Display Toggle | Click “View Schema” button, then “Hide Schema” | Schema panel appears and disappears correctly |
| TC\_UI\_02 | Schema Refresh | Click “Refresh” button in schema panel | Schema content updates, loading indicator shown |
| TC\_ERR\_01 | Empty message submission | Click “Send” with no input | No message sent, or a gentle prompt to type something |
| TC\_ERR\_02 | LLM API Key Invalid/Unavailable (Simulated) | Any NLQ | Error message indicating failure to communicate with AI model |
| TC\_ERR\_03 | DB Unavailable (Simulated by stopping MySQL service) | Any NLQ or SQL command | Error message indicating database connection failure |

# Chapter 5: Results and Discussions

## 5.1 User Interface Representation

The DataFlow application presents a clean, modern, and interactive user interface, designed to facilitate easy communication with the SQL LLM agent. *(Refer to* ***Figure 5.1: Main Chat Interface of DataFlow*** *- This would be a screenshot of the initial UI.)*

The UI is composed of several key areas:

1. **Header:** Displays the application title “DataFlow” and a button to toggle the visibility of the database schema (“View Schema” / “Hide Schema”).

2. **Schema Display Area:** When toggled, this section appears below the header. It lists the detected databases, their tables, and the columns within each table. A “Refresh” button allows users to update this schema view. *(Refer to* ***Figure 5.4: Database Schema Display Panel****)*

3. **Chat History:** This is the primary interaction area. It displays a chronological log of messages:

* **User Messages:** Questions asked by the user or /run commands, typically aligned to the right and styled with a distinct background (e.g., bg-indigo-100).
* **Assistant Responses:** Replies from the DataFlow agent, aligned to the left. These can include:
  + Generated SQL queries (displayed in a <pre><code> block).
  + Query results (rendered as an HTML table).
  + Insights (formatted text, potentially using Markdown).
  + Informational messages or error messages.

4. **Input Area:** Located at the bottom, this consists of:

* A text input field where users type their natural language questions or /run SQL commands.
* A “Send” button to submit the message. A loading spinner is shown on the button during processing.

The interface uses Tailwind CSS for styling, providing a responsive design that adapts to different screen sizes. Lucide icons enhance visual cues, and marked.js ensures that insights or messages containing Markdown are rendered correctly.

**5.1.1 Brief Description of Various Modules of the System (from a User/Results Perspective)**

* **Natural Language Query Module:**

This is the core interaction module. The user types a question in plain English. The system processes this, generates an SQL query, executes it, and displays the SQL, results, and insights. *(Refer to* ***Figure 5.2: Example of Natural Language Query and System Response****)*

* **Direct SQL Execution Module:** Activated when the user prefixes their input with /run. This module bypasses the NL-to-SQL generation and directly executes the provided SQL statement. Results (for SELECT/SHOW) or a success/failure message are displayed. *(Refer to* ***Figure 5.3: Example of Direct SQL Execution using /run command****)*
* **Schema Viewer Module:** Provides transparency by allowing users to see the structure of the databases the agent is interacting with. This helps users formulate more effective questions and understand the data’s context.
* **Results Display Module:** Presents data retrieved from SELECT or SHOW queries in a clear, scrollable HTML table format. Column headers are derived from the query.
* **Insights Generation Module:** After displaying query results, this module presents textual analysis or summaries generated by the LLM, aiming to provide context or highlight interesting aspects of the data.
* **Error Reporting Module:** If issues occur (e.g., invalid SQL, database error, LLM API problem, ambiguous query), the system displays an informative error message in the chat interface. *(Refer to* ***Figure 5.5: Example of an Error Message Display****)*

**5.2 Snapshots of System with Brief Detail of Each** *(As these are visual, descriptive placeholders are provided. You would insert actual screenshots in your report.)*

