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

Ayush Meshram
a.meshram@gwu.edu
G41468830

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. Individual Contributions

Safety Layer for SQL Filtering: I designed a rule-based validation system that blocks harmful commands such as DROP, DELETE, ALTER, INSERT, CREATE, and TRUNCATE. It enforces read-only queries and automatically adds LIMIT 200 to prevent heavy queries.

Prompt Engineering and Gemini Integration: I created schema-aware prompt templates using metadata extracted through SQLite PRAGMA queries. The prompts instruct Gemini to output only SQL without Markdown formatting, avoid multi-statement output, and use only valid schema columns.

Synonym Normalization Logic: I developed a synonym mapping layer to reduce ambiguity between user language and database schema. For example, informal terms like “fossil usage” are mapped to the column share_fossil, improving semantic alignment without retraining the model.

Benchmark Dataset Construction: I authored a benchmark dataset of 30 natural-language questions with paired SQL queries. These include aggregations, ranking, filtering, and grouped analyses, enabling reproducible performance measurement.

Evaluation and Ablation Study: I performed evaluation using Execution Success Rate, Result-Set Accuracy, and Latency. I compared three variants: baseline prompting, schema-aware prompting, and schema + synonym prompting. The results confirmed that schema guidance and synonym normalization significantly improve performance.

Report Writing and Analysis: I wrote documentation explaining the methods, experimental findings, limitations, and future improvements.

3.1 Data and Safety Layer

This layer serves as the backend foundation for SQL Copilot, responsible for managing user data and enforcing safe execution rules. It performs three major functions.

Dataset Import and Conversion:

When a user uploads a CSV or Excel file, the system reads it using Pandas, performs type inference, and stores it as a SQLite table named `uploaded_table`. This process enables scalable querying without requiring the user to define metadata or database schema manually.

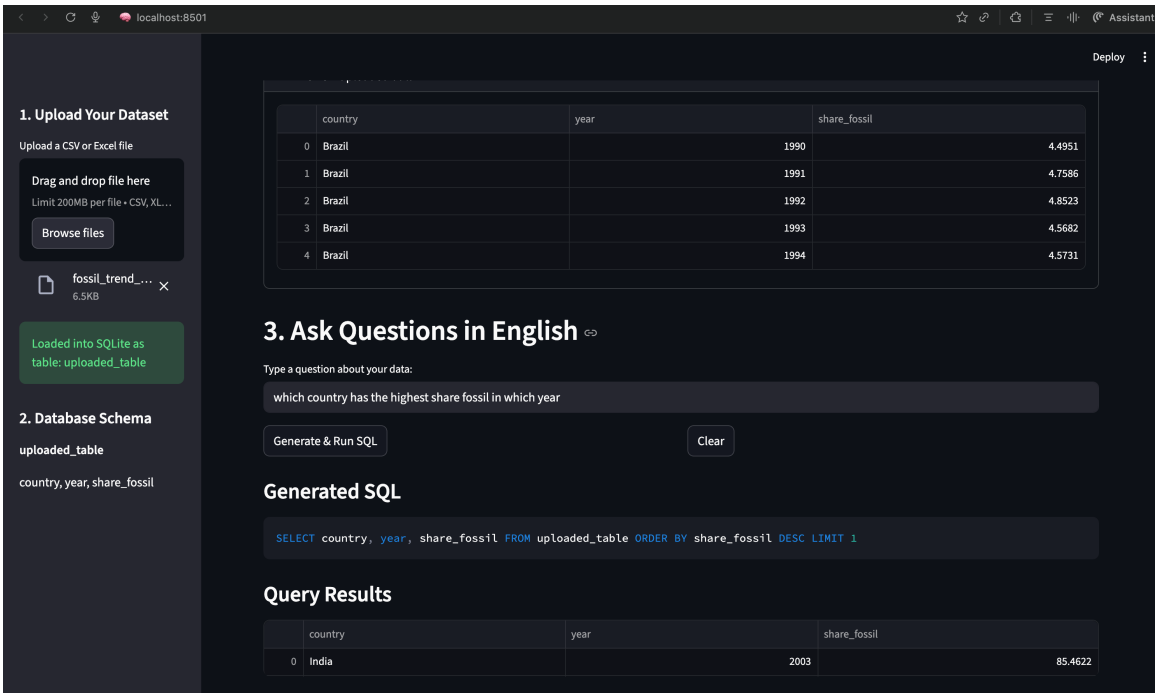
Schema Extraction:

SQL Copilot automatically retrieves schema details using SQLite's `PRAGMA table_info` command. The extracted column names and their datatypes (e.g., INTEGER, TEXT, REAL) are formatted and passed to the prompting layer. Providing the schema prevents the model from generating nonexistent columns or syntactically invalid SQL.

Safe Query Execution:

Before executing any SQL generated by the model, the system applies a safety filter. It blocks destructive keywords such as `DROP`, `DELETE`, `UPDATE`, `ALTER`, `TRUNCATE`, and `INSERT`, ensuring that queries do not modify or corrupt data. