Project Title:

IntelliSQL: Intelligent SQL Querying with LLMs Using Gemini 2.5 Flash

---

Submitted by:
BATTULA AKHILA
reg.no. 22FE1A4204, CSM

Team ID : LTVIP2026TMIDS65861
Team Size : 4
Team Leader : Battula Akhila
Team member : Jessie Evans Nannam
Team member : Karrevula Bhargav
Team member : Kode Yogitha

Vignan's Lara Institute of Technology and Science
Department of ACSE

Under the Guidance of:
SmartBridge Internship Program
Google Cloud Generative AI

ABSTRACT

IntelliSQL is an AI-powered web application designed to bridge the gap between natural language understanding and structured database querying. Traditional database systems require users to possess knowledge of Structured Query Language (SQL) to retrieve and analyze stored data. However, many non-technical users find SQL syntax complex and difficult to learn. This project addresses that limitation by enabling users to interact with databases using plain English queries.

The system leverages Google's Gemini 2.5 Flash large language model (LLM) to convert natural language input into syntactically correct SQL queries. A multi-page Streamlit web interface is developed to provide an intuitive and interactive user experience. When a user submits an English query, the application processes it through a structured prompt engineering layer, sends it to the Gemini API, receives the generated SQL statement, and executes it on a SQLite database. The query results are then displayed in a formatted tabular form within the interface.

The architecture follows a modular design consisting of a presentation layer, LLM processing layer, and database execution layer. Security considerations such as API key protection, query validation, and restriction to SELECT statements are implemented to ensure safe database interactions. The project demonstrates how modern generative AI models can be integrated with traditional database systems to improve accessibility, efficiency, and user experience.

IntelliSQL successfully showcases the practical application of large language models in real-world data retrieval scenarios and highlights the transformative potential of AI-driven database interfaces.

INTRODUCTION

Database management systems play a critical role in storing, organizing, and retrieving structured data across various industries. Structured Query Language (SQL) is the standard language used to interact with relational databases. Although SQL is powerful and efficient, it requires users to understand syntax, commands, and query structure. This requirement creates a barrier for non-technical users who need to access data but lack programming knowledge.

With the advancement of Artificial Intelligence and Natural Language Processing (NLP), it has become possible to interpret human language and convert it into structured commands. Large Language Models (LLMs), such as Google's Gemini series, are capable of understanding context, intent, and semantic meaning within natural language inputs. These capabilities make them highly suitable for translating English questions into executable SQL queries.

The integration of LLMs with database systems introduces a new paradigm of human-computer interaction. Instead of writing complex SQL statements manually, users can simply describe what they need in plain English. The system then automatically generates the corresponding SQL query and executes it on the database. This approach significantly improves usability, accessibility, and productivity.

The IntelliSQL project is developed to demonstrate this AI-driven database interaction model. By combining Streamlit for user interface development, Gemini 2.5 Flash for natural language processing, and SQLite for data storage, the system provides a seamless pipeline from user input to structured data output. The project emphasizes modular architecture, prompt engineering, security considerations, and deployment readiness.

PROBLEM STATEMENT

In traditional database systems, retrieving information requires knowledge of Structured Query Language (SQL). While SQL is powerful and expressive, it demands familiarity with syntax, logical operators, aggregation functions, and relational concepts. Many users who need access to organizational data, such as managers, analysts, or students, may not possess the technical expertise required to construct correct SQL queries.

Existing database interfaces typically rely on either manual SQL input or rigid form-based filtering systems. Manual SQL entry increases the risk of syntax errors and requires training. On the other hand, predefined filtering interfaces limit flexibility and cannot accommodate complex or dynamic user requests. These limitations create inefficiencies in data retrieval and restrict accessibility.

Although natural language interfaces have been explored in the past, earlier systems relied on rule-based approaches and lacked contextual understanding. Such systems struggled with variations in phrasing, ambiguity, and complex query structures.

With the emergence of large language models, it is now possible to accurately interpret user intent and convert natural language into structured database queries. However, integrating LLMs securely and efficiently into database-driven applications presents architectural and validation challenges.

Therefore, there is a need for an intelligent system that:

- Allows users to query databases using natural language.
- Converts English queries into syntactically correct SQL statements.
- Ensures safe and secure database execution.
- Provides an intuitive and user-friendly interface.

The IntelliSQL project addresses this need by designing and implementing an AI-powered natural language to SQL translation system using Google Gemini and Streamlit.

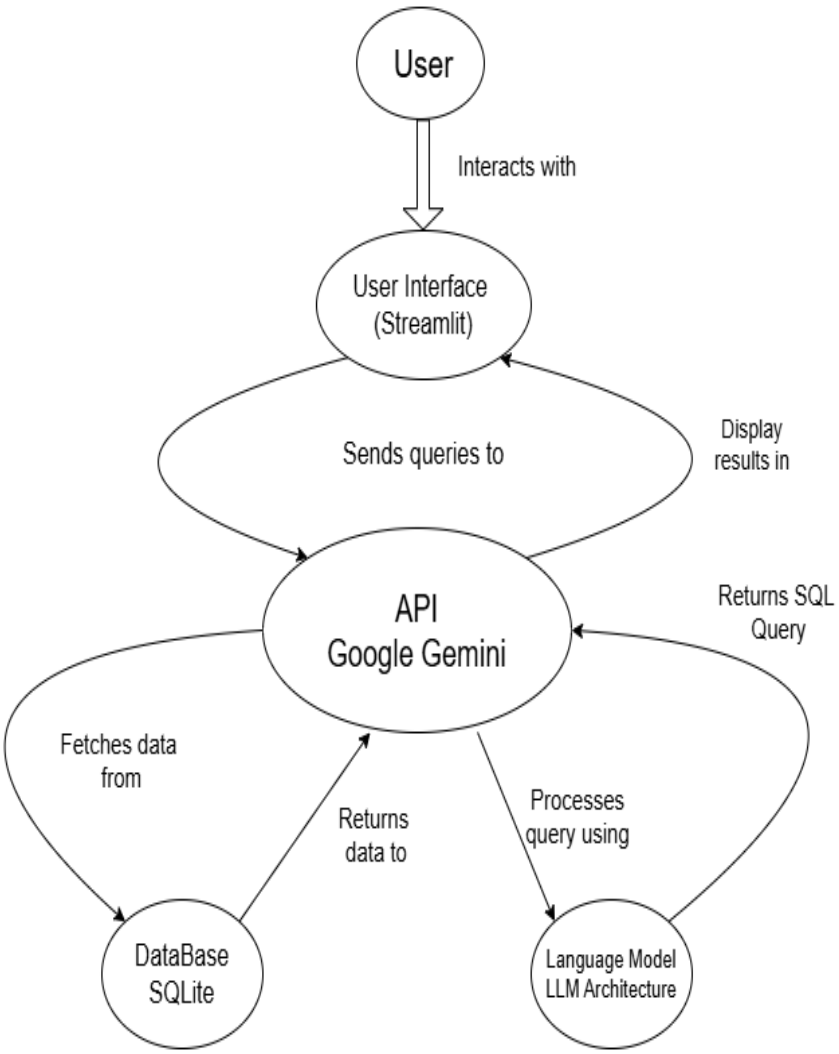
OBJECTIVES

The primary objective of the IntelliSQL project is to design and develop an intelligent web-based system capable of converting natural language queries into executable SQL statements using a large language model.

The specific objectives of the project are as follows:

1. To design a user-friendly multi-page web interface using Streamlit for natural language query input and result visualization.
2. To integrate Google Gemini 2.5 Flash large language model for accurate English-to-SQL translation.
3. To implement structured prompt engineering techniques to ensure reliable and consistent SQL generation.
4. To develop a modular architecture separating the user interface layer, LLM processing layer, and database execution layer.
5. To execute generated SQL queries on a SQLite database and display the results in a structured tabular format.
6. To implement security measures such as API key protection and restriction of database operations to SELECT queries.
7. To ensure that the system is deployment-ready and scalable for future enhancements.

SYSTEM ARCHITECTURE



The IntelliSQL system follows a modular and layered architecture that ensures separation of concerns and maintainability. The architecture is designed to process natural language input from the user and convert it into structured database queries through an AI-powered pipeline.

The system begins at the user interface layer, developed using Streamlit. Users interact with the application through a browser, where they enter natural language queries in a text input field. Upon submission, the query is passed to the processing layer.

The processing layer contains the get_response() function, which integrates with Google Gemini 2.5 Flash through the official SDK. A structured system prompt is combined with the user query to guide the model in generating accurate and syntactically correct SQL statements. Prompt engineering ensures that the model outputs only SELECT queries and does not produce explanatory text.

Once the SQL query is generated, it is forwarded to the database execution layer. The read_query() function establishes a connection with the SQLite database (data.db), executes the SQL command, and retrieves the results. Security validation ensures that only SELECT statements are executed, preventing destructive database operations.

Finally, the retrieved results are converted into a tabular format using Pandas and displayed back in the Streamlit interface. This completes the end-to-end pipeline from natural language input to structured data output.

The modular architecture enhances scalability, security, and ease of maintenance, allowing future integration with cloud databases, authentication systems, and advanced AI features.

TECHNOLOGY STACK

The IntelliSQL project is developed using a combination of modern web technologies, artificial intelligence tools, and database systems. Each technology was selected based on performance, ease of integration, and scalability.

1. Programming Language – Python
   Python was used as the primary programming language due to its simplicity, extensive library support, and strong ecosystem for artificial intelligence and web development.

2. Web Framework – Streamlit
   Streamlit was used to build the multi-page web application interface. It enables rapid development of interactive web applications using pure Python without requiring front-end technologies such as HTML, CSS, or JavaScript.

3. Large Language Model – Google Gemini 2.5 Flash
   Gemini 2.5 Flash was used for natural language processing and SQL query generation. It provides advanced contextual understanding, fast inference speed, and high accuracy in text generation tasks.