* **Figure 5.1: Main Chat Interface of DataFlow**
  + *Detail:* Shows the initial screen with the header, empty chat history (or a welcome message), and the message input field. The schema panel would typically be hidden initially.
* **Figure 5.2: Example of Natural Language Query and System Response**
  + *Detail:* Shows the chat history with:
    1. A user’s natural language question (e.g., “Show employees in the Engineering department”).
    2. The assistant’s response, including:
       - The generated SQL query.
       - A table displaying the filtered employee records.
       - Textual insights about the result set.
* **Figure 5.3: Example of Direct SQL Execution using /run command**
  + *Detail:* Shows the chat history with:
    1. A user’s command (e.g., /run SELECT product\_name, price FROM StoreDB.products LIMIT 5;).
    2. The assistant’s response, including:
       - Confirmation of the query executed (same as input).
       - A table displaying the product names and prices.
       - Potentially insights, if configured for /run commands too.
* **Figure 5.4: Database Schema Display Panel**
  + *Detail:* Shows the schema panel expanded, listing databases like SQLLLM and StoreDB, with their respective tables (employees, salaries, products) and columns.
* **Figure 5.5: Example of an Error Message Display**
  + *Detail:* Shows the chat history where the user might have entered an invalid SQL query via /run, or an ambiguous natural language query. The assistant’s response is an error message explaining the issue (e.g., “SQL Error: You have an error in your SQL syntax…” or “Error: Cannot answer question with available schema.”).

**5.3 Back Ends Representation (Database to be Used)**

The DataFlow system is designed to primarily interact with **MySQL** databases. The backend Python application (sql\_assistant.py) uses the mysql-connector-python library to establish connections and execute queries.

For demonstration and development, the project includes a script gen-data.py that sets up and populates two distinct sample MySQL databases:

1. **SQLLLM:** Contains tables related to employees and their salaries.

2. **StoreDB:** Contains a table for product information.

The system can discover and interact with any user-accessible non-system MySQL databases on the configured host.

**5.3.1 Snapshots of Database Tables with Brief Description** *(These would ideally be screenshots from a database client like MySQL Workbench, or text representations of DESCRIBE table\_name;.)*

* **Snapshot of SQLLLM.employees Table Structure:**
  + *Description:* Stores basic information about employees.
  + *Columns:* employee\_id (INT, PK), first\_name (VARCHAR), last\_name (VARCHAR), date\_of\_birth (DATE), department (VARCHAR). *(Example textual representation)*
* +-----------------+-------------+  
  | Field | Type |  
  +-----------------+-------------+  
  | employee\_id | int |  
  | first\_name | varchar(50) |  
  | last\_name | varchar(50) |  
  | date\_of\_birth | date |  
  | department | varchar(50) |  
  +-----------------+-------------+
* **Snapshot of SQLLLM.salaries Table Structure:**
  + *Description:* Stores salary details for employees, linked by employee\_id.
  + *Columns:* employee\_id (INT, FK), full\_name (VARCHAR), salary (INT), departments (VARCHAR).

+-------------+--------------+

* | Field | Type |  
  +-------------+--------------+  
  | employee\_id | int |  
  | full\_name | varchar(100) |  
  | salary | int |  
  | departments | varchar(50) |  
  +-------------+--------------+
* **Snapshot of StoreDB.products Table Structure:**
  + *Description:* Stores information about products available in a store.
  + *Columns:* product\_id (INT, PK, AI), product\_name (VARCHAR), category (VARCHAR), price (DECIMAL), stock\_quantity (INT), last\_updated (TIMESTAMP). *(Example textual representation)*
* +------------------+---------------+  
  | Field | Type |  
  +------------------+---------------+  
  | product\_id | int |  
  | product\_name | varchar(100) |  
  | category | varchar(50) |  
  | price | decimal(10,2) |  
  | stock\_quantity | int |  
  | last\_updated | timestamp |  
  +------------------+---------------+

# Chapter 6: Conclusion and Future Scope

## Conclusion

The DataFlow project successfully demonstrates the potential of integrating Large Language Models with SQL databases to create an intelligent and user-friendly data interaction system. By translating natural language queries into SQL, executing them, and providing insightful summaries, DataFlow significantly lowers the barrier for non-technical users to access and understand database information. The interactive chat interface, coupled with schema awareness and the ability to handle direct SQL commands, provides a flexible and powerful tool.

