A MAJOR PROJECT REPORT ON

DataFlow: An SQL LLM Agent for Intelligent Database Interaction

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE FINAL YEAR OF

BACHELOR OF TECHNOLOGY

(Computer Science & Engineering)

Submitted By:

Utkarsh Prajapati (2101660100061)

Anish Chauhan (2101660100013)

Adarsh (2101660100007)

Anshil Gautam (2101660100017)

Under the guidance of

Sh. Sri Nath Dwivedi

Head of Department

Department of Computer Science & Engineering

Dr. Ambedkar Institute of Technology for Divyangjan (AITD)

Certificate of Approval

This is certified that Utkarsh Prajapati (2101660100061), Anish Chauhan (2101660100013), Adarsh (2101660100007), and Anshil Gautam (2101660100017) have successfully completed the major project entitled: **"DataFlow: An SQL LLM Agent for Intelligent Database Interaction"** under the able guidance of **Sh. Sri Nath Dwivedi** toward the fulfillment of the final year course in Computer Science & Engineering.

Internal Examiner

External Examiner

Project Head (HoD)

Sh. Sri Nath Dwivedi

Candidate Declaration

We hereby certify that the work which is being presented in the report, entitled **"DataFlow: An SQL LLM Agent for Intelligent Database Interaction"**, in the partial fulfillment of the requirement for the final year of Bachelor in Technology (Computer Science & Engineering) and submitted to Dr. Ambedkar Institute of Technology for Divyangjan (AITD), Kanpur, is an authentic record of our own work carried out during the period January 2025 to June, 2025 under the supervision of Sh. Sri Nath Dwivedi, Head of Department, Computer Science & Engineering.

The matter presented in this report has not been submitted elsewhere for the award of any other degree from any Institutions.

Date Signature of Candidate

1. Utkarsh Prajapati (2101660100061)

2. Anish Chauhan (2101660100013)

3. Adarsh (2101660100007)

4. Anshil Gautam (2101660100017)

This is to certify that the above statement made by the candidate is correct to the best of our knowledge.

Date: Signature of Project Guide

Acknowledgements

We are highly grateful to the Director/Head of 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. Sri 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. Sri 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)

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.

Table of Contents

Chapter 1: Introduction

1.1 Objective

The primary objectives of the DataFlow project are:

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

1.2 Project Overview

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 Lan- guage 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 of- fers 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.

1.3 Benefits

The DataFlow project offers several benefits by addressing the identified need for democratized and simplified data access:

- **Enhanced Accessibility:** Empowers non-technical users to retrieve and understand database information without requiring SQL knowledge.
- **Increased Efficiency:** Automates the tedious and error-prone task of manual SQL query writing, saving time for both technical and non-technical users.
- **Improved Decision-Making:** By providing not just raw data but also LLM- generated insights, the system helps users derive valuable conclusions more quickly.
- **Reduced Learning Curve:** The intuitive natural language interface minimizes the training required to interact with complex databases.
- **Dynamic Adaptability:** The system's ability to dynamically fetch and use schema information allows it to work with various database structures without extensive pre-configuration for each.
- **Flexibility:** Caters to different user needs by supporting both natural language queries and direct SQL execution for advanced users.

1.4 Scope of Project

The scope of the DataFlow project encompasses the following key areas:

- Development of a web-based chat interface for user interaction.
- Implementation of a Python FastAPI backend to manage application logic.
- Integration with the Google Gemini API for natural language understanding, SQL generation, and insight generation.
- Connectivity with MySQL databases for schema fetching and query execution.
- Dynamic schema discovery for multiple non-system databases.

- Translation of natural language questions into executable SQL queries.
- Execution of generated SQL queries and retrieval of results.
- Presentation of query results in a tabular format.
- Generation and display of textual insights based on query results.
- Provision of a direct SQL execution facility (e.g., via /run command).
- Creation of a data generation script (gen-data.py) to populate two sample databases (SQLLLM and StoreDB) for demonstration and testing purposes.
- Handling of basic conversational fallback when SQL generation is not possible.

The project focuses on SELECT queries for natural language interaction but allows broader SQL command execution via the direct run facility, with associated security considerations noted.

1.5 Development Methodology / Development Theory

1.5.1 SDLC Model Used

- An Iterative and Incremental development model, closely aligned with Agile principles, was deemed most suitable for the DataFlow project. This choice was driven by the nature of the project, particularly its reliance on emerging LLM technologies and the need for flexibility in refining natural language processing components.
- Iterative Nature: The system was developed in cycles. Core functionalities, such as basic natural language to SQL translation for a single table, were implemented first. Subsequent iterations added more complex features like multi- database schema awareness, insight generation, direct SQL execution, and UI enhancements.
- Incremental Delivery: Each iteration aimed to produce a working increment of the software, allowing for early feedback and continuous testing of integrated components.
- Adaptability and Refinement: This model provided the flexibility to incorporate feedback and make adjustments, especially crucial for prompt engineering with the LLM, which often requires experimentation and iterative refinement to achieve desired accuracy and robustness.

1.5.2 Project Phases (Scheduling Overview)

While a formal Gantt or PERT chart was not strictly implemented for this academic project, the development followed a logical sequence of phases, with estimations for each:

1. Phase 1: Conceptualization and Requirement Analysis (Estimated Duration: 1 Week)
    - Defining the project's scope, objectives, and key features.

- Researching the capabilities of LLMs (specifically Google Gemini) for SQL generation and natural language understanding.
- Outlining the basic system architecture and technology stack.

2. Phase 2: Backend Core Development (Estimated Duration: 3-4 Weeks)
    - Setting up the FastAPI application framework.
    - Implementing core database connection logic, including functions for establishing connections (get_db_connection) and executing queries (execute_sql_query)
    - Developing the mechanism for fetching and processing database schema information (fetch_all_tables_and_columns).
3. Phase 3: LLM Integration (Estimated Duration: 3-4 Weeks)
    - Integrating the backend with the Google Gemini API using the google-generativeai library.
    - Developing functions for SQL generation (generate_sql_with_gemini) and insight generation (get_insights_with_gemini).
    - Extensive prompt engineering, testing, and refinement to improve the accuracy of SQL generation and the relevance of insights.
    - Implementing logic for conversational fallback responses.
4. Phase 4: Frontend Development (Estimated Duration: 2-3 Weeks)
    - Designing and implementing the HTML structure for the chat interface.
    - Styling the interface using Tailwind CSS for a responsive and modern look.
    - Writing JavaScript code for handling user input, API communication with the backend, and dynamically updating the chat history, schema display, and results.
5. Phase 5: Data Generation Script Development (Estimated Duration: 1 Week)
    - Creating the gen-data.py script using the Faker library to generate sample data.
    - Implementing logic to create and populate the SQLLLM and StoreDB

databases for demonstration and testing.

