IntelliSQL – Intelligent SQL Querying with LLM's using Gemini

Chaitanya Agarwal (23BIT0060) *Debopam Bera (22BCE0522)*

Sai Mukesh (22BLC1232) *Tapaswee Dasari (22BRS1204)*

I. INTRODUCTION

In the era of digital transformation, data has become the lifeblood of organizations, driving critical business decisions and operational efficiency. The exponential growth of data volumes across industries—from finance and healthcare to retail and manufacturing—has made the ability to interact with databases a fundamental skill for professionals at all levels. Traditionally, Structured Query Language (SQL) has served as the standard interface for querying and managing relational databases. However, SQL's technical complexity and the necessity to understand database schemas pose significant barriers for non-technical users, such as business analysts, managers, and other stakeholders who need timely access to data but lack programming expertise.

To address these challenges, innovative solutions have emerged that leverage advancements in Natural Language Processing (NLP) and Large Language Models (LLMs). IntelliSQL exemplifies this new generation of tools by providing a user-friendly interface that translates plain English instructions into executable SQL commands using Google's Gemini API. By allowing users to express their data needs in natural language, IntelliSQL democratizes data access, streamlines data manipulation, and empowers a broader range of users to engage with organizational data directly and intuitively.

II. PROBLEM STATEMENT

Despite SQL's ubiquity and power, interacting with traditional SQL interfaces requires a deep understanding of:

- The underlying database schema (table and column names, relationships)
- SQL command syntax and execution nuances
- Error handling and debugging

For non-technical users, these requirements create a steep learning curve and often necessitate reliance on IT or data specialists, leading to bottlenecks and delayed insights. While graphical user interfaces (GUIs) offer some relief by providing visual tools for basic operations, they often lack the flexibility and scalability needed for complex or custom queries. As a result, organizations face a persistent gap between the data they possess and the ability of their teams to leverage it effectively.

The core challenge, therefore, is to design a system that can:

- Accurately interpret natural language queries, even when ambiguous or complex
- Map user intent to the correct database schema elements
- Generate syntactically and semantically correct SQL statements
- Execute these statements securely and efficiently
- Present results in an accessible, user-friendly format

Solution:

IntelliSQL aims to close this gap by integrating LLM-powered natural language interfaces with robust backend execution, making database interaction as simple as having a conversation.

III. OBJECTIVES

The primary objectives of the IntelliSQL project are:

- Web-Based Natural Language Interface: Develop an intuitive web application that enables users to input queries in plain English and receive corresponding SQL results.
- Comprehensive SQL Operation Support: Facilitate a wide range of SQL operations, including SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, and ALTER, thereby supporting both data retrieval and manipulation.
- Intuitive User Experience: Design a user interface that displays results in a clear, tabular format and provides easy navigation between different functionalities (Home, About, Query Assistant).
- Secure API Integration: Use environment variables (via dotenv) to securely manage API keys and sensitive configurations, ensuring best practices in application security.
- Schema-Agnostic Scalability: Architect the system to adapt dynamically to any database schema, allowing seamless integration with diverse datasets and future extensibility.

These objectives collectively ensure that IntelliSQL is not only accessible and powerful but also secure, flexible, and ready for enterprise-scale deployment.

IV. REQUIREMENTS ANALYSIS

A focused requirements analysis ensures IntelliSQL meets user and organizational needs effectively.

A. Functional Requirements:

- Natural Language to SQL: The system must accurately convert plain English queries into valid SQL statements.
- Comprehensive SQL Support: Support for SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, and ALTER commands.
- Schema Adaptability: Ability to work with any database schema without hardcoded mappings.
- User Feedback: Provide clear error messages and display results in an easy-to-read tabular format.
- Secure API Usage: Use environment variables to protect API keys and sensitive data.

B. Non-Functional Requirements:

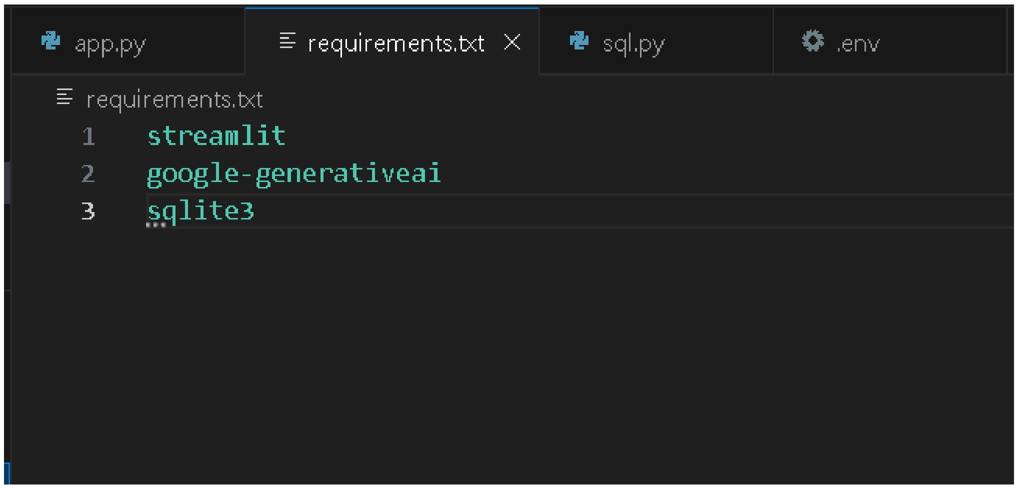
- Usability: The interface should be intuitive for non-technical users.
- Performance: Fast response times for query generation and execution.
- Security: Prevent SQL injection, restrict to single queries, and secure sensitive information.
- Scalability: Design should allow for future integration with larger or cloud databases.
- Reliability: Handle errors gracefully and maintain database integrity.

