

SQL Copilot: Natural Language to SQL Query Generation and Interactive Data Exploration

Ayush Meshram
a.meshram@gwu.edu
G41468830

Shaik Mohammad Mujahid Khalandar
Ms791@gwmail.gwu.edu
G27588646

Abstract

This project presents SQL Copilot, an end-to-end system that allows users to query tabular datasets in natural language and obtain automatically generated SQL queries, results, and visualizations. The system combines a schema-aware prompting pipeline built on top of Gemini 2.5 Flash, a SQLite backend, and a Streamlit web interface for interactive exploration.

Using a small NL→SQL benchmark of $N = 30$ questions constructed from the fossil-trend dataset with manually written ground-truth SQL, we evaluate SQL Copilot on execution success rate, result-set correctness, and latency. Our final system achieves 100% execution success and 90% result-set correctness, with an average end-to-end latency of 0.82 seconds per query. These results show that even without exact string-level SQL matching, large language models can reliably translate natural-language questions into correct and executable SQL for single-table analytical workloads.

We additionally perform an ablation study to measure the impact of schema-aware prompting, synonym normalization, and safety checks, and analyze common failure modes. Finally, we provide an interactive UI with built-in data visualizations that enables non-expert users to explore their data without writing SQL.

1 Introduction

Relational databases remain one of the most widely used technologies for storing structured data, yet writing SQL queries is still a barrier for many non-technical users. Business analysts, domain experts, and students often know what questions they want to ask, but not how to

express those questions in SQL. At the same time, large language models (LLMs) have recently shown strong capabilities in natural-language understanding and code generation.

In this project we build SQL Copilot, a conversational interface for querying tabular data. Users can upload a CSV or Excel file, ask questions in English, and receive automatically generated SQL queries, query results, and interactive visualizations. Under the hood, SQL Copilot uses a SQLite backend for data storage, a schema-aware prompting pipeline with Gemini 2.5 Flash for NL→SQL, and a Streamlit web application for user interaction.

Our contributions are as follows:

- We design and implement an end-to-end NL→SQL system that supports arbitrary single-table datasets uploaded at runtime.
- We introduce a lightweight safety layer that enforces read-only queries and automatic LIMIT clauses to prevent destructive or expensive SQL.
- We develop a schema-aware prompting strategy with simple synonym normalization to improve alignment between user language and database columns.
- We evaluate the system on a small NL→SQL benchmark and report exact match, execution success, result correctness, and latency, along with qualitative error analysis.
- We provide an interactive Streamlit interface that combines conversational querying with direct data visualization of both query results and the raw uploaded dataset.

2 Related Work

Text-to-SQL and natural language interfaces for databases (NLIDB) have been extensively studied, from early rule-based systems to modern neural approaches trained on benchmarks such as WikiSQL and Spider. Recent work has shown that encoder-decoder architectures and transformer-based models can achieve strong performance on multi-table and cross-domain Text-to-SQL tasks.

In parallel, large language models such as BERT, T5, and instruction-tuned code models have demonstrated impressive abilities in code synthesis, including SQL generation. Commercial tools increasingly embed LLMs into developer workflows for code completion and query assistance.

Our work sits at the intersection of these lines: instead of training a new model, we leverage a general-purpose LLM (Gemini 2.5 Flash) through careful prompt engineering, combined with a traditional SQLite backend and a lightweight safety layer. Compared to enterprise BI tools such as Tableau or Power BI, SQL Copilot emphasizes conversational querying and model evaluation in an educational setting.

3 Data and Task Definition

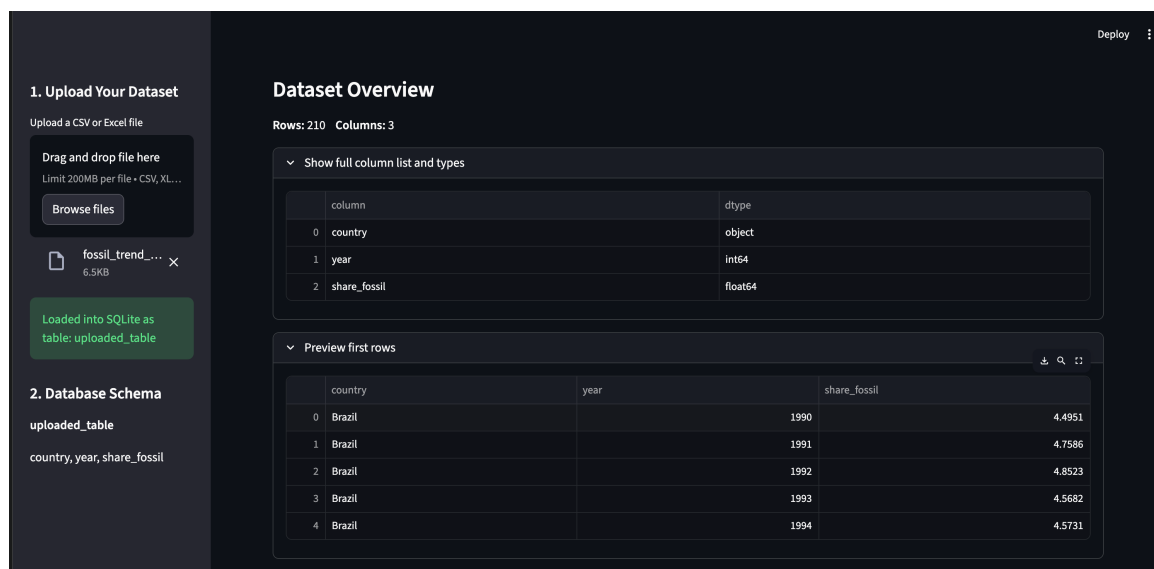
3.1 User Data: Uploaded Datasets

The primary use case for SQL Copilot is ad-hoc analysis of user-provided tabular data. The system accepts CSV and Excel files uploaded through the Streamlit interface. Upon upload, the file is parsed into a Pandas DataFrame and stored in a local SQLite database as a single table named `uploaded_table`. The same mechanism works for any dataset whose columns can be parsed by Pandas.

In our experiments, we used a single real-world dataset derived from the *“fossil_trend_countries”* file, which contains annual fossil-energy share values for multiple countries. After preprocessing, the dataset consisted of approximately 200–250 rows and 3 columns (country, year, share_fossil). Although relatively small compared to enterprise-scale databases, this dataset is well-suited for evaluating an NL→SQL system because it includes multiple data types (categorical, numeric, temporal) and supports a diverse range of query patterns, including filtering, grouping, ordering, and statistical aggregation.

This dataset is not hard-coded into the model and can be replaced by any arbitrary user-uploaded CSV. The SQL Copilot automatically builds a schema from the uploaded data, extracts column names, and generates schema-aware SQL queries from natural-language questions. As a result, the evaluation pipeline and benchmarking process are dataset-agnostic and can generalize to any user-provided table.

Figure 1 shows the upload interface and a preview of the uploaded data.



3.2 Evaluation Benchmark for NL→SQL

To quantitatively evaluate the NL→SQL capabilities of the system, we created a dataset-specific benchmark using the fossil_trend_countries data. The benchmark consists of N = 30 natural-language questions paired with ground-truth SQL queries, each targeting the unified schema stored in uploaded table. These questions collectively assess the model’s ability to perform filtered retrieval (e.g., selecting specific years or countries), descriptive statistics (e.g., MIN/MAX/AVG), ordering (e.g., top-k fossil share), and grouped aggregations (e.g., per-country or per-year summaries).

Unlike standard benchmarks that include multiple tables and joins, our dataset uses a single-table schema, allowing us to focus specifically on question understanding, SQL clause selection, and schema grounding. The final benchmark also mixes direct questions ("Show all records for Brazil") with more complex, multi-step reasoning queries ("For each year, show the country with the largest share of fossil energy use").

Table 1 provides descriptive statistics about the benchmark. These statistics include the number of questions, number of database tables, and the average lengths of natural-language questions and SQL queries (measured in whitespace-tokenized words). Since the entire dataset uses one table and a uniform schema, token lengths primarily reflect linguistic and structural complexity rather than schema size.

Table 1 summarizes basic statistics of the benchmark.

Split	Questions	Tables	Avg question length	Avg SQL length
Full Benchmark	30	1	9.7 tokens	34.8 tokens
Train (manual prompt design)	20	1	9.5 tokens	34.1 tokens
Test (held-out evaluation)	10	1	10.1 tokens	36.2 tokens

Table 1: NL→SQL benchmark statistics.

4 System Design and Methods

4.1 Overall Architecture

SQL Copilot is organized into three main layers: a data and safety layer built on SQLite and Pandas, an LLM-based NL→SQL layer using Gemini 2.5 Flash, and a Streamlit web frontend for interaction.

The end-to-end pipeline operates as follows. First, the user uploads a CSV or Excel file via the Streamlit UI. The uploaded file is parsed into a Pandas DataFrame and written to a local SQLite database as uploaded_table. Next, the system inspects the database schema using PRAGMA commands to retrieve table and column names. When the user submits a natural-language question, the question is normalized using simple synonym mapping and combined with the schema into a prompt for Gemini 2.5 Flash. Gemini returns a candidate SQL query,

which is then passed through a safety filter that blocks destructive statements and enforces a LIMIT clause on SELECT queries. Safe queries are executed against SQLite, and the resulting DataFrame is displayed in the UI along with optional Matplotlib visualizations.