4. Database – SQLite
   SQLite was chosen as the relational database management system for this project. It is lightweight, serverless, and suitable for local development and demonstration purposes.

5. Prompt Engineering
   Structured prompt engineering techniques were implemented to guide the LLM in generating accurate and secure SQL queries.

6. Environment Management – Python Virtual Environment (venv)
   A virtual environment was used to isolate dependencies and maintain project reproducibility.

7. Version Control – Git and GitHub
   Git was used for version control, and the project repository is hosted on GitHub for submission and collaboration.

DATABASE DESIGN

The IntelliSQL system uses a relational database designed using SQLite. The database is named data.db and contains a single table called STUDENTS. The schema was intentionally kept simple to demonstrate natural language to SQL translation effectively while supporting filtering, aggregation, and conditional queries.

The relational model was selected because SQL is designed for structured relational data. SQLite provides a lightweight and serverless database engine suitable for demonstration and local deployment purposes.

Table Name: STUDENTS

The STUDENTS table contains the following attributes:

1. NAME (TEXT)
   Stores the name of the student.
2. CLASS (TEXT)
   Represents the academic qualification or course of the student (e.g., BTech, MTech, MSc, etc.).
3. MARKS (INTEGER)
   Stores the academic score obtained by the student.
4. COMPANY (TEXT)
   Indicates the company where the student is placed or associated.

The MARKS attribute is defined as an integer to support aggregation operations such as COUNT, AVG, MAX, and MIN.

```
CREATE TABLE IF NOT EXISTS STUDENTS(
    NAME TEXT,
    CLASS TEXT,
    MARKS INTEGER,
    COMPANY TEXT
);
```

The table is initialized with sample records to demonstrate query functionality:

Sita | BTech | 85 | INFOSYS
Ram | MTech | 78 | TCS
Shibin | MSc | 90 | INFOSYS
Riya | BSc | 88 | WIPRO
Diya | MCom | 92 | EY

These records allow testing of filtering conditions, aggregation queries, and data retrieval operations.

PROMPT ENGINEERING STRATEGY

Prompt engineering plays a critical role in ensuring accurate and reliable SQL query generation. Large Language Models generate outputs based on the instructions and context provided to them. Without structured guidance, the model may produce explanations, markdown formatting, or unsafe database commands.

To address this, a carefully designed system prompt was implemented. The prompt explicitly defines the database schema, table name, column names, and generation rules. This ensures that the model understands the structure of the database before generating SQL statements.

The prompt enforces strict constraints such as:

- Generating only SQL queries.
- Using exact column names as defined in the schema.
- Avoiding explanations or additional text.
- Restricting output to SELECT statements.
- Preventing destructive operations such as DELETE, UPDATE, DROP, or INSERT.

By combining schema definition, rules, and examples within the prompt, the model is guided toward deterministic and secure SQL generation.

"You are an expert in converting English questions to SQL queries.

The SQL database table name is STUDENTS.
The table STUDENTS contains columns:
NAME, CLASS, MARKS, COMPANY.

Rules:
Only generate SQL queries.
Use correct column names exactly as given.
Do not explain anything.
Return only the SQL statement.
Do not include markdown formatting.
Do not generate DELETE, DROP, UPDATE, INSERT statements."

Few-shot examples were included within the prompt to improve consistency and reliability. By providing sample input-output pairs, the model learns the expected format and response structure.

Example:

How many entries are present?
SELECT COUNT(*) FROM STUDENTS;

Show students working in INFOSYS.
SELECT * FROM STUDENTS WHERE COMPANY='INFOSYS';

This technique significantly improves SQL accuracy and reduces formatting errors.

Prompt engineering ensures that the system behaves predictably and securely. Without proper prompt constraints, the model could generate ambiguous queries or unsafe commands. By defining strict output rules and schema awareness, the system maintains control over LLM behavior while leveraging its natural language understanding capabilities.

IMPLEMENTATION DETAILS

The IntelliSQL project follows a modular architecture to ensure maintainability, scalability, and separation of concerns. The application is divided into multiple Python modules, each responsible for a specific functionality. This structure improves readability and simplifies debugging and future enhancements.

The project consists of the following main files:

- app.py
- database.py
- llm_engine.py
- prompt_config.py

Each module performs a clearly defined role within the system.

1. **app.py** – Application Controller and User Interface

The app.py file serves as the main entry point of the application. It initializes the Streamlit configuration, manages page navigation, and coordinates interactions between the user interface, LLM engine, and database layer.

Key Responsibilities:

- Configures the Streamlit page layout and sidebar navigation.
- Defines three pages: Home, About, and Intelligent Query Assistance.
- Accepts natural language input from users.
- Calls the get_response() function to generate SQL queries.
- Validates that only SELECT statements are executed.
- Displays query results in tabular format using Pandas.

This file acts as the controller layer in the system architecture.

2. **database.py** – Database Management Layer

The database.py file manages all interactions with the SQLite database. It contains functions responsible for database initialization and SQL execution.

Key Functions:

- init_db(): Creates the STUDENTS table if it does not exist and inserts sample records.
- read_query(sql, db): Executes SQL queries and retrieves results safely.

This module ensures that database operations remain isolated from the user interface and LLM processing logic.

3. **llm_engine.py** – LLM Integration Layer

The llm_engine.py file handles integration with Google Gemini 2.5 Flash through the official SDK. It manages API configuration and sends structured prompts to the model.

Key Responsibilities:

- Loads API key securely using dotenv.
- Initializes the Gemini client.
- Combines the system prompt with user input.
- Sends requests to the model.
- Returns generated SQL output.

This layer is responsible for translating natural language into SQL commands.

SECURITY CONSIDERATIONS

Security is a critical component when integrating large language models with database systems. Since the IntelliSQL application automatically executes SQL statements generated by an AI model, safeguards must be implemented to prevent misuse or unintended database operations.

The following security measures were incorporated into the system:

1. API Key Protection
   The Gemini API key is stored securely in a .env file and loaded using the python-dotenv library. The key is never hardcoded within the application. Additionally, the .env file is included in the .gitignore configuration to prevent accidental exposure in version control repositories.

2. Restriction to SELECT Queries
   Before executing any generated SQL query, the system validates that the query begins with the SELECT keyword. This ensures that destructive commands such as DELETE, UPDATE, DROP, and INSERT are not executed on the database.

3. Prompt-Level Constraints
   The structured system prompt explicitly instructs the LLM to generate only SELECT queries and avoid unsafe operations. This adds an additional layer of protection at the AI output level.

4. Local Database Isolation
   The SQLite database is stored locally within the application environment. This limits exposure and prevents unauthorized remote access.

5. Modular Validation Layer
   By separating SQL generation and SQL execution into different modules, the system ensures that validation logic can be extended or strengthened in future implementations.

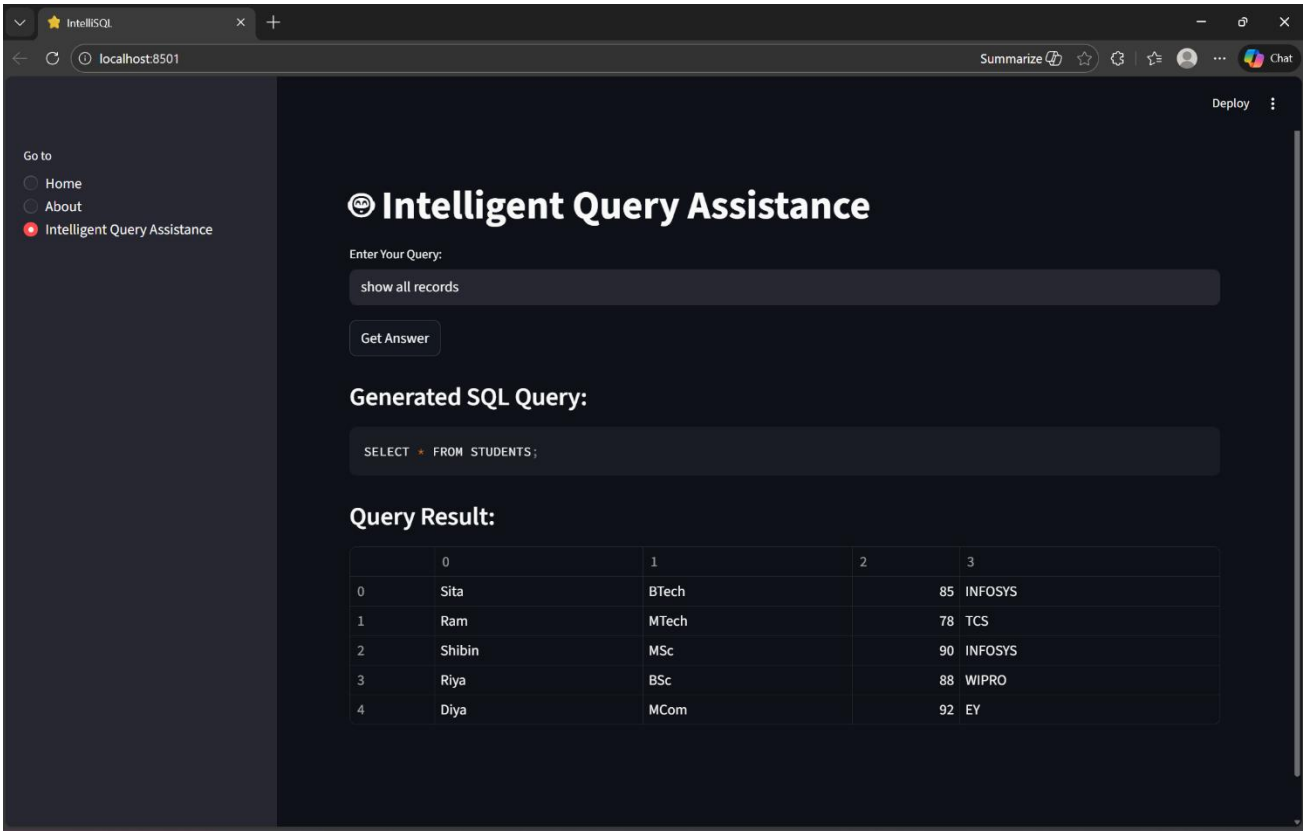
These security measures collectively ensure safe AI-driven database interaction while maintaining system integrity.

TESTING AND RESULTS

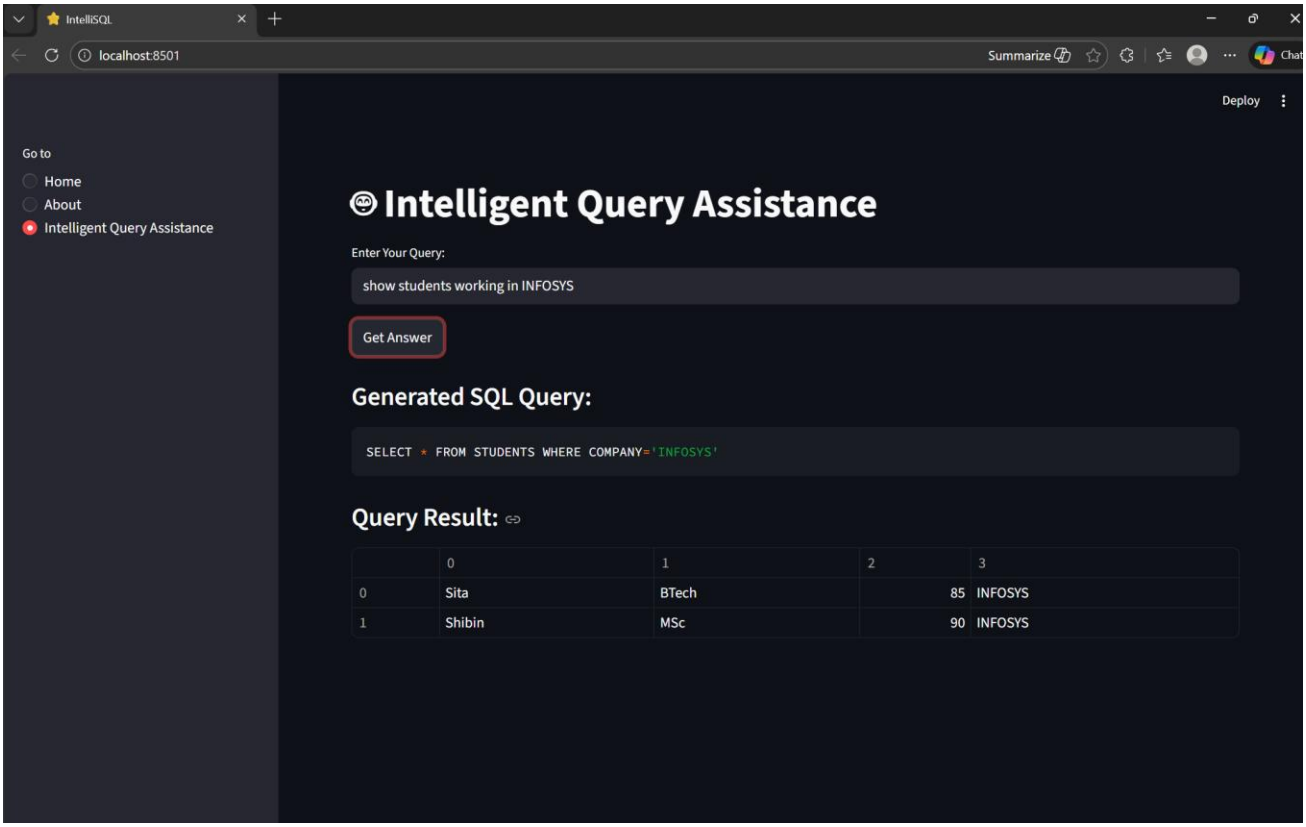
The IntelliSQL system was tested using multiple natural language queries to evaluate the accuracy of SQL generation and correctness of database execution. The testing process focused on validating filtering operations, aggregation queries, conditional statements, and overall system stability.

Each test case includes the user input, the generated SQL query, and the resulting output retrieved from the SQLite database.

Test Case 1
User Input: show all records
Generated SQL: SELECT * FROM STUDENTS;
Result Description: Displays all five student records from the database.



Test Case 2
User Input: show students working in INFOSYS
Generated SQL: SELECT * FROM STUDENTS WHERE COMPANY='INFOSYS';
Result Description: Displays student records where COMPANY is INFOSYS.