6. Phase 6: Testing and Refinement (Estimated Duration: 2 Weeks - Iterative and Ongoing)
    - Conducting unit testing for individual backend functions where feasible.

- Performing integration testing to ensure seamless interaction between the frontend, backend, LLM, and database.
- Informal user acceptance testing to gather feedback on usability and functionality.
- Debugging issues, refining LLM prompts, and optimizing application logic based on test results and feedback.

7. Phase 7: Documentation (Estimated Duration: 1 Week)
    - Writing the project report, including detailed descriptions of the system architecture, design, implementation, and results.
    - Preparing supporting documentation like README files.

This phased approach allowed for a structured development process while accommodating the iterative nature of working with AI components.

1.5.3 Gantt Chart / PERT Chart

A formal Gantt chart or PERT chart was not utilized for this project. However, the project scheduling outlined in the "Project Phases" section above reflects the planned timeline and dependencies between major development activities.

1.5.4 Coding Standards Overview

The project adhered to common coding standards to ensure readability, maintainability, and consistency:

6.1 Python (sql_assistant.py, gen-data.py):

- PEP-8 style guidelines were followed for code layout, naming conventions.
- Code was organized into modular functions and classes with clear responsibilities.
- Meaningful names were chosen for variables, functions, and classes.
- Docstrings and inline comments were used to explain complex logic, function purposes, and important considerations (e.g., security warnings).
- Error handling was implemented using ‘try-except’ blocks to manage potential exceptions gracefully.
- The ‘logging’ module was used in sql_assistant.py for recording application events and errors.

6.2 HTML (index.html):

- Semantic HTML5 tags were used where appropriate to define the structure of the content.
- Code was well-indented and formatted for readability.

6.3 JavaScript (within index.html):

- Modern JavaScript (ES6+) syntax was employed.
- Logic was encapsulated within functions (e.g. addMessageToChat, ‘fetch-Schema‘, ‘sendMessage‘).
- Comments were used to explain the purpose of functions and complex code sections.
- Variables were declared using ‘const’ and ‘let‘ appropriately.
- Asynchronous operations for API calls were handled using ‘async/await’.

6.4 General:

- Consistent naming conventions were maintained across different parts of the project.
- A ‘.gitignore’ file was used to exclude unnecessary files and directories (e.g., ‘.env‘, ‘pycache ,virtual environment folders) from version control.

1.6 Report Layout

This project report is structured into several chapters to provide a comprehensive overview of the DataFlow system.

- Chapter 1 (Introduction): Introduces the project, its objectives, scope, the prob- lem it addresses, and the proposed solution. It also covers the development methodology.
- Chapter 2 (Requirement Analysis): Details the feasibility study (technical, economical, operational), technical specifications including hardware and software requirements, and descriptions of the technologies used.
- Chapter 3 (Project Design): Describes the design methodology, including the Software Requirement Specification (SRS), Entity-Relationship (ER) Diagrams, Data Flow Diagrams (DFDs), and other relevant flow diagrams or illustrations like the system architecture.
- Chapter 4 (Testing): Outlines the testing strategies, techniques, and sample test cases used to validate the system’s functionality and robustness.

- Chapter 5 (User Interface): Presents snapshots of the system's user interface with brief descriptions.
- Chapter 6 (Conclusion & Future Work): Summarizes the project's achievements and discusses potential areas for future development and enhancement.

The report also includes preliminary pages such as the Certificate of Approval, Candidate Declaration, Acknowledgements, Abstract, Table of Contents, List of Figures, and List of Tables, followed by a Bibliography and Appendices, adhering to the specified formatting guidelines.

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:** Development requires knowledge of Python, web development fundamentals, SQL, and API integration, which are common skills in software development.
- **Economic 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 it 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 Technical Specification

2.2.1 Hardware Requirement

Table 1: *Hardware Requirements*

Component	Minimum Specification (Development/Standalone)
Processor	Standard Dual-Core processor or higher (e.g., Intel i3/AMD Ryzen 3 or newer
RAM	4GB or higher (8GB+ recommended for smoother development)
Storage	At least 500MB free disk space (for project files, Python, MySQL, and dependencies)
Network	Internet connection (for LLM API access and package downloads)

2.2.2 Software Requirement

Table 2: *Software Requirements*

Category	Requirement
Development Environment	
Operating System	Windows (10/11), macOS (Catalina+), or Linux (e.g., Ubuntu 20.04+)
Python Version	3.9 or higher
MySQL Server	Version 5.7+ or 8.0+

Web Browser (for testing)	Chrome, Firefox, Edge, or Safari (JavaScript-enabled)
Code Editor / IDE	Visual Studio Code, PyCharm, or any Python-aware editor
Version Control	Git
Package Manager	pip (Python package installer)
Runtime Environment (User-Side)	
Web Browser	JavaScript-enabled browser (Chrome, Firefox, Edge, Safari)
Internet Access	Required for LLM API and web app access
Key Python Libraries & Frameworks	*(managed via requirements.txt)*
Web Framework	FastAPI, Uvicorn (ASGI server)
Database Connector	mysql-connector-python
LLM SDK	google-generativeai
Data Generation	Faker
Environment Management	python-dotenv
Data Validation	Pydantic
HTTP Client	httpx
Frontend Rendering	HTML, Tailwind CSS, JavaScript, Marked.js, Lucide Icons (served by FastAPI)

2.3 Technology Descriptions

The DataFlow project leverages a combination of modern programming languages, frameworks, and APIs to achieve its functionality.