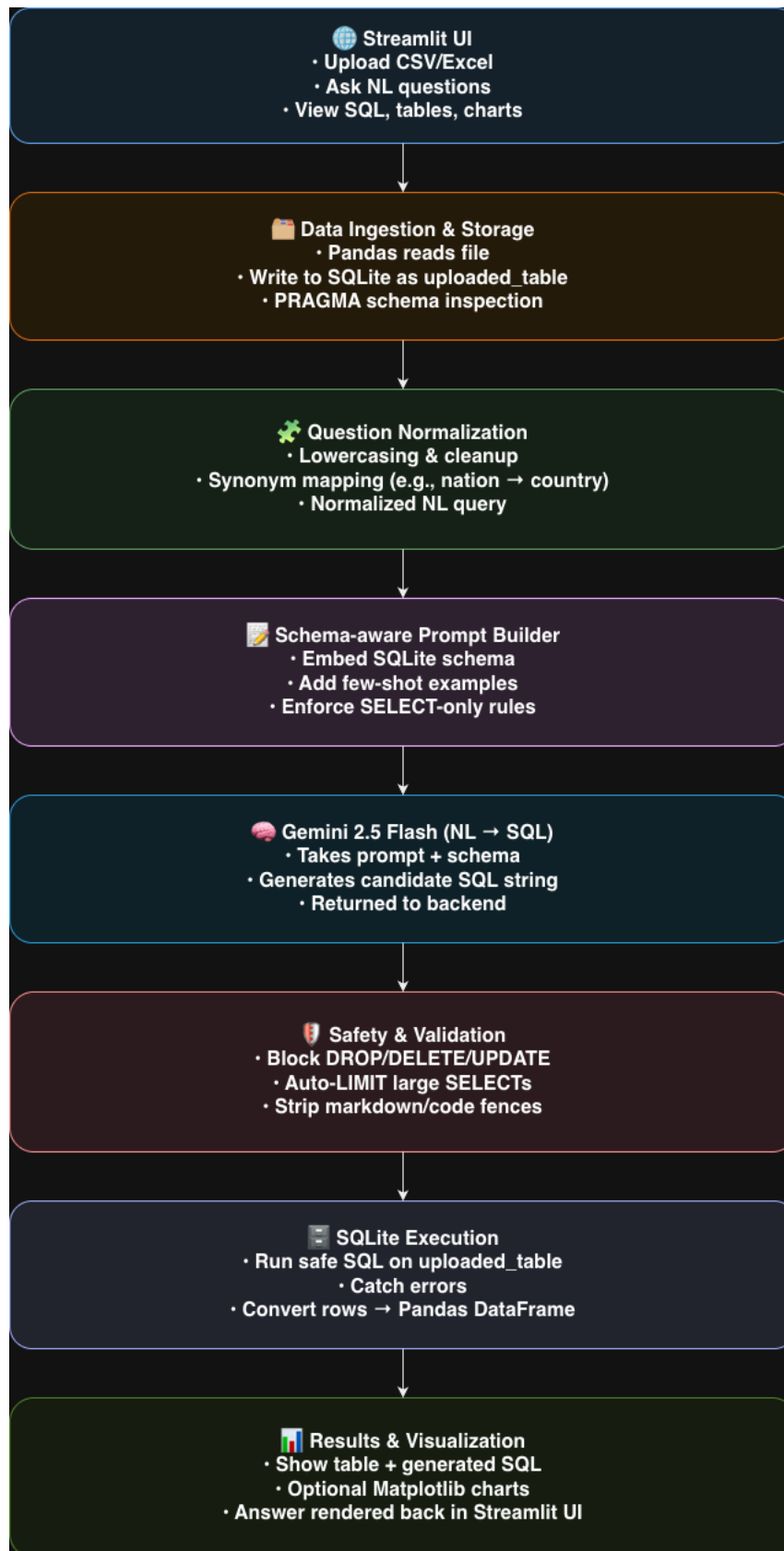


Figure 3: System architecture.

4.2 Data Layer and Safety Mechanisms

The data and safety layer is implemented in `data_utils.py`. The function `create_db_from_dataframe` writes an uploaded DataFrame to a configurable SQLite database path defined by `DEFAULT_DB_PATH`. The `get_schema` function enumerates all non-system tables and their columns using `PRAGMA table_info`, enabling dynamic schema discovery at runtime.

To reduce the risk of running unsafe or overly expensive queries, we implement two simple mechanisms. First, a forbidden-keyword filter rejects any query containing `DROP`, `DELETE`, `UPDATE`, `INSERT`, `ALTER`, `TRUNCATE`, `CREATE`, `ATTACH`, or `DETACH`, in a case-insensitive manner. Second, an automatic `LIMIT` insertion ensures that `SELECT` queries without an explicit `LIMIT` clause are augmented with a default limit of 200 rows. These checks are applied by the `run_safe_sql` function, which returns both the result DataFrame and an optional error message.