V. PROJECT DEVELOPMENT

The development of IntelliSQL was organized into distinct phases to ensure a systematic and efficient workflow from conception to deployment.

Phase 1: Requirement Gathering and Analysis

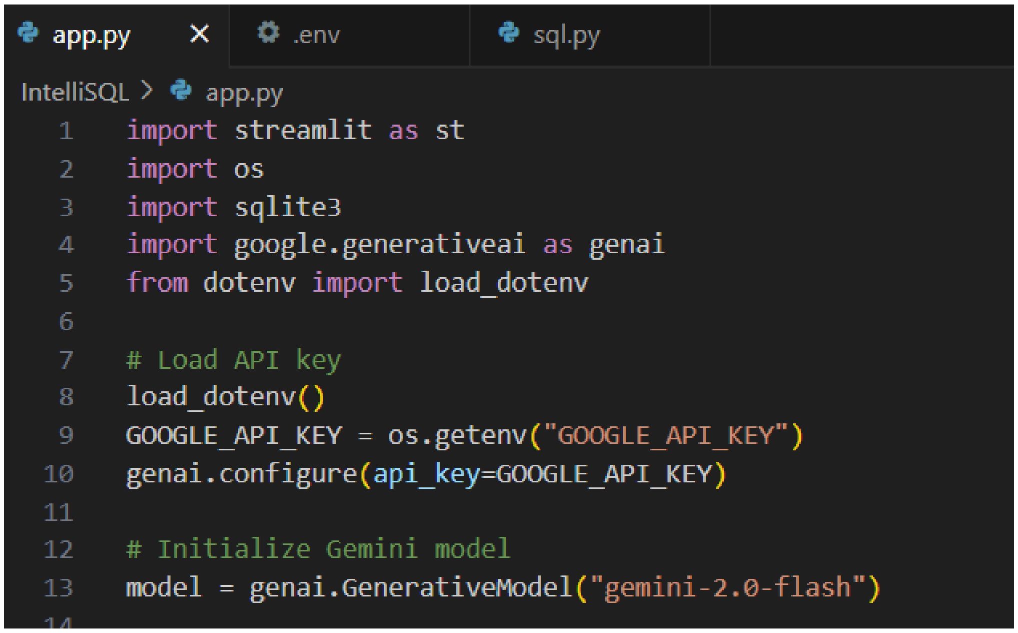
- Identified user needs and expectations through stakeholder meetings.
- Defined detailed functional and non-functional requirements.
- Analyzed similar solutions to benchmark features and spot improvement areas.



```
streamlit
google-generativeai
sqlite3
```

Phase 2: System Design

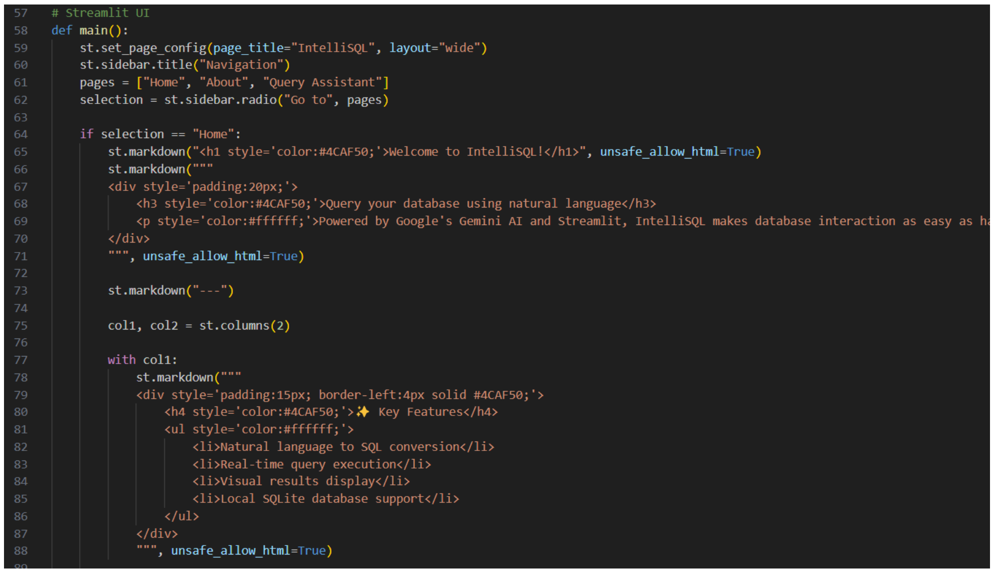
- Designed the overall architecture, specifying interactions between frontend, backend, LLM, and database.
- Developed UI wireframes and planned user experience flows.
- Established security protocols for API keys and sensitive data.