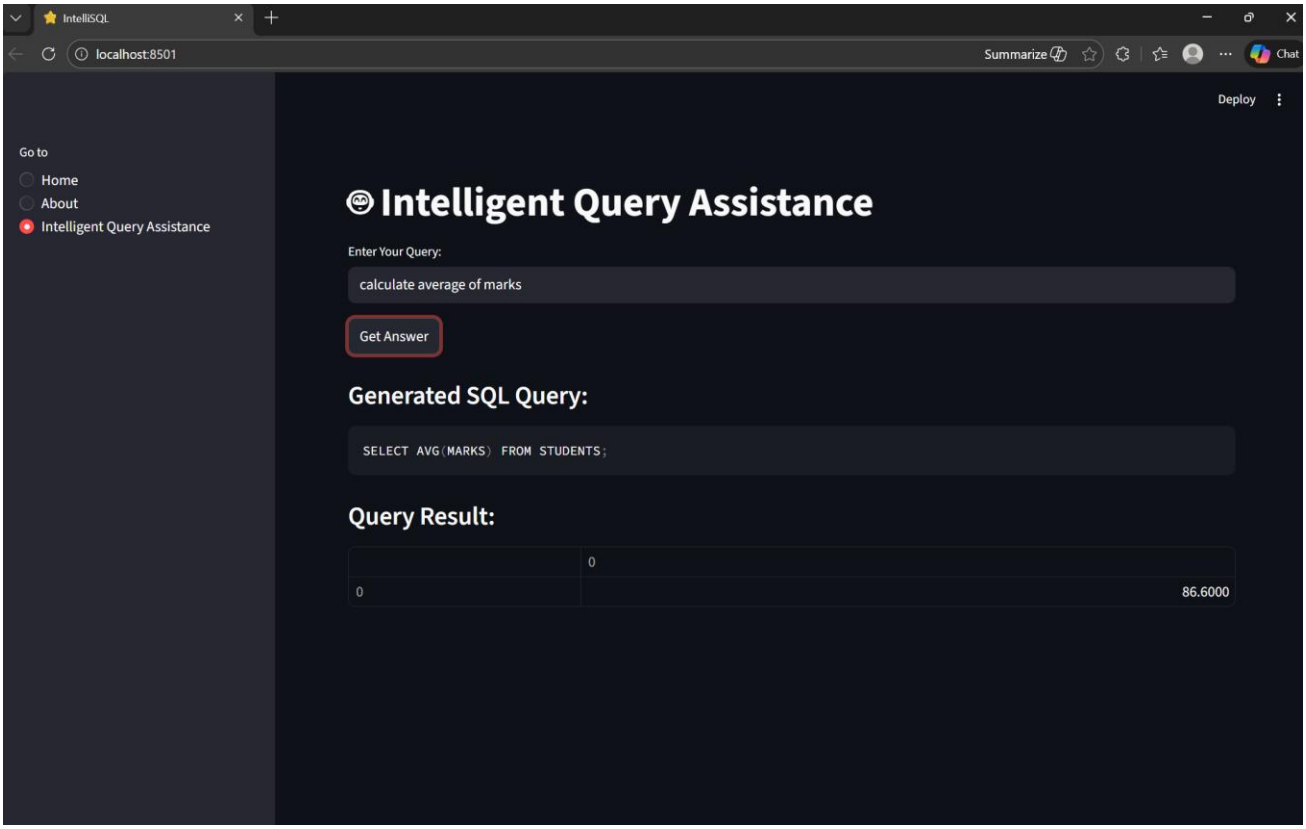


Test Case 3

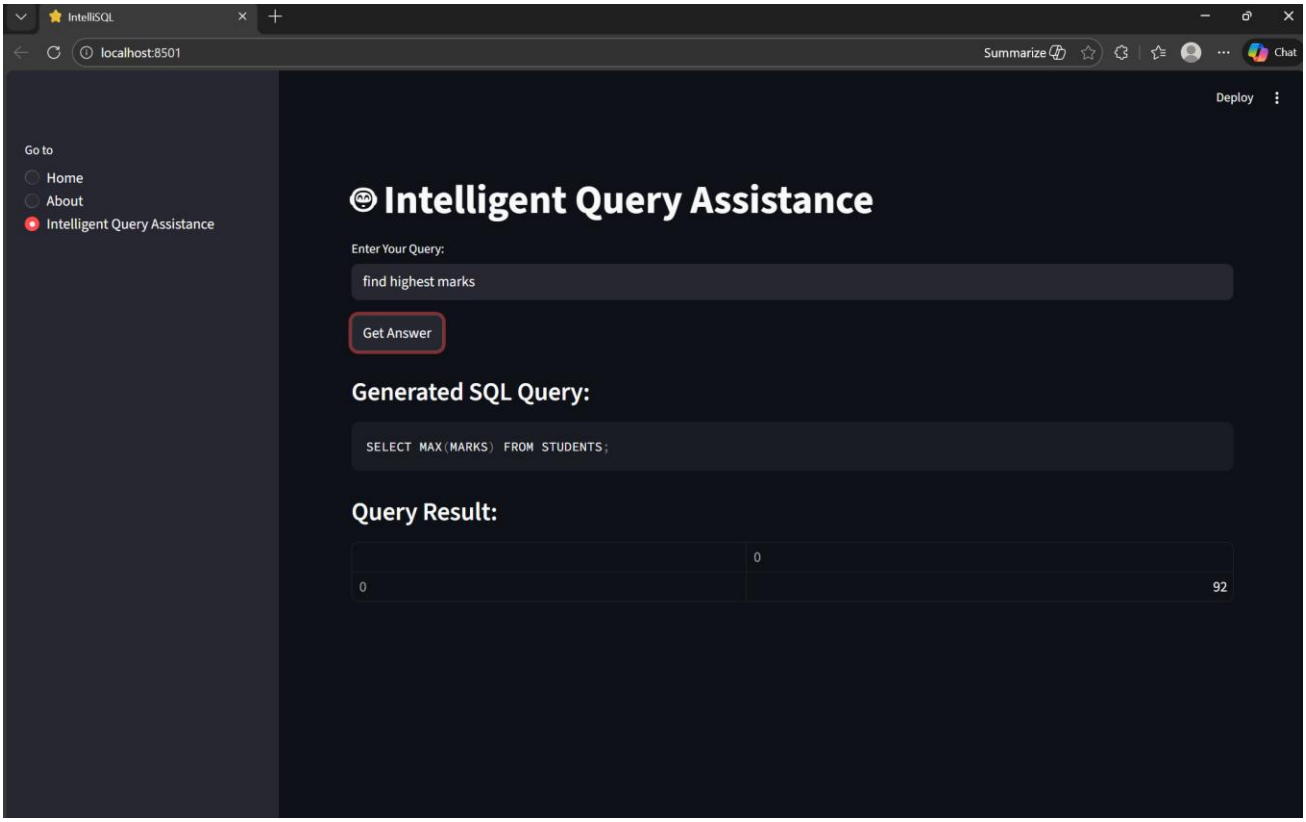
User Input: calculate average of marks