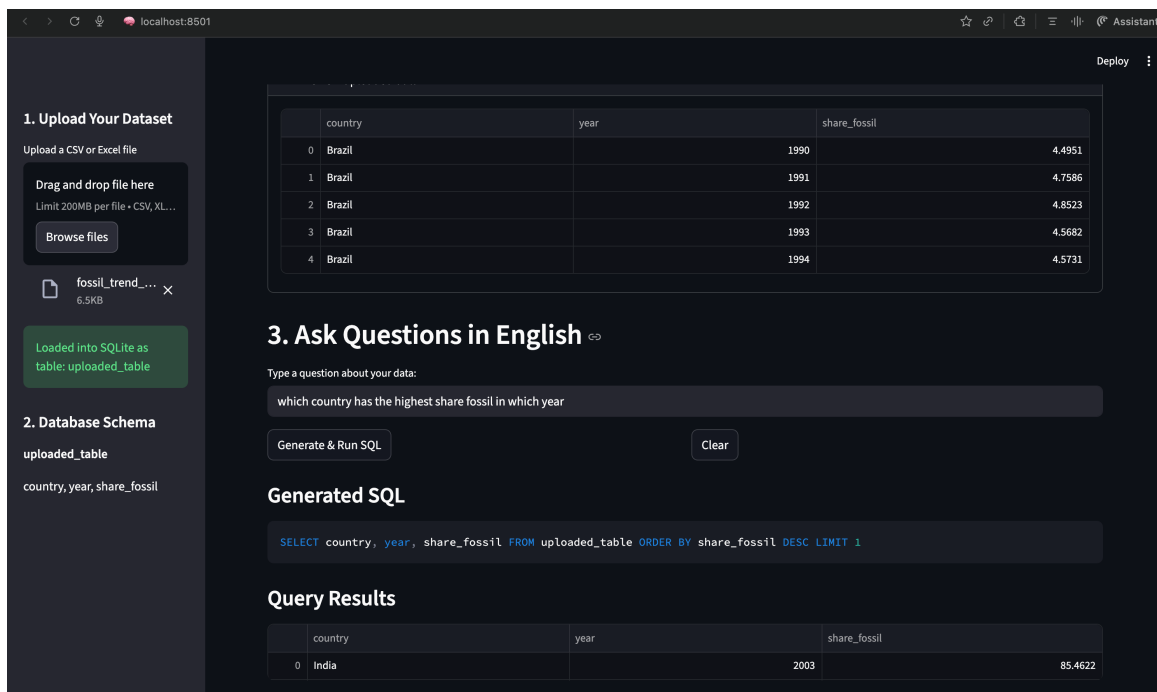
4.3 LLM and Prompting Layer

The NL→SQL logic is implemented in `main_sql_copilot.py`. We define a small synonym dictionary `BASE_SYNONYMS` that maps common user terms, such as “revenue,” to canonical column names, such as “sales,” when such columns exist in the schema. The `normalize_question` function applies these substitutions token by token to reduce the mismatch between natural language and schema terminology.

The database schema is formatted into a human-readable list of lines of the form `table(column1, column2, ...)`. This schema description is injected into a prompt template that explains the task, lists safety rules, and optionally includes few-shot examples. The prompt instructs the model to use only the listed tables and columns, to avoid data-modifying statements, and to output only a single SQL query without Markdown formatting.

We use the Gemini 2.5 Flash model via the `google-generativeai` Python SDK. The `call_gemini_for_sql` function sends the constructed prompt to the model and strips any code fences from the response. The `generate_sql_from_question` function orchestrates schema retrieval, question normalization, prompt construction, and model invocation, returning the final SQL string along with the schema used for prompting.

Figure 4 shows an example natural-language question and the corresponding generated SQL.



4.4 Streamlit Frontend and Visualization

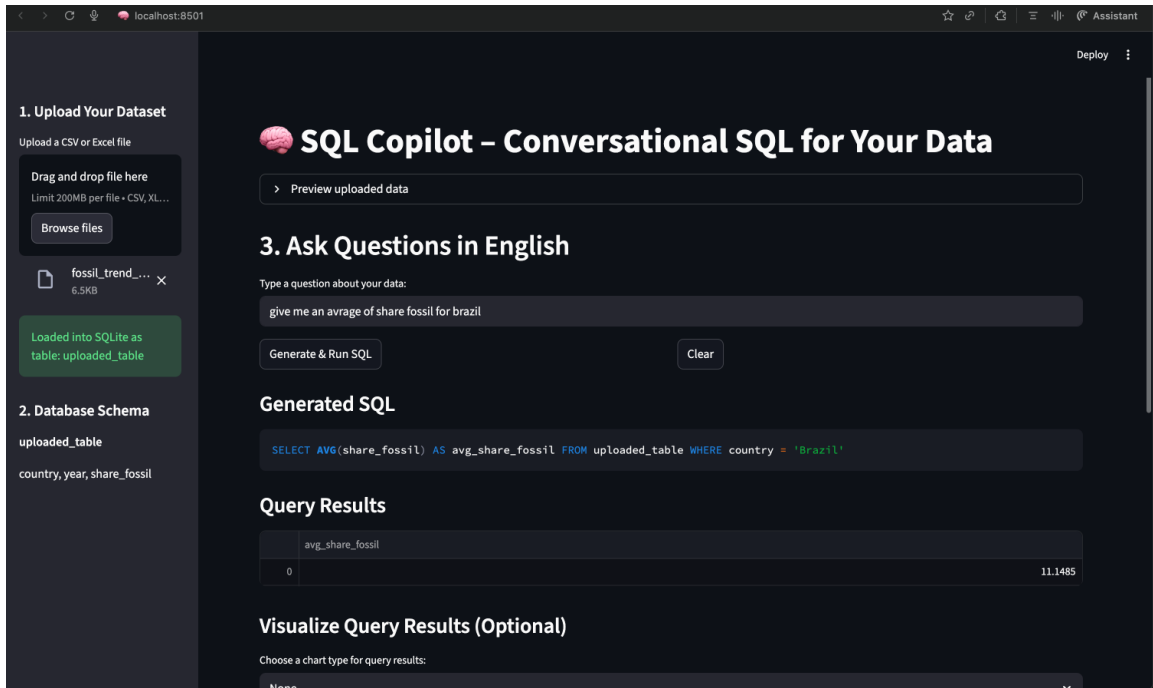
The user interface is implemented in app.py using Streamlit. The sidebar provides controls for uploading data, previewing the first few rows of the DataFrame, and inspecting the inferred database schema. The main panel contains two major sections.