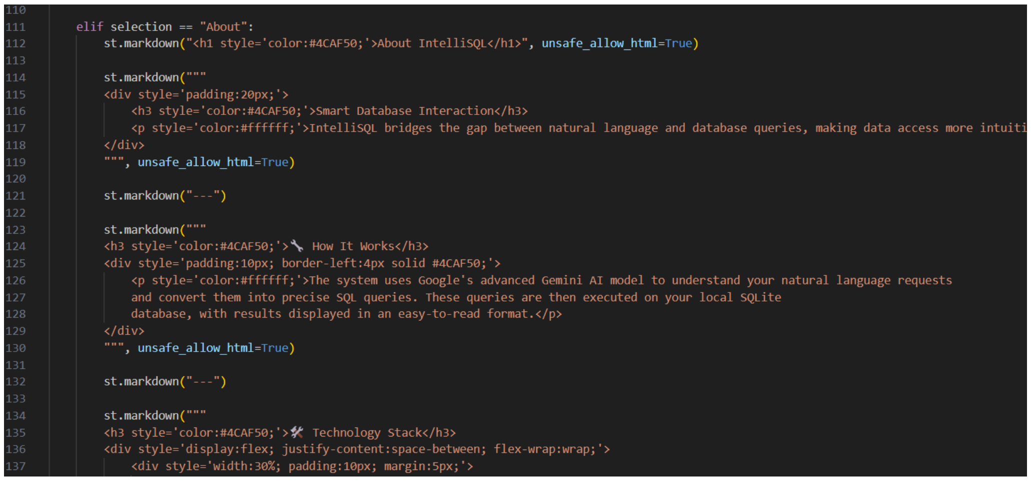
```python
import streamlit as st
import os
import sqlite3
import google.generativeai as genai
from dotenv import load_dotenv

# Load API key
load_dotenv()
GOOGLE_API_KEY = os.getenv("GOOGLE_API_KEY")
genai.configure(api_key=GOOGLE_API_KEY)

# Initialize Gemini model
model = genai.GenerativeModel("gemini-2.0-flash")
```

Phase 3: Implementation

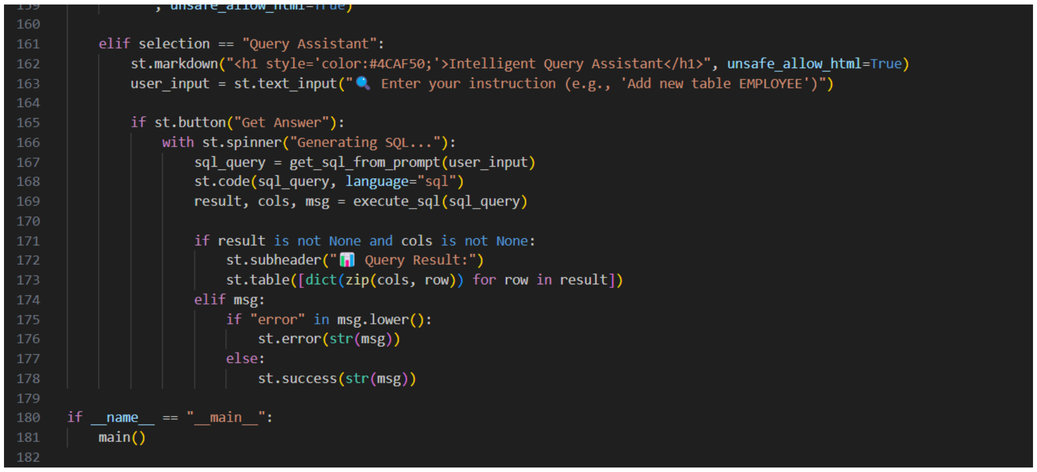
- Built the Streamlit-based frontend for user interaction.



```python
# Streamlit UI
def main():
    st.set_page_config(page_title="IntelliSQL", layout="wide")
    st.sidebar.title("Navigation")
    pages = ["Home", "About", "Query Assistant"]
    selection = st.sidebar.radio("Go to", pages)

    if selection == "Home":
        st.markdown("<h1 style='color:#4CAF50;'>Welcome to IntelliSQL!</h1>", unsafe_allow_html=True)
        st.markdown("""
        <div style='padding:20px;'>
            <h3 style='color:#4CAF50;'>Query your database using natural language</h3>
            <p style='color:#ffffff;'>Powered by Google's Gemini AI and Streamlit, IntelliSQL makes database interaction as easy as ha
        </div>
        """, unsafe_allow_html=True)

        st.markdown("---")

        col1, col2 = st.columns(2)

        with col1:
            st.markdown("""
            <div style='padding:15px; border-left:4px solid #4CAF50;'>
                <h4 style='color:#4CAF50;'>✨ Key Features</h4>
                <ul style='color:#ffffff;'>
                    <li>Natural language to SQL conversion</li>
                    <li>Real-time query execution</li>
                    <li>Visual results display</li>
                    <li>Local SQLite database support</li>
                </ul>
            </div>
            """, unsafe_allow_html=True)
```