Table 3: *Key Technologies and Libraries Used in DataFlow*

Technology/Tool	Description and Purpose
Python 3.x	Primary backend language; chosen for readability, vast libraries, and web/AI compatibility.
FastAPI	High-performance Python web framework for REST APIs; uses Pydantic for data validation.
Uvicorn	ASGI server to run FastAPI apps; known for being lightweight and fast.
MySQL	Relational database system; stores and retrieves application data.
mysql-connector-python	Official Oracle Python driver for MySQL; handles query execution and schema access.
Google Gemini API	LLM by Google; translates natural language to SQL, generates insights, and fallback replies.
google-generativeai	Python SDK to simplify interactions with Gemini API.
HTML5	Standard markup language to structure the web interface.
Tailwind CSS	Utility-first CSS framework for custom and rapid frontend styling.
JavaScript (ES6+)	Enables dynamic frontend behavior—messaging, UI updates, state management.
Marked.js	JS library to parse and render Markdown from LLM into HTML.
Lucide Icons	Open-source SVG icon pack to enhance UI with visuals.
Faker	Generates realistic fake data for database seeding in gen-data.py.
python-dotenv	Loads environment variables securely from .env files.
Pydantic	Validates data using type hints; used by FastAPI for models.
Git	Distributed version control for source code tracking, collaboration, and history.

Chapter 3: Project Design (Drawing/Blueprint) Methodology

3.1 Software Requirement Specification (SRS)

The Software Requirement Specification (SRS) outlines the functional, non-functional, and data requirements of the DataFlow system. It serves as a foundational document for guiding the design and development process.

3.4.1 Data Requirements

- **User Input:** The system must accept natural language questions and direct SQL commands (prefixed with /run) from the user via the chat interface.
- **Database Schema Information:** The system needs to access and process information about the connected MySQL databases, including database names, table names, column names, and column data types. This is essential for context-aware SQL generation.
- **Database Content:** The system interacts with the actual data stored within the user-specified tables in the MySQL databases to retrieve information based on queries.
- **Configuration Data:** Secure storage and retrieval of database connection credentials (host, user, password) and the Google Gemini API key are required. This is managed via an .envfile.
- **LLM Responses:** The system processes various types of responses from the LLM, including generated SQL queries, textual insights, and conversational replies.
- **Query Results:** Data fetched from the database (rows and columns) needs to be handled and formatted for display and for generating insights.

3.4.1 Functional Requirements

- **FR1 (Natural Language Query Input):** The system shall provide a chat inter- face for users to submit queries in natural language (English).
- **FR2 (Schema Fetching and Display):** The system shall automatically fetch the schema (databases, tables, columns) of all accessible non-system MySQL databases and display it to the user.
- **FR3 (NL to SQL Translation):** The system shall utilize the Google Gemini API to translate valid natural language user queries into syntactically correct and semantically appropriate SQL queries, considering the fetched database schema.
- **FR4 (SQL Query Execution):** The system shall execute the generated SQL queries (primarily SELECT for NLQ, but also other types via /run) against the target MySQL database.
- **FR5 (Results Display):** The system shall display the results of successful SELECT or SHOW queries in a clear, tabular format within the chat interface. For DML/DDL commands via /run, a success or failure message shall be displayed.

- **FR6 (Insight Generation):** For successful SELECT queries, the system shall use the Google Gemini API to generate and display textual insights or summaries based on the retrieved data.
- **FR7 (Direct SQL Execution):** The system shall allow users to execute arbitrary SQL commands directly by prefixing their input with /run. This includes DML and DDL commands, subject to database user permissions and a confirmation step for modifying queries.
- **FR8 (Error Handling and Feedback):** The system shall provide clear and informative feedback or error messages to the user in cases of invalid input, SQL generation failure, query execution errors, API communication issues, or when a natural language query cannot be reasonably answered.
- **FR9 (Conversational Fallback):** If the LLM determines that a natural language question cannot be translated into an SQL query based on the available schema, the system shall attempt to provide a polite conversational response.
- **FR10 (Sample Data Generation):** The system shall include an auxiliary script (gen-data.py) capable of creating and populating two distinct sample MySQL databases (SQLLLM and StoreDB) with realistic data for demonstration and testing purposes.
- **FR11 (Modifying Query Confirmation):** For SQL queries (whether LLM-generated or from /run if LLM generates it) identified as potentially modifying data (INSERT, UPDATE, DELETE, DDL), the system shall prompt the user for explicit confirmation before execution.
- **FR12 (Configuration Management):** The system shall allow users to configure database connection parameters and the Gemini API key through the UI, which updates a .env file and applies settings dynamically.

3.4.1 Non-Functional Requirements

- **NFR1 (Performance):**
    - Schema fetching: Should ideally complete within 2-5 seconds for a moder- ate number of databases/tables.
    - LLM-based SQL generation and insight generation: Response times should ideally be within 5-15 seconds, understanding that this is subject to external API latency and complexity.
    - SQL query execution: Time will vary based on query complexity and database size. For typical ad-hoc queries on sample data, it should be near-instantaneous. The system limits displayed rows (e.g., to 100) to manage frontend rendering performance.
- **NFR2 (Reliability & Availability):**
    - The system should handle common errors gracefully (e.g., database connection timeouts, API errors, malformed SQL) and provide informative messages rather than crashing.

- The backend should manage database connections efficiently, using connection pooling as implemented by mysql-connector-python if configured or managed implicitly.
- For a deployed version, availability would depend on the hosting infrastructure. The application itself should be robust to stay running.

- **NFR3 (Maintainability):**
  - The Python backend code (sql_assistant.py) is structured modularly with distinct functions for database interaction, LLM communication, schema processing, etc.
  - Configuration (database credentials, API keys) is externalized using a .env file, promoting separation of concerns.
  - Code includes comments to explain complex logic and critical sections.
  - Dependencies are managed via requirements.txt for easy environment replication.
  - Logging is implemented to trace operations and assist in debugging.

- **NFR4 (Security):**
  - API keys and database credentials must not be hardcoded and are managed through the .env file, which is included in .gitignore.
  - SQL Injection Mitigation (Critical Concern): While the LLM is prompted to generate safe SQL, executing LLM-generated SQL or user-provided SQL via /run inherently carries risks. The system implements a confirmation step for queries identified as data-modifying. For a production environment, more stringent measures like strict input validation, query sanitization, an allow-list/block-list for SQL commands, and principle of least privilege for database users would be paramount. The current academic project acknowledges this risk.

- **NFR5 (Usability - Look and Feel):**
  - The user interface shall be intuitive, clean, and easy to navigate, resembling a familiar chat application.
  - Chat history, generated SQL, query results (in tables), insights, and schema information should be presented clearly and distinctly.
  - The schema display should be collapsible and easy to understand.
  - Loading indicators (spinners) should be used during potentially long operations like API calls or query execution.
  - The interface is designed to be responsive using Tailwind CSS, adapting to different screen sizes.