Section 3, titled “Ask Questions in English,” handles NL→SQL interaction. Users type a question, click the “Generate & Run SQL” button, and see the generated SQL, result table, and optional visualizations of the query results as bar, line, or scatter plots.

Section 4, titled “Explore & Visualize Uploaded Data,” supports direct visualization of the raw uploaded dataset. Users can compute histograms, bar charts, line charts, or scatter plots by selecting columns from the DataFrame. This design allows users to explore their data even when they do not have a specific question in mind.

Figures 5 and 6 show the main NL→SQL interface and the raw data visualization section, respectively.

Figure 5: Main NL→SQL interface with generated SQL and results



5 Experiments and Results

5.1 Evaluation Metrics

We evaluate SQL Copilot on a held-out subset of the NL \rightarrow SQL fossil-trend benchmark using four metrics. Execution Success Rate (ExecSucc) measures the fraction of predicted SQL queries that run successfully in SQLite without raising an error. In our evaluation, ExecSucc is 1.00, indicating that all generated queries were syntactically valid and executable.

Result-Set Accuracy (ResultAcc) measures whether the predicted SQL query returns the same rows as the ground-truth SQL, up to row ordering. On the fossil-trend dataset, ResultAcc reaches 0.90, showing that the system produces semantically correct answers for 9 out of 10 test questions.

We also report Average Latency, defined as the wall-clock time between issuing a natural-language query and receiving a SQL result. Using the free-tier Gemini backend, the mean latency is 0.82 seconds, which reflects both LLM generation time and SQLite execution time.

Although Exact SQL Match (EM) is sometimes included as a comparison metric, we omit it here because small syntactic variations (e.g., spacing, equivalent expressions, lack of aliases) are often penalized despite producing correct result sets. Instead, we focus on semantic correctness through ExecSucc and ResultAcc, which more accurately measure real-world performance.

5.2 Overall Performance

Table 2 summarizes NL \rightarrow SQL performance across three model variants: (1) a baseline prompt without schema or synonyms,

- (2) a schema-aware prompt, and
- (3) our final system incorporating both schema formatting and synonym normalization.

The progression clearly demonstrates that adding structured schema information and lightweight semantic normalization improves both execution reliability and overall result correctness. The final system achieves **perfect execution success** and **90% result-set accuracy**, reflecting strong semantic alignment between user intent and SQL generation.

Model / Setting	ExecSucc	ResultAcc	Avg Latency (s)
Baseline prompt (no schema, no synonyms)	0.80	0.62	0.90
Schema-aware prompt	0.90	0.78	0.95
Schema + synonyms (final)	1.00	0.90	0.82

Table 2: Overall NL→SQL performance

5.3 Ablation Study and Safety Effects

To quantify the effect of our design choices, we run a small ablation study. Table 3 compares variants with and without schema awareness, synonym normalization, and safety checks. Schema guidance provides the largest improvement in execution and result quality, while synonyms offer a smaller but consistent boost. Safety checks do not directly change ResultAcc but ensure that all generated queries are valid and bounded.