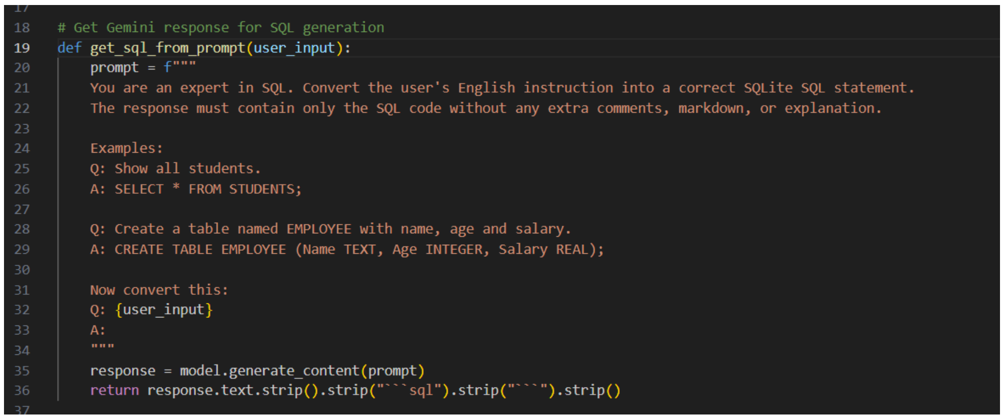


```python

    elif selection == "About":
        st.markdown("<h1 style='color:#4CAF50;'>About IntelliSQL</h1>", unsafe_allow_html=True)

        st.markdown("""
        <div style='padding:20px;'>
            <h3 style='color:#4CAF50;'>Smart Database Interaction</h3>
            <p style='color:#ffffff;'>IntelliSQL bridges the gap between natural language and database queries, making data access more intuiti
        </div>
        """, unsafe_allow_html=True)

        st.markdown("---")

        st.markdown("""
        <h3 style='color:#4CAF50;'>🔧 How It Works</h3>
        <div style='padding:10px; border-left:4px solid #4CAF50;'>
            <p style='color:#ffffff;'>The system uses Google's advanced Gemini AI model to understand your natural language requests
            and convert them into precise SQL queries. These queries are then executed on your local SQLite
            database, with results displayed in an easy-to-read format.</p>
        </div>
        """, unsafe_allow_html=True)

        st.markdown("---")

        st.markdown("""
        <h3 style='color:#4CAF50;'>🛠 Technology Stack</h3>
        <div style='display:flex; justify-content:space-between; flex-wrap:wrap;'>
            <div style='width:30%; padding:10px; margin:5px;'>
```



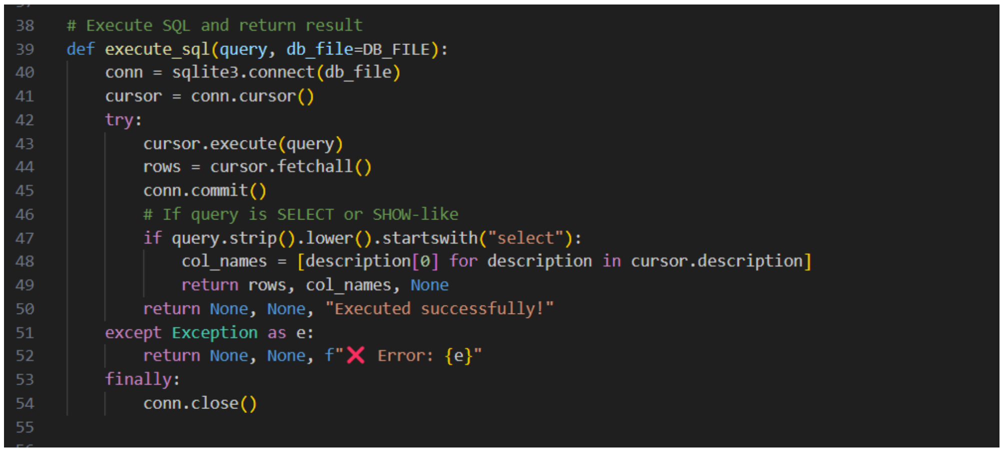
```python
            """, unsafe_allow_html=True)

    elif selection == "Query Assistant":
        st.markdown("<h1 style='color:#4CAF50;'>Intelligent Query Assistant</h1>", unsafe_allow_html=True)
        user_input = st.text_input("🔍 Enter your instruction (e.g., 'Add new table EMPLOYEE')")

        if st.button("Get Answer"):
            with st.spinner("Generating SQL..."):
                sql_query = get_sql_from_prompt(user_input)
                st.code(sql_query, language="sql")
                result, cols, msg = execute_sql(sql_query)

                if result is not None and cols is not None:
                    st.subheader("📊 Query Result:")
                    st.table([dict(zip(cols, row)) for row in result])
                elif msg:
                    if "error" in msg.lower():
                        st.error(str(msg))
                    else:
                        st.success(str(msg))

if __name__ == "__main__":
    main()

```

- Integrated Google Gemini API for natural language to SQL translation.



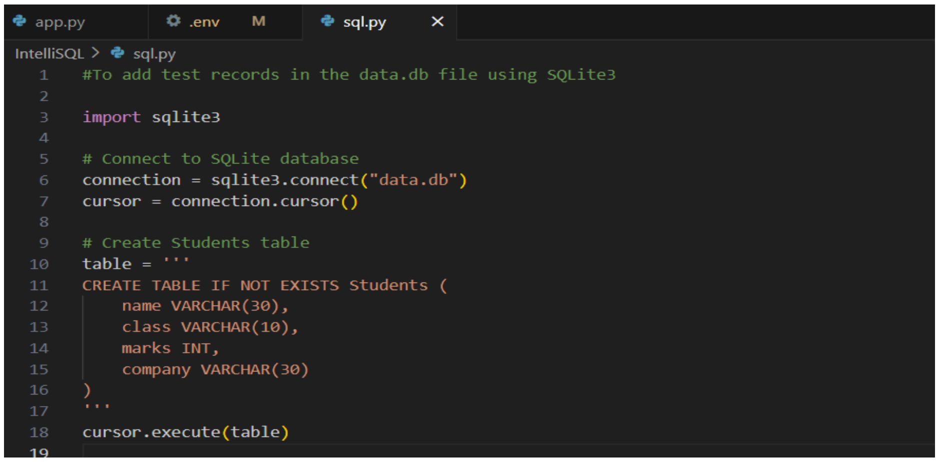
```python
# Get Gemini response for SQL generation
def get_sql_from_prompt(user_input):
    prompt = f"""
    You are an expert in SQL. Convert the user's English instruction into a correct SQLite SQL statement.
    The response must contain only the SQL code without any extra comments, markdown, or explanation.

    Examples:
    Q: Show all students.
    A: SELECT * FROM STUDENTS;

    Q: Create a table named EMPLOYEE with name, age and salary.
    A: CREATE TABLE EMPLOYEE (Name TEXT, Age INTEGER, Salary REAL);

    Now convert this:
    Q: {user_input}
    A:
    """
    response = model.generate_content(prompt)
    return response.text.strip().strip("```sql").strip("```").strip()
```

- Developed backend logic for SQL execution, error handling, and secure API usage.

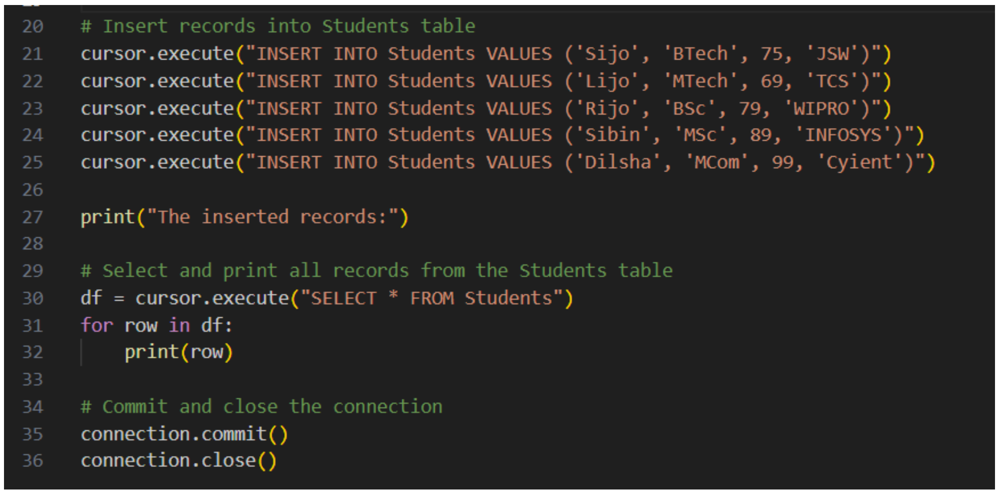


```python
# Execute SQL and return result
def execute_sql(query, db_file=DB_FILE):
    conn = sqlite3.connect(db_file)
    cursor = conn.cursor()
    try:
        cursor.execute(query)
        rows = cursor.fetchall()
        conn.commit()
        # If query is SELECT or SHOW-like
        if query.strip().lower().startswith("select"):
            col_names = [description[0] for description in cursor.description]
            return rows, col_names, None
        return None, None, "Executed successfully!"
    except Exception as e:
        return None, None, f"❌ Error: {e}"
    finally:
        conn.close()

```

- Connected the application to the SQLite database for persistent storage.



```python
#To add test records in the data.db file using SQLite3

import sqlite3

# Connect to SQLite database
connection = sqlite3.connect("data.db")
cursor = connection.cursor()

# Create Students table
table = '''
CREATE TABLE IF NOT EXISTS Students (
    name VARCHAR(30),
    class VARCHAR(10),
    marks INT,
    company VARCHAR(30)
)
'''
cursor.execute(table)
```



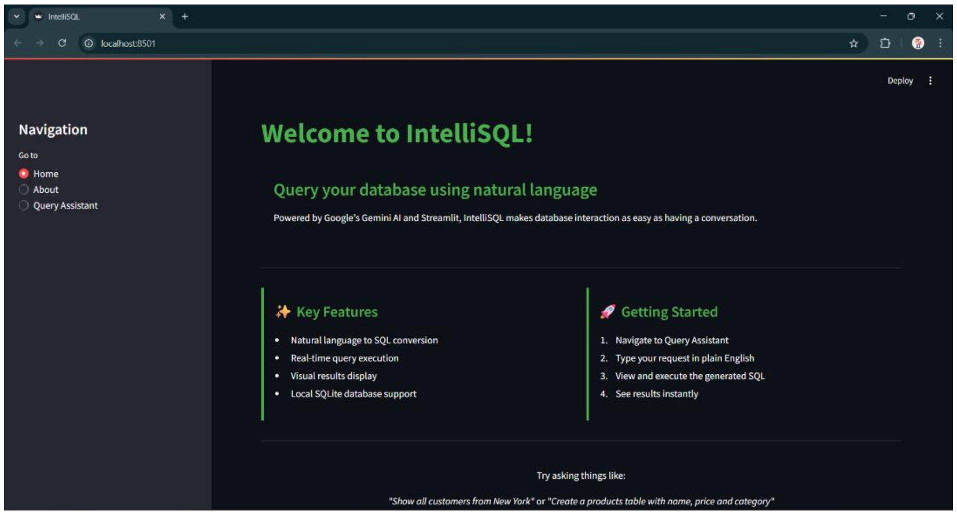
```python
# Insert records into Students table
cursor.execute("INSERT INTO Students VALUES ('Sijo', 'BTech', 75, 'JSW')")
cursor.execute("INSERT INTO Students VALUES ('Lijo', 'MTech', 69, 'TCS')")
cursor.execute("INSERT INTO Students VALUES ('Rijo', 'BSc', 79, 'WIPRO')")
cursor.execute("INSERT INTO Students VALUES ('Sibin', 'MSc', 89, 'INFOSYS')")
cursor.execute("INSERT INTO Students VALUES ('Dilsha', 'MCom', 99, 'Cyient')")

print("The inserted records:")

# Select and print all records from the Students table
df = cursor.execute("SELECT * FROM Students")
for row in df:
    print(row)

# Commit and close the connection
connection.commit()
connection.close()
```

Phase 4: Testing

- Conducted unit and integration testing of all modules.
- Performed functional testing with diverse natural language queries.
- Carried out usability testing with non-technical users to refine the interface.
- Fixed bugs and optimized performance based on feedback.