Generated SQL: SELECT AVG(MARKS) FROM STUDENTS;

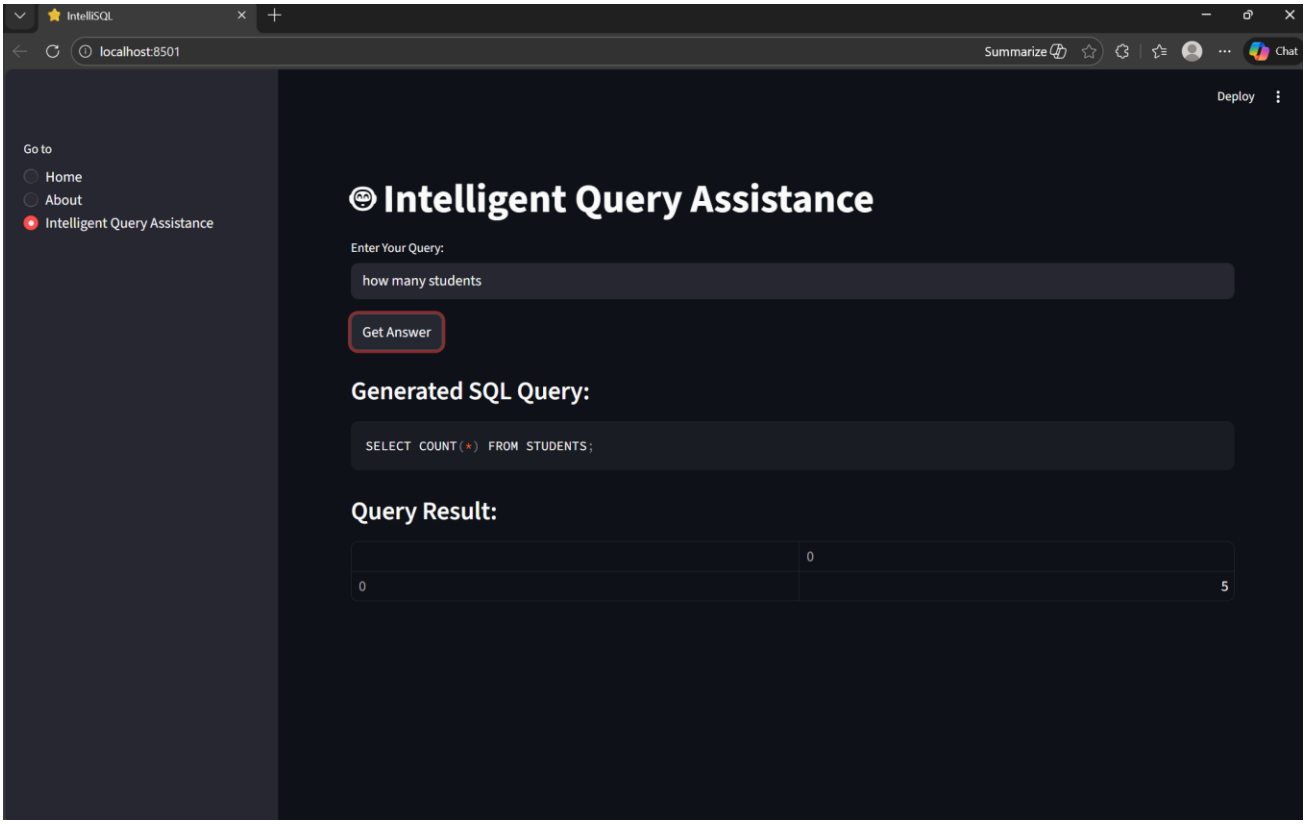
Result Description: Returns the average marks of all students.