3.2 ER & Data Flow Diagrams

Entity-Relationship (ER) diagrams visually represent the structure of the databases. For DataFlow, we consider the sample databases SQLLLM and StoreDB generated by gen-data.py.

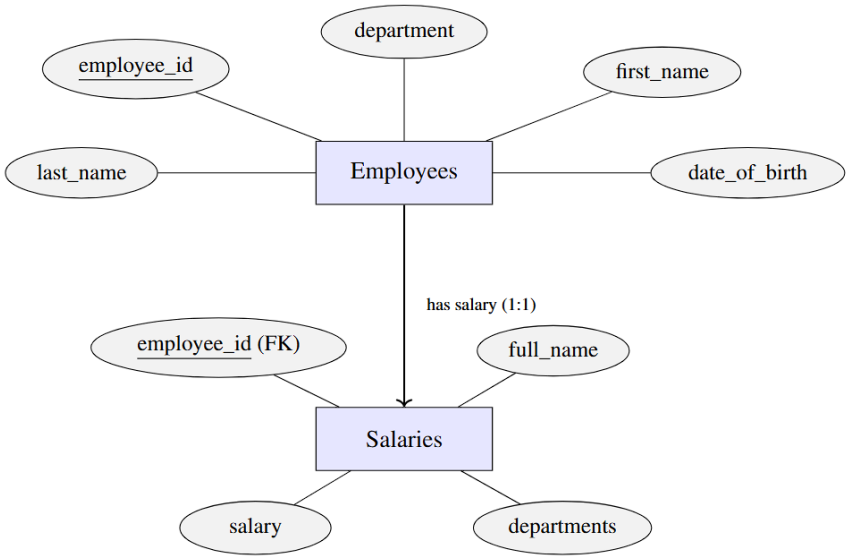

Figure 1: *ER Diagram for SQLLLM Database. A one-to-one relationship is implied between Employees and Salaries via employee_id.*

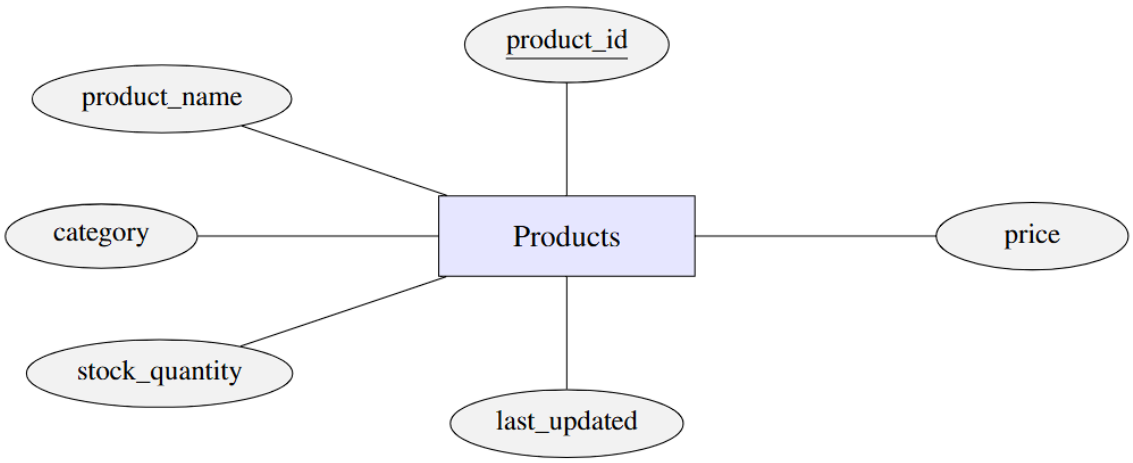

Figure 2*: ER Diagram for StoreDB Database. Shows the single Products entity and its attributes.*

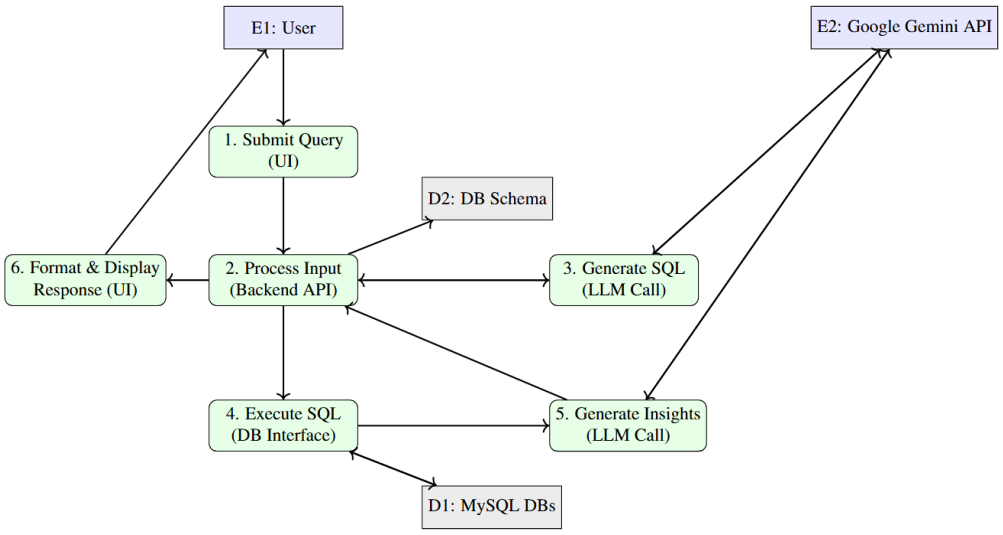

Figure 3: *Data Flow Diagram (DFD - Level 0/1) for a Natural Language Query*

3.3 Flow Diagram / Illustration (System Architecture)

The overall architecture of the DataFlow system illustrates the interaction between its major components.