Phase 5: Deployment

- Deployed the application locally with all configurations and environment variables in place.
- Ensured data persistence and real-time updates in the SQLite database.
- Documented setup steps for future deployments and scalability.



Welcome to IntelliSQL!

Query your database using natural language

Powered by Google's Gemini AI and Streamlit, IntelliSQL makes database interaction as easy as having a conversation.

✨ Key Features

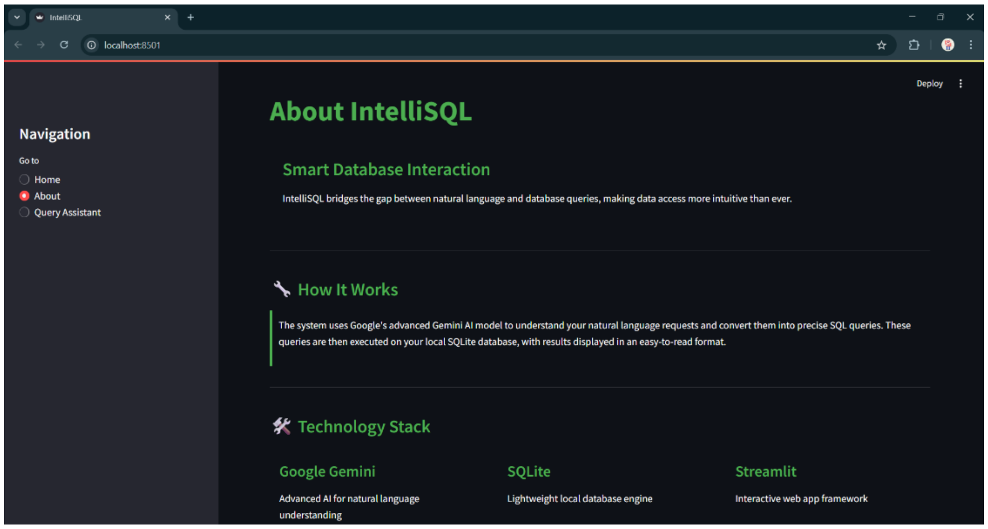
- Natural language to SQL conversion
- Real-time query execution
- Visual results display
- Local SQLite database support

🚀 Getting Started

1. Navigate to Query Assistant
2. Type your request in plain English
3. View and execute the generated SQL
4. See results instantly

Try asking things like:

"Show all customers from New York" or "Create a products table with name, price and category"



About IntelliSQL

Smart Database Interaction

IntelliSQL bridges the gap between natural language and database queries, making data access more intuitive than ever.

🔧 How It Works

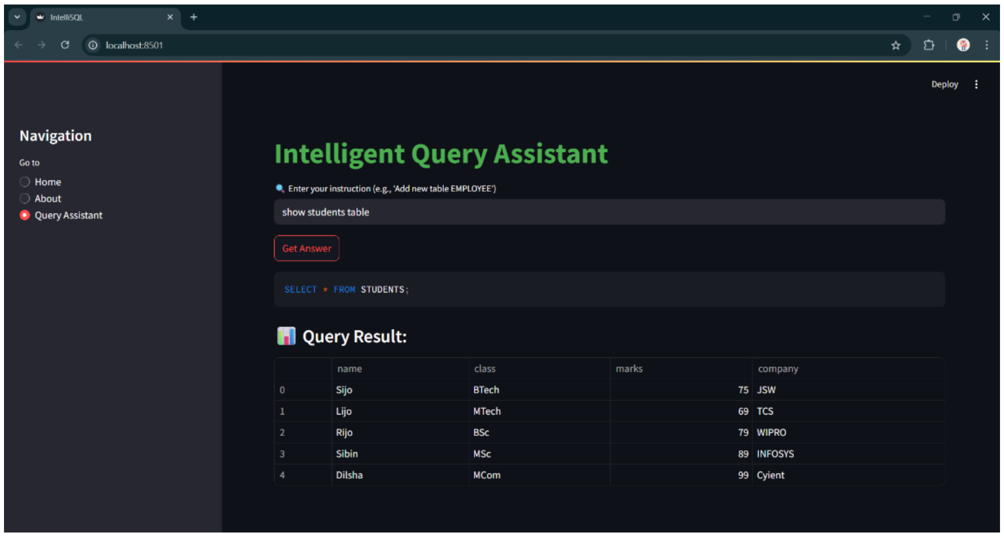
The system uses Google's advanced Gemini AI model to understand your natural language requests and convert them into precise SQL queries. These queries are then executed on your local SQLite database, with results displayed in an easy-to-read format.