Test Case 4
User Input: find highest marks
Generated SQL: SELECT MAX(MARKS) FROM STUDENTS;
Result Description: Returns the maximum marks value stored in the database.



Test Case 5
User Input: how many students
Generated SQL: SELECT COUNT(*) FROM STUDENTS;
Result Description: Returns total number of students.



The system demonstrated consistent performance across all tested cases. The Gemini 2.5 Flash model accurately interpreted natural language variations and generated syntactically correct SQL queries. The modular validation layer ensured that only safe SELECT statements were executed.

No runtime errors were observed during standard testing scenarios. The results were displayed in a structured tabular format using the Streamlit interface.

DEPLOYMENT STRATEGY

The IntelliSQL application is designed to run locally using Streamlit. After installing dependencies listed in the requirements.txt file, the application can be launched using the command:

streamlit run app.py

This starts a local web server, typically accessible at http://localhost:8501. The local deployment approach ensures quick testing, debugging, and demonstration during development.

To improve portability and environment consistency, the application can be containerized using Docker. A Dockerfile is provided to package the application along with its dependencies into a lightweight container.

The Docker image ensures that the application runs consistently across different systems without dependency conflicts. This approach enhances reproducibility and simplifies deployment.

Example Docker configuration:

```
FROM python:3.10
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["streamlit", "run", "app.py", "--server.port=8501", "--server.address=0.0.0.0"]
```

Containerization prepares the project for deployment on cloud platforms such as AWS, Google Cloud, or Azure.

For production-level deployment, the system can be extended to use cloud infrastructure. Possible deployment options include:

- Google Cloud Run
- AWS Elastic Beanstalk
- Azure App Services

The SQLite database can be replaced with scalable cloud databases such as PostgreSQL or Google Cloud SQL for improved performance and concurrency support.

The API key can be securely managed using environment variables or secret management services in cloud platforms.

The deployment-ready structure of IntelliSQL demonstrates practical software engineering skills beyond basic development. The use of environment isolation, dependency management, and containerization reflects industry-standard practices.

FUTURE ENHANCEMENTS

Although IntelliSQL successfully demonstrates natural language to SQL translation using large language models, several enhancements can further improve its functionality and scalability.

1. Support for Multiple Tables and JOIN Queries
   Currently, the system operates on a single table. Future versions can extend support to multiple related tables and enable automatic JOIN query generation.
2. Dynamic Schema Detection
   Instead of manually defining the schema in the prompt, the system can dynamically read database schema information and generate context-aware prompts automatically.
3. Conversational Query Support
   The system can be enhanced with conversational memory to support follow-up queries such as “show only those above 85 marks” after a previous query.
4. Role-Based Authentication
   User authentication and access control mechanisms can be integrated to restrict data visibility based on user roles.
5. Cloud Database Integration
   Replacing SQLite with PostgreSQL or cloud-native databases would improve scalability and concurrent access support.
6. Query History and Logging
   Maintaining query logs and usage history would help in monitoring, auditing, and improving system performance.
7. Performance Optimization
   Caching frequently used queries and optimizing LLM response time can enhance overall system efficiency.
8. AI Safety Enhancements
   Advanced validation layers and rule-based SQL sanitization mechanisms can be implemented to further strengthen security.

These enhancements would transform IntelliSQL from a demonstration system into a production-ready intelligent database interface platform.

CONCLUSION

The IntelliSQL project successfully demonstrates the integration of large language models with relational database systems to enable natural language-based data retrieval. By leveraging Google Gemini 2.5 Flash, Streamlit, and SQLite, the system provides a seamless pipeline that converts English queries into executable SQL statements.

The project highlights the practical application of prompt engineering techniques, modular software architecture, and secure API integration. The implementation ensures that generated SQL queries are validated and restricted to safe operations, maintaining database integrity while leveraging AI-driven automation.

Through systematic testing, the application demonstrated accurate SQL generation across various filtering and aggregation scenarios. The modular design allows easy scalability, cloud deployment readiness, and future enhancements such as multi-table support and conversational query handling.

IntelliSQL represents a significant step toward intelligent database interfaces, showcasing how artificial intelligence can simplify complex technical tasks and improve accessibility for non-technical users. The project reflects modern AI engineering principles and aligns with real-world industry applications of large language models.