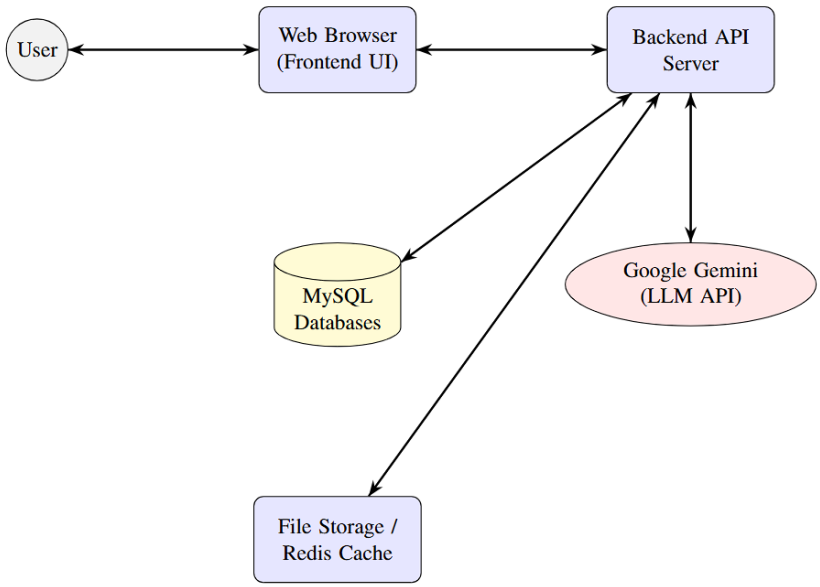

Figure 4: *System Architecture Diagram for DataFlow*

- ## Operational Workflow (Methodology Text)

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

1. User Input: The user types a natural language question (e.g., "Show me all products in the Electronics category") into the chat interface provided by index.html.
2. Request to Backend: The frontend JavaScript captures this message and sends it as a JSON payload to the /chat API endpoint of the FastAPI backend (sql_assistant.py).
3. Schema Fetching (Context Building): Upon receiving the request, if not already cached or if a refresh is triggered, the backend calls its fetch_all_tables_and_column function. This function connects to the MySQL server, identifies all non-system databases, and for each, lists its tables and their respective columns. This schema information is crucial context for the LLM.
4. SQL Generation via LLM: The backend constructs a detailed prompt. This prompt includes the user's natural language question and the comprehensive database schema string. This combined prompt is then sent to the Google Gemini API through the generate_sql_with_gemini function.
5. LLM Processing and Response: The Gemini API processes the prompt and returns what it determines to be the most appropriate SQL query to answer the user's question (e.g., SELECT * FROM StoreDB.products WHERE category = 'Electronics').
6. Initial SQL Validation (Backend): The backend receives the SQL from Gemini. It checks if the LLM returned a specific "Error: Cannot answer..." message or if the response seems malformed. If it's an unanswerable error, it may trigger a conversational fallback. If the query is identified as potentially modifying data (e.g., INSERT, UPDATE, DELETE, DDL commands), the backend sends a confirm_execution type response to the frontend, asking for user confirmation.
7. Query Pre-processing (e.g., SHOW TABLES;): For specific simple queries like a plain SHOW TABLES;, if multiple user databases exist, the backend might respond asking the user for clarification. If only one user database exists, it might prepend a USE database_name; statement.
8. SQL Execution (or Confirmation): If the query is a SELECT or SHOW statement (and not requiring confirmation), or if it's a DML/DDL from /run that has been confirmed (via /execute_confirmed_sql endpoint), the SQL query is passed to the execute_sql_query function. This function establishes a database connection and executes the query against the MySQL server.
9. Database Error Handling: If execute_sql_query encounters any database- level errors (e.g., syntax error, table not found), the error is caught, logged, and an appropriate error message is prepared.
10. Insight Generation (for SELECT/SHOW): If the query was a SELECT/SHOW type and executed successfully, returning results and column information, these details (original

question, generated SQL, results preview, column types) are sent to the Gemini API again, this time via the get_insights_with_gemini function, to generate a textual analysis or summary.

11. Response Aggregation and Formatting: The backend aggregates all relevant information: the type of response (result, info, error, confirmation), the generated SQL (if any), column names, query results (as a list of tuples), and the generated insights (if any). This is packaged into a JSON response.

12. Display to User: The frontend JavaScript receives this JSON response. It then dynamically updates the chat interface in index.html to display the information to the user in a structured and readable way (e.g., SQL in a code block, results in an HTML table, insights as formatted text).

For commands initiated with /run, steps 3-6 related to LLM-based SQL generation from natural language are bypassed. The user-provided SQL is taken (with confirmation for modifying queries), potentially pre-processed (step 7), and then sent for execution from step 8 onwards.

3.4.1 Database Design Details

The DataFlow system is designed to primarily interact with MySQL databases. The backend Python application (sql_assistant.py) uses the mysql-connector-python library. For demonstration, the gen-data.py script sets up two sample MySQL databases: SQLLLM and StoreDB.

SQLLLM.employees Table Structure

Description: Stores basic information about employees.

```
+-----------------+----------------------+
| Field           | Type                 |
+-----------------+----------------------+
| employee_id     | int                  |
| first_name      | varchar(50)          |
| last_name       | varchar(50)          |
| date_of_birth   | date                 |
| department      | varchar(50)          |
+-----------------+----------------------+
```

SQLLLM.salaries Table Structure

Description: Stores salary details for employees, linked by employee_id.

```
+-------------+----------------------+
| Field       | Type                 |
+-------------+----------------------+
| employee_id | int                  |
| full_name   | varchar(100)         |
| salary      | int                  |
| departments | varchar(50)          |
+-------------+----------------------+
```

StoreDB.products Table Structure

Description: Stores information about products available in a store.

```
+------------------+-------------------------+
| Field            | Type                    |
| product_id       | int                     |
| product_name     | varchar(100)            |
| category         | decimal(10,2)           |
| price            | int                     |
+------------------+-------------------------+
```

Chapter 4: Testing

The DataFlow system underwent several testing phases to ensure its functionality, reliability, and usability. Testing focused on the core capabilities: natural language to SQL conversion, direct SQL execution, schema handling, insight generation, and user interface responsiveness.

4.0.1 Testing Techniques Employed

- Manual Testing: This was the primary method used throughout development. It involved direct interaction with the web interface by inputting a wide variety of natural language queries, /run commands with different SQL statements, and observing the system's responses. This included testing edge cases and error conditions.
- Black Box Testing: The system was tested from a user's perspective, focusing on inputs and outputs without considering the internal code structure. This ensured that the application meets user requirements and behaves as expected under various scenarios.
- Error Guessing and Boundary Value Analysis (Informal): Based on experience with SQL and natural language systems, potential error-prone inputs were tested. This included ambiguous natural language questions, syntactically incorrect SQL commands, queries targeting non-existent tables/columns, and empty inputs.
- Integration Testing (Informal): Focused on ensuring that all major components of the system (Frontend UI, FastAPI Backend, Google Gemini API, MySQL Database) work together correctly. For example, a test would verify that a natural language query correctly flows through to SQL generation, database execution, result retrieval, insight generation, and final display on the UI.
- Usability Testing (Informal): The chat interface and schema display were informally tested for ease of use, clarity of information, and overall user experience.
- Unit Testing (Conceptual): While formal automated unit test suites (e.g., using pytest) were not implemented for all functions due to the academic project scope, individual functions in the backend (sql_assistant.py), such as those for database connection, query execution logic, and prompt construction, were tested in isolation during development to ensure they performed their specific tasks correctly.