The use of Python with FastAPI for the backend, standard web technologies for the frontend, and the Google Gemini API for AI capabilities has resulted in a robust and responsive application. The inclusion of a data generation script for sample databases (SQLLLM and StoreDB) further enhances its utility for demonstration and testing. The project achieves its primary objectives of simplifying database querying and making data more accessible. It serves as a strong foundation for a new paradigm of human-data interaction.

## Future Scope

While DataFlow provides a comprehensive set of features, there are several avenues for future enhancement and development:

1. **Broader Database Support:** Extend compatibility beyond MySQL to other popular SQL databases like PostgreSQL, SQL Server, and SQLite. This would involve handling dialect-specific SQL generation.

2. **Enhanced Security Measures:**

* Implement robust input sanitization and validation for /run commands to mitigate SQL injection risks more comprehensively.
* Explore query allow-listing or finer-grained permission controls based on user roles.
* Integrate user authentication and authorization to control access to specific databases or tables.

3. **Advanced NLP and Contextual Understanding:**

* Improve handling of highly complex or ambiguous natural language queries.
* Implement conversational context awareness, allowing the LLM to remember previous turns in the conversation for follow-up questions.
* Support for more complex analytical queries (e.g., window functions, CTEs) through natural language.

4. **Data Visualization:** Integrate basic charting or visualization capabilities to represent query results graphically, in addition to tabular data.

5. **User Customization and Saved Queries:** Allow users to save frequently used queries or customize LLM behavior (e.g., preferred insight style).

6. **Fine-Tuning LLM:** For specific domains or databases, explore fine-tuning the LLM on relevant schema and query patterns to improve SQL generation accuracy and relevance of insights.

7. **Performance Optimization:** For very large databases or high traffic, further optimize query execution, schema fetching, and LLM API interactions.

8. **Expanded DML/DDL through Natural Language:** Cautiously explore enabling data modification (INSERT, UPDATE, DELETE) through natural language, with strong confirmation steps and safeguards.

9. **Integration with Other Tools:** Provide APIs for DataFlow to be integrated into other business applications or BI dashboards.

10. **Improved Error Granularity and Suggestions:** Offer more specific suggestions to users when queries fail or when the LLM cannot interpret a request.

These potential enhancements could further elevate DataFlow into an even more powerful and indispensable tool for data interaction and analysis.

# References/Bibliography

* **Python:** Official Python Website. (2024). Retrieved from <https://www.python.org>
* **FastAPI:** Tiangolo, S. (2024). FastAPI - A Python framework for building APIs. Retrieved from <https://fastapi.tiangolo.com/>
* **MySQL:** Oracle Corporation. (2024). MySQL. Retrieved from <https://www.mysql.com/>
* **Google Gemini API:** Google. (2024). Gemini API. Google AI for Developers. Retrieved from <https://ai.google.dev/>
* **Faker Library:** Faker (Python Package Index). Retrieved from <https://pypi.org/project/Faker/> (Documentation: <https://faker.readthedocs.io/>)
* **Tailwind CSS:** Tailwind Labs Inc. (2024). Tailwind CSS - A utility-first CSS framework. Retrieved from <https://tailwindcss.com/>
* **MySQL Connector/Python:** Oracle Corporation. (2024). MySQL Connector/Python Developer Guide. Retrieved from <https://dev.mysql.com/doc/connector-python/en/>
* **Marked.js:** Marked.js Organization. (2024). A markdown parser and compiler. Built for speed. Retrieved from <https://marked.js.org/>
* **Lucide Icons:** Lucide Contributors. (2024). Beautiful & consistent icon toolkit. Retrieved from <https://lucide.dev/>

# Appendix

## Appendix A: Development Environment

* **Operating System:** Windows 11, Ubuntu 22.04 LTS
* **Programming Language Versions:**
  + Python: 3.11
* **Database Server:**
  + MySQL
* **Key Software Tools:**
  + Code Editor/IDE: Visual Studio Code
  + Web Browser: Google Chrome
  + Version Control: Git
  + Terminal/Command Prompt: Windows Terminal, Bash
* **Key Python Libraries (from requirements.txt):**
  + fastapi==0.111.0
  + uvicorn==0.30.1
  + google-generativeai (Version as installed)
  + mysql-connector-python==9.0.0
  + pydantic==2.7.4
  + python-dotenv (Version as installed)
  + Faker==28.4.1
  + Jinja2==3.1.4 (though not directly used in sql\_assistant.py for templates, FastAPI might use it or it’s a transitive dependency)