🛠 Technology Stack

Google Gemini — Advanced AI for natural language understanding

SQLite — Lightweight local database engine

Streamlit — Interactive web app framework



Intelligent Query Assistant

🔍 Enter your instruction (e.g., 'Add new table EMPLOYEE')

show students table

```
SELECT * FROM STUDENTS;
```

📊 Query Result:

	name	class	marks	company
0	Sijo	BTech	75	JSW
1	Lijo	MTech	69	TCS
2	Rijo	BSc	79	WIPRO
3	Sibin	MSc	89	INFOSYS
4	Dilsha	MCom	99	Cyient

VI. TECHNOLOGY STACK

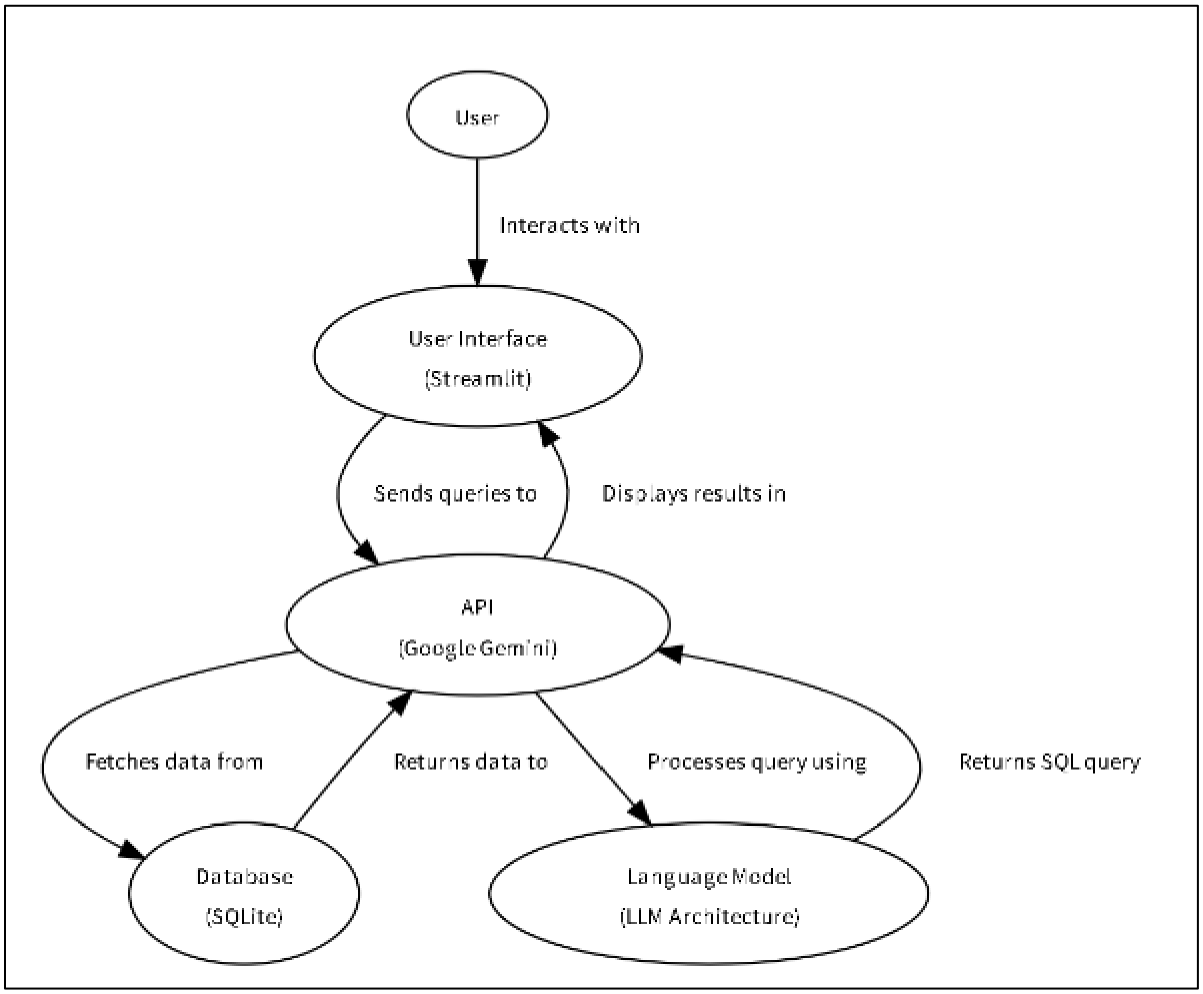
IntelliSQL is built using a modern, modular technology stack that balances simplicity, performance, and scalability:

- Frontend: Streamlit, a Python-based web framework, enables rapid development of interactive data applications with minimal code and a focus on user experience.
- Backend: Python scripts manage application logic and database interactions, leveraging SQLite for lightweight, file-based storage that is ideal for prototyping and small to mid-sized deployments.
- LLM Integration: Google Gemini API, accessed via the google.generativeai library, powers the natural language to SQL translation, providing state-of-the-art language understanding and generation capabilities.

- Security: The dotenv library manages sensitive API keys and configuration variables, keeping them out of the codebase and version control for enhanced security.
- Database: SQLite (data.db) offers a self-contained, serverless database solution that is easy to set up and maintain, making it suitable for local and embedded use cases.

This stack ensures that IntelliSQL is both robust and accessible, with the flexibility to evolve as requirements grow.

VII. SYSTEM ARCHITECTURE



IntelliSQL's architecture is designed for modularity, maintainability, and clear separation of concerns. The workflow proceeds through several well-defined stages:

1. User Input: The user enters a natural language query into the Streamlit web interface.

2. LLM Processing: The query is sent to the Gemini LLM, accompanied by a carefully engineered prompt that provides schema context and instructs the model to return clean SQL output.
3. SQL Generation: Gemini returns an SQL statement, which is parsed and sanitized to remove extraneous formatting or markdown.
4. SQL Execution: The backend executes the SQL command using SQLite, handling all standard operations while enforcing single-statement execution for security and compatibility.
5. Result Presentation: Query results (or success/error messages) are displayed in a dynamic, tabular format within the UI, providing immediate feedback to the user.
6. Error Handling: Any errors encountered during SQL generation or execution are caught, processed, and communicated in clear, non-technical language to help users refine their queries.