4.0.2 Test Plan and Sample Test Cases

A test plan was devised to cover the key functionalities. Below are sample test cases that represent the types of tests conducted.

Table 4: *Sample Test Cases for System Validation*

Test ID	Description	Expected Input (Example)	Expected Output / Behavior
TC_NLQ_01	Simple NLQ - Fetch all records	"Show all employees"	SQL: SELECT * FROM SQLLLM.employees;, results table, and AI-generated insights
TC_NLQ_02	NLQ with WHERE clause	"Products in Electronics under $100"	SQL with WHERE category = 'Electronics' AND price < 100, filtered results, and insights
TC_NLQ_03	NLQ requiring JOIN	"Salaries of Engineering employees?"	SQL joining employees and salaries tables, filtered results, insights
TC_NLQ_04	NLQ unanswerable	"Weather today?"	Error message: "Cannot answer..." or fallback response
TC_SQL_01	Direct SQL - Valid SELECT	/run SELECT name, price FROM StoreDB.products LIMIT 5;	Results table displayed
TC_SQL_04	Direct SQL - Invalid	/run SELEC * FROM employees;	SQL error message from database
TC_UI_01	Schema Toggle	Click "View/Hide Schema" button	Schema panel appears/disappears

Each test case was executed, and the observed behavior was compared against the expected outcome. Issues identified during testing were logged, debugged, and retested until resolved.

Chapter 5: Conclusion and Future Work

5.1 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.

5.2 Future Work

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.

11. **Voice Input Support:** Add functionality to accept user queries via voice input.

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

Bibliography

[1] Official Python Website. (2024). Python Software Foundation. Retrieved from https://www.python.org

[2] Tiangolo, Sebastián. (2024). FastAPI - A Python framework for building APIs. Retrieved from https://fastapi.tiangolo.com/

[3] Oracle Corporation. (2024). MySQL. Retrieved from https://www.mysql. com/

[4] Google. (2024). Gemini API. Google AI for Developers. Retrieved from https: //ai.google.dev/

[5] Faker (Python Package Index). Retrieved from https://pypi.org/ project/Faker/ (Documentation available at: https://faker. readthedocs.io/)

[6] Tailwind Labs Inc. (2024). Tailwind CSS - A utility-first CSS framework. Retrieved from https://tailwindcss.com/

[7] Oracle Corporation. (2024). MySQL Connector/Python Developer Guide. Retrieved from https://dev.mysql.com/doc/connector-python/en/

[8] Marked.js Organization. (2024). A markdown parser and compiler. Built for speed. Retrieved from https://marked.js.org/

[9] Lucide Contributors. (2024). Beautiful & consistent icon toolkit. Retrieved from https://lucide.dev/

[10] Pydantic. (2024). Pydantic Documentation. Retrieved from https://docs.pydantic.dev/

[11] python-dotenv (Python Package Index). Retrieved from https://pypi.org/project/python-dotenv/

[12] Uvicorn. (2024). Uvicorn - The lightning-fast ASGI server. Retrieved from https://www.uvicorn.org/

List of Figures

List of Tables

Chapter 6: User Interface

This chapter presents snapshots of the DataFlow application's user interface, illustrating its key features and how users interact with the system.

6.1 Main Chat Interface

The main interface provides a familiar chat-like environment. Users can type natural language queries or direct SQL commands into the input field at the bottom. The chat history displays the conversation, including user inputs, generated SQL, query results, and AI-generated insights.

6.2 Natural Language Query and System Response

This snapshot shows an example of a user asking a question in natural language. The system responds with the SQL query it generated, a table containing the results fetched from the database, and textual insights derived from those results.

6.3 Direct SQL Execution using /run Command

Users with SQL knowledge can execute queries directly using the /run prefix. This example shows a user running a SELECT query. The system displays the executed query and the resulting data table. For modifying queries, a confirmation step is included.

6.4 Database Schema Display Panel

DataFlow allows users to view the schema of the connected databases. This panel, toggled by a button, lists databases, their tables, and columns, helping users formulate queries or understand the data structure. A refresh button allows updating the schema view.

6.5 Error Message Display

The system provides feedback when errors occur. This could be due to an ambiguous natural language query that the LLM cannot translate, an invalid SQL command, a database error during execution, or issues with API communication. The error messages aim to be informative.

6.6 Configuration Popup

Users can configure database connection details (MySQL host, user, password) and the Gemini API key through a dedicated configuration popup in the UI. Saving the configuration updates the backend settings.