## Appendix B: Key Code Snippets

* **Snippet 1: FastAPI /chat endpoint structure (from sql\_assistant.py)**
* # sql\_assistant.py (Simplified excerpt)  
  @app.post("/chat", response\_class=JSONResponse)  
  async def handle\_chat(chat\_request: ChatRequest):  
   user\_message = chat\_request.message.strip()  
   response\_data: Dict[str, Any] = {"type": "error", "content": "An unexpected error occurred."}  
   query\_to\_run = None  
   schema = None  
    
   try:  
   if user\_message.lower().startswith("/run "):  
   query\_to\_run = user\_message[5:].strip()  
   # ... (validation for empty query) ...  
   else:  
   logger.info(f"Processing natural language query: {user\_message}")  
   schema = fetch\_all\_tables\_and\_columns()  
   # ... (error handling for schema fetch) ...  
    
   generated\_sql = generate\_sql\_with\_gemini(user\_message, schema)  
    
   # ... (handle specific LLM errors, fallback to conversational) ...  
   # ... (handle other LLM generation errors) ...  
   query\_to\_run = generated\_sql  
    
   # ... (logic for 'SHOW TABLES;' modification) ...  
    
   if query\_to\_run:  
   logger.info(f"Executing final query: {query\_to\_run}")  
   results, columns, col\_types, status, db\_error = execute\_sql\_query(query\_to\_run)  
    
   if status == 3: # SQL Error  
   # ... (format error response\_data) ...  
   elif status == 2: # DML/DDL Success  
   # ... (format success info response\_data) ...  
   elif status == 1: # SELECT/SHOW Success  
   insights = ""  
   if results is not None and columns and col\_types:  
   # ... (get original user intent for insights prompt) ...  
   insights = get\_insights\_with\_gemini(...)  
   response\_data = {"type": "result", "query": query\_to\_run, ...}  
   # ... (other status handling) ...  
   # ... (fallback response\_data if no query\_to\_run) ...  
   return JSONResponse(content=jsonable\_encoder(response\_data))  
   # ... (exception handling: HTTPException, general Exception) ...
* **Snippet 2: Gemini API call for SQL Generation (Conceptual from sql\_assistant.py)**
* # sql\_assistant.py (Simplified excerpt from generate\_sql\_with\_gemini)  
  def generate\_sql\_with\_gemini(user\_query: str, schema: Dict[str, Dict[str, List[str]]]) -> Optional[str]:  
   schema\_string = "..." # Format schema into string  
   prompt = f"""You are an expert SQL assistant...  
  Database Schema:  
  {schema\_string}  
  User Question: "{user\_query}"  
  ... (Instructions) ...  
  SQL Query:"""  
    
   try:  
   model = genai.GenerativeModel(GEMINI\_MODEL\_NAME)  
   response = model.generate\_content(prompt)  
   sql\_query = response.text.strip()  
   # ... (cleanup and basic validation) ...  
   return sql\_query  
   except Exception as e:  
   logger.error(f"Error calling Gemini API for SQL generation: {e}", exc\_info=True)  
   return "Error: Failed to communicate with the AI model..."
* **Snippet 3: Table creation in gen-data.py (Example: products table)** python # gen-data.py (Excerpt from create\_store\_tables\_if\_not\_exist) create\_products\_table\_query = f""" CREATE TABLE IF NOT EXISTS `{MYSQL\_DATABASE\_STORE}`.products ( product\_id INT AUTO\_INCREMENT PRIMARY KEY, product\_name VARCHAR(100) NOT NULL, category VARCHAR(50), price DECIMAL(10, 2), stock\_quantity INT, last\_updated TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP ) """ cursor.execute(create\_products\_table\_query)