This architecture supports all major SQL operations and ensures that the application remains responsive, secure, and user-friendly.

VIII. IMPLEMENTATION DETAILS

- Model Selection: The `gemini-2.0-flash` model is chosen for its balance of speed and cost-effectiveness, ensuring rapid responses without excessive API usage.
- Secure Configuration: The `.env` file stores API keys and sensitive settings, which are loaded at runtime and excluded from version control to prevent accidental exposure.
- Persistent Storage: The SQLite database file (`data.db`) is updated in real time, reflecting changes from INSERT, UPDATE, and DELETE operations immediately.
- Error Management: The application includes robust error handling to catch syntax errors, invalid references, and attempts at multi-statement execution, providing actionable feedback to users.
- UI Structure: The Streamlit app is organized into three main tabs—Home (welcome and instructions), About (project overview), and Query Assistant (interactive querying)—with sidebar navigation for easy access.

These implementation choices ensure both developer productivity and a seamless end-user experience.

IX. KEY FEATURES

- Natural Language Querying: Users can interact with the database by typing questions or commands in plain English, without needing to know SQL or the underlying schema.

- Comprehensive SQL Support: The system supports a full range of SQL operations, from basic SELECTs to complex CREATE, DROP, and ALTER statements, making it suitable for both data exploration and database management.
- Error Feedback: When errors occur (e.g., invalid queries, missing tables), the system provides clear, human-readable messages to help users correct their input, reducing frustration and learning barriers.
- UI Navigation: A clean, multi-tab interface allows users to switch between informational pages and the interactive query assistant with ease.
- Live Execution: All changes to the database are applied instantly, and results are updated in real time, ensuring that users always see the most current data.
- Security and Privacy: API keys are managed securely, and only single-statement queries are executed to minimize the risk of SQL injection or accidental data loss.

These features collectively make IntelliSQL a powerful tool for democratizing data access within organizations.

X. CHALLENGES FACED

- Prompt Engineering: Crafting prompts that yield accurate, unambiguous SQL output from Gemini required iterative testing and refinement. The prompts must provide enough schema context and clear instructions to avoid markdown formatting or irrelevant information.
- Model Limitations: LLMs occasionally produce hallucinated output or include extraneous formatting, necessitating post-processing to extract clean SQL. Ensuring the model adheres strictly to the schema and user intent remains an ongoing challenge.
- Multi-Statement Handling: Since SQLite restricts execution to a single statement per query, the system must detect and reject multi-statement inputs, guiding users to break their requests into separate steps.
- Error Handling: Translating raw exceptions into user-friendly messages improves usability but requires careful mapping of error types to actionable advice.
- UI Design: Achieving an interface that is both visually appealing and highly functional required thoughtful design, balancing simplicity with the need to display complex results and error messages.

These challenges reflect broader issues in the field of natural language interfaces for databases, including ambiguity resolution, schema mapping, and robust error management.

XI. TESTING AND RESULTS

- Functional Testing: The application was tested with a diverse set of natural language queries, covering all major SQL operations and varying levels of complexity. Results showed high accuracy and reliability when queries were well-formed and within the model's training scope.
- Prompt Refinement: Iterative improvements to prompt engineering led to more consistent and accurate SQL generation, reducing the incidence of hallucinated or malformed queries.
- Usability Testing: Non-technical users were able to retrieve and manipulate data successfully with minimal instruction, validating the system's accessibility goals.
- Performance: For small to mid-sized SQLite databases, the system delivered rapid responses (typically within 1–3 seconds for query generation and execution), supporting real-time data exploration.

These results demonstrate IntelliSQL's effectiveness in bridging the gap between natural language and structured data access.

XII. CONCLUSION

IntelliSQL showcases the transformative potential of LLM-powered natural language interfaces for databases. By abstracting away the complexities of SQL and schema knowledge, the application enables users of all backgrounds to interact with data directly, fostering a culture of self-service analytics and data-driven decision-making. The integration of Google's Gemini API ensures state-of-the-art language understanding, while the modular architecture and secure design make IntelliSQL both robust and adaptable to evolving organizational needs.

As organizations continue to embrace data democratization, tools like IntelliSQL will play a crucial role in unlocking the full value of enterprise data assets, reducing bottlenecks, and empowering teams to make faster, more informed decisions.

XIII. FUTURE SCOPE

The roadmap for IntelliSQL includes several promising enhancements:

- Contextual Memory and Multi-Turn Conversations: Implementing memory capabilities will allow the system to maintain conversational context, enabling more complex, multi-step queries and follow-up questions.

- Cloud Database Integration: Expanding support to cloud-based databases such as MySQL, PostgreSQL, and BigQuery will broaden the application's applicability to enterprise environments.
- User Authentication and Session History: Introducing user accounts and query history tracking will enhance security, personalization, and auditability.
- Schema Introspection: Enabling the system to dynamically describe the available tables and columns will allow users to ask meta-questions like "What tables do I have?" or "Show me the columns in the orders table."
- Voice-to-Text Input: Adding speech recognition will further lower accessibility barriers, allowing users to interact with the database hands-free.
- Export Functionality: Supporting export of query results to CSV or Excel will facilitate offline analysis and reporting.

These enhancements will further solidify IntelliSQL's position as a leading solution for natural language database interaction, driving greater inclusivity and innovation in data-driven organizations.

By integrating the latest advances in AI, NLP, and user-centric design, IntelliSQL not only addresses current barriers to database access but also sets the stage for the next generation of intelligent, conversational data tools.

⁂