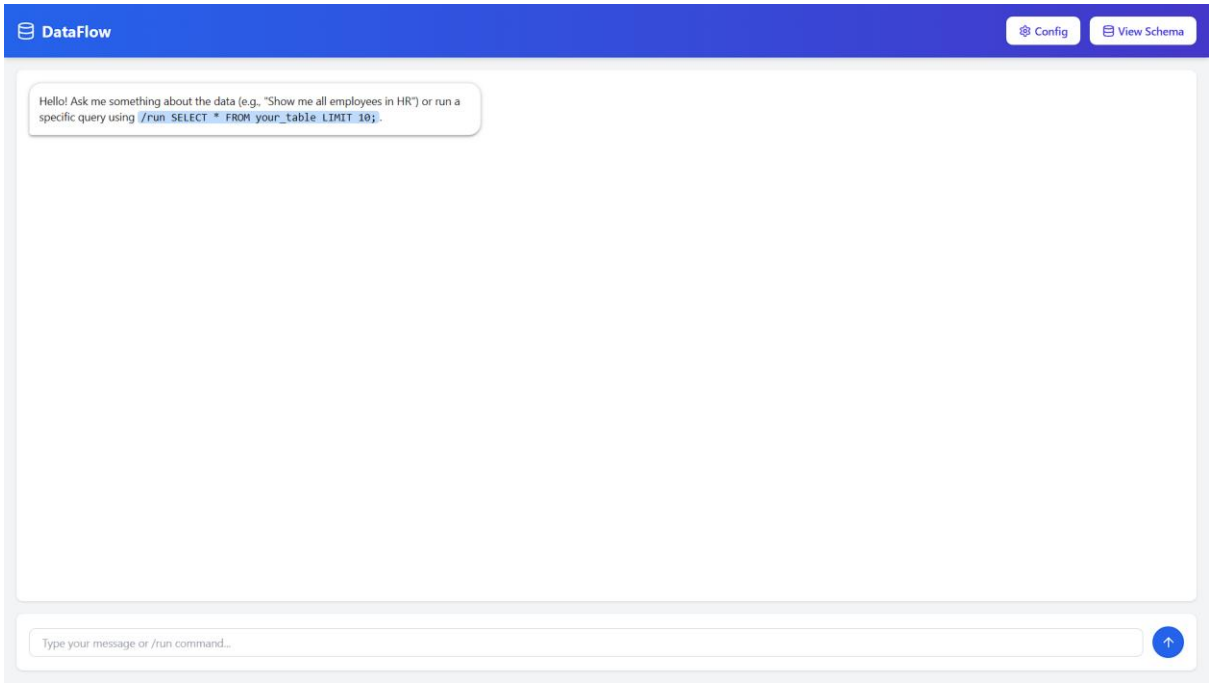

Figure 5: *Main Chat Interface of DataFlow, showing chat history and input area.*

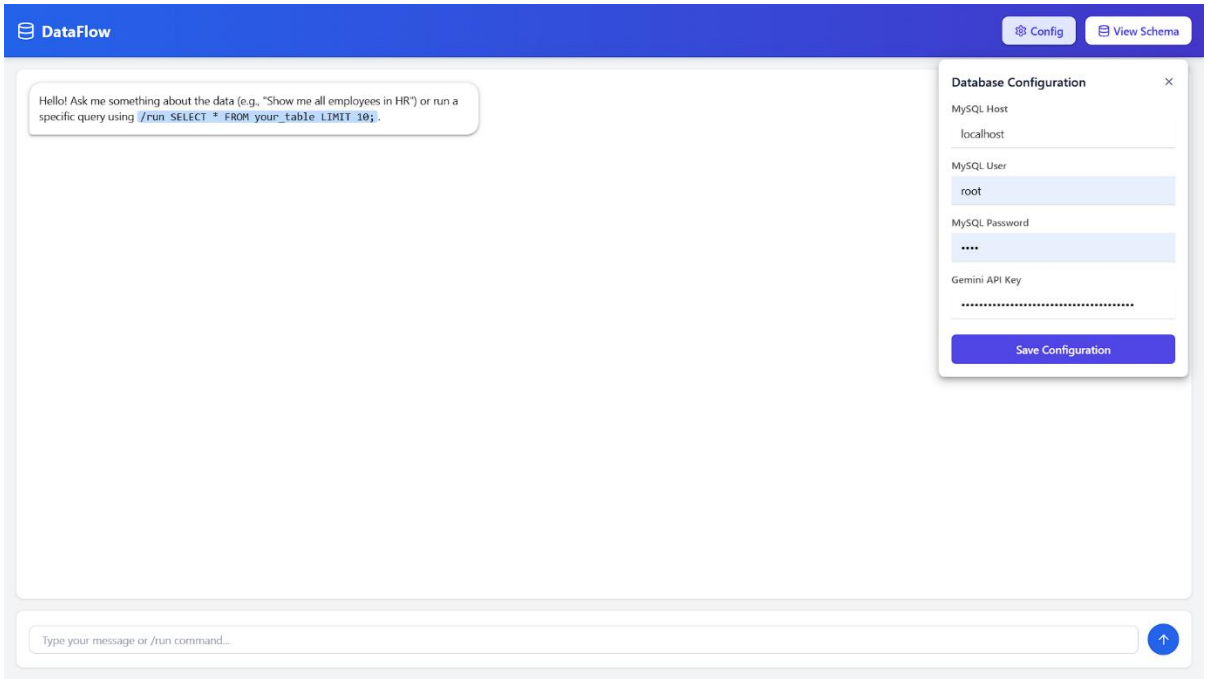

Figure 6*: Config Panel, showing Host, User, Password, API Key Input Fields*

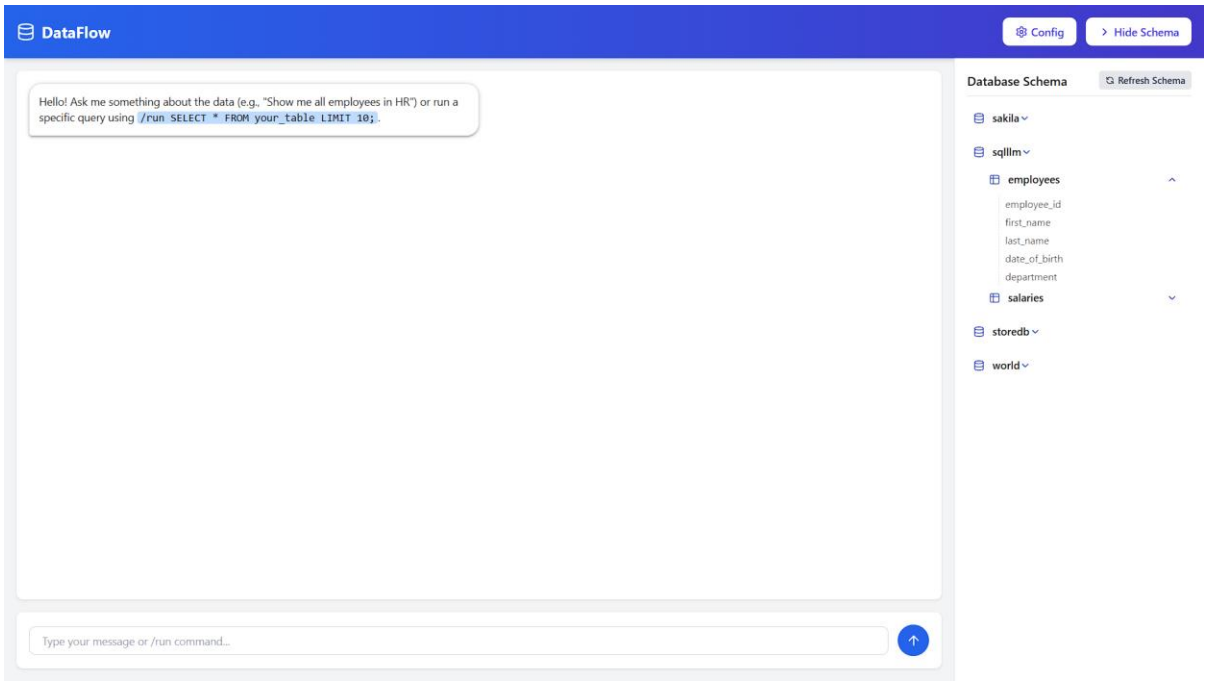

Figure 7: *Database Schema Display Panel, showing available databases, tables, and columns.*

