Assignment 3: Build-Your-Own SQL Database Agent (From Scratch)

Soorya Sivaramakrishnan

Dept. of Computer Engineering

Sardar Patel Institute of Technology

Mumbai, India

soorya.sivaramakrishnan22@apit.ac.in

The SQL Database Agent was implemented as a lightweight, interpretable system that allows a Large Language Model to interact safely with a SQLite database through a structured **ReAct (Reasoning → Acting → Observation)** loop.
The agent uses **Google Gemini 2.5 Flash** for reasoning and a modular codebase. A **synthetic dataset** was generated via *Mockaroo* with two relational tables (sample, emp, 1,000 rows each) sharing a common id column.
Testing progressed through four complexity levels: basic tool invocation, conditional SQL queries, arithmetic and edge-case reasoning, and multi-table joins.
Across all levels, the agent maintained read-only integrity and achieved approximately **82 % functional accuracy** with full traceability.

Reflection

The primary design trade-off in this project was between **complexity and simplicity**.
While a minimal implementation ensured clarity, interpretability, and fast execution, it also limited the agent's ability to handle more complex, multi-step reasoning tasks.
Queries that required chaining multiple conditions or integrating arithmetic and relational reasoning often exceeded the model's short-context reasoning capability.

A second trade-off involved **robustness versus flexibility**.
By designing all components from scratch, the system achieved transparency and control, but this came at the cost of reduced adaptability—especially when facing ambiguous or abstract natural-language queries.

During the experiment, a few key limitations and failures were observed:

1. **Mockaroo Data Generation Issues:**
The synthetic data generated through Mockaroo occasionally introduced inconsistencies such as mismatched field types or unrealistic value combinations. These errors, while minor, sometimes propagated downstream, causing model misinterpretations or failed query executions.

2. **Premature Reasoning Cut-offs:**
The model occasionally stopped reasoning too early and produced direct answers seemingly based on its pretrained biases rather than fully interpreting the context or reasoning through the steps. Attempts to make the prompts stricter did not consistently improve results — in fact, overly rigid prompts often degraded performance by constraining the model's flexibility and creativity.

3. **Sensitivity to Spelling and Syntax:**
   The model exhibited high sensitivity to even minor spelling or formatting variations. Small typographical errors (e.g., “employe” instead of “employee”) sometimes led to incorrect SQL generation or total failure to recognize the intent of the query.

Future iterations of this system will focus on:

1. **Using More Robust Models:**
   Exploring more advanced and stable models such as **Gemini 2.5 Pro**, which showed improved reasoning depth and tolerance to linguistic variation, though current testing faced **rate-limit constraints**. Once availability improves, it can offer a more resilient alternative.

2. **Enhanced Error-Handling and Normalization:**
   Introducing automated spell-correction, prompt-sanitization, and adaptive reasoning loops to mitigate small input inconsistencies and prevent premature answer generation.

3. **Refined Data Simulation:**
   Implementing validation layers on top of Mockaroo or switching to controlled data-generation pipelines to ensure schema consistency and better alignment with model expectations.