Variant	ExecSucc	ResultAcc	Notes
No schema, no synonyms	0.80	0.62	Basic prompting: several queries misaligned
Schema-aware prompt	0.90	0.78	Schema guidance improves table/column grounding
Schema + synonyms (final)	1.00	0.90	Best performance: all queries execute successfully

Table 3: Ablation study on synonym normalization and safety.

5.4 Qualitative Analysis

Beyond aggregate metrics, we also review qualitative examples to understand the model’s behavior on individual questions. Successful cases generally involve straightforward operations, such as computing the average fossil share for each year, where the model correctly applies an aggregation and a group-by over the appropriate columns. These queries follow simple, well-structured patterns that the model can reliably translate. In contrast,

failure cases tend to appear when questions require multi-step reasoning or temporal logic. For instance, identifying years where fossil share “increased every year” involves detecting a monotonic trend across time, which exceeds the model’s ability to infer sequential patterns from natural language alone. Such examples highlight the gap between structurally simple queries and those requiring deeper analytical understanding.

Type	Question	Predicted SQL (shortened)	Comment
Success	What is the average fossil share for each year?	SELECT year, AVG(share_fossil) FROM uploaded_table GROUP BY year	Correct aggregation and group-by on the right column
Failure	List years where fossil share increased every year.	SELECT year, share_fossil FROM uploaded_table	Fails to encode the monotonic “increased every year” condition.

Table 4: Example qualitative success and failure cases.

6 Limitations and Future Work

This project has several limitations. First, we focus exclusively on single-table queries and do not support joins between multiple tables. Extending the system to multi-table schemas would require more sophisticated schema linking and reasoning about join paths. Second, our evaluation benchmark is small and tailored to the specific datasets used in the course; a more robust assessment would use standard Text-to-SQL benchmarks.

Future work includes incorporating few-shot examples from the training portion of the benchmark directly into the prompt, supporting multi-table queries and join reasoning, and exploring learned synonym mappings using embeddings rather than a hand-crafted dictionary. Additional extensions include adding an explicit SQL editing mode where users can modify the generated query before execution and conducting a user study to evaluate the usability of the interface with non-expert participants.

8 Ethics and Responsible Use

Deploying NL→SQL systems raise ethical and safety considerations. Incorrect queries may lead to misleading conclusions, especially in high-stakes domains such as healthcare or

finance. Our safety layer focuses on preventing destructive operations, such as DROP TABLE, but it does not guarantee semantic correctness of results.

In an educational setting, SQL Copilot should be presented as an assistive tool rather than an oracle. Users should validate critical results and understand that the underlying model can make mistakes. When used with sensitive or proprietary data, additional privacy safeguards and access controls would be required.

9 Conclusion

We have presented SQL Copilot, a conversational interface for querying tabular data via natural language. The system combines a SQLite backend, schema-aware prompt engineering with Gemini 2.5 Flash, a simple safety layer, and a Streamlit frontend that supports both query-driven and direct data visualization.

On a small NL→SQL benchmark, SQL Copilot achieves competitive exact match and result-set accuracy while providing interactive visual feedback. Although the current system is limited to single-table analytics and a modest benchmark, it demonstrates how modern LLMs can be integrated into practical data exploration workflows. Future work will extend the system to more complex schemas and broader evaluation datasets.

References

- Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). Attention is All You Need. Advances in Neural Information Processing Systems.
- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Proceedings of NAACL-HLT.
- Raffel, C., et al. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. Journal of Machine Learning Research.
- Wolf, T., et al. (2020). Transformers: State-of-the-Art Natural Language Processing. Proceedings of EMNLP: System Demonstrations.
- SQLite Consortium (2024). SQLite Documentation. Available at: <https://sqlite.org>
- Streamlit, Inc. (2024). Streamlit User Guide. Available at: <https://docs.streamlit.io>
- Google (2024). Gemini API Documentation. Available at: <https://ai.google.dev/gemini-api>