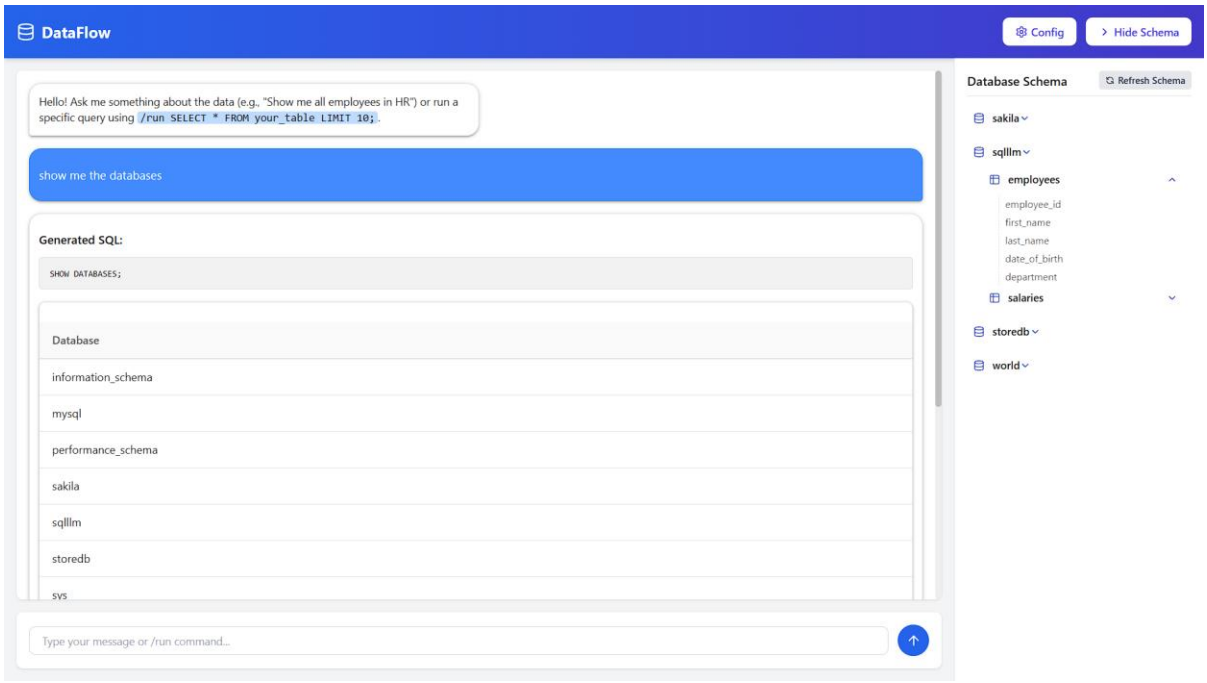

Figure 8: *Example of a Natural Language Query and the System's Detailed Response (Generated SQL, Results Table, Insights).*

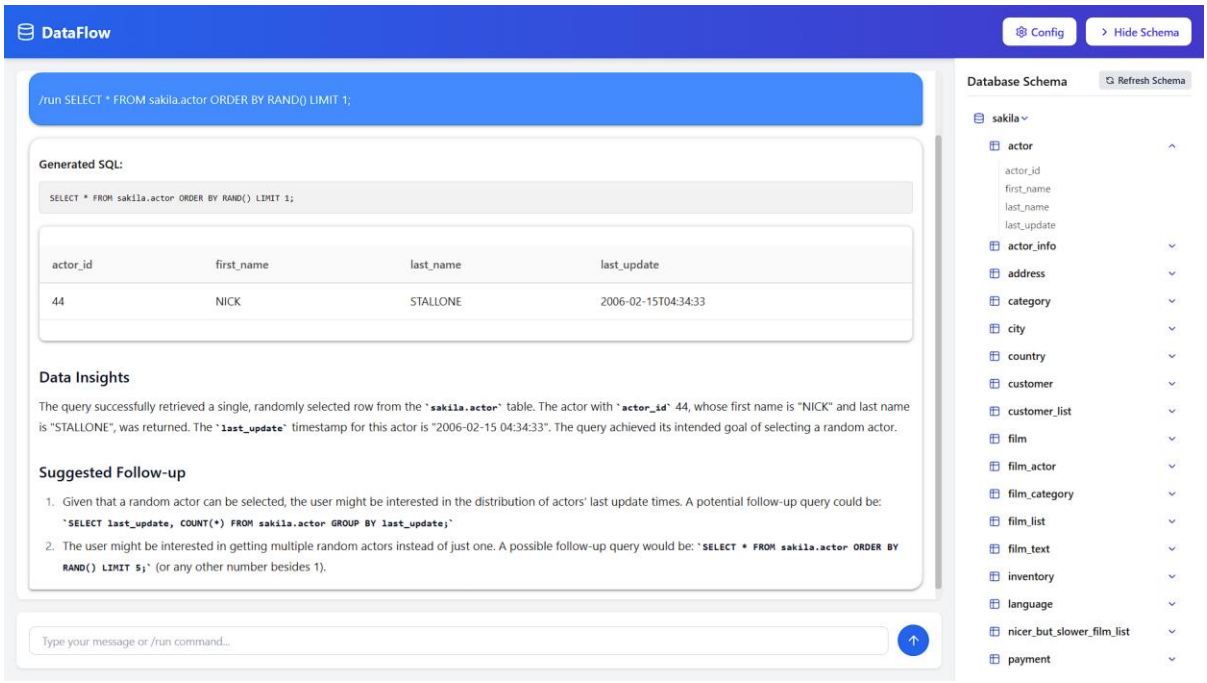

Figure 9: *Example of Direct SQL Execution using the /run command and the displayed results.*

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 (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)**

```
# 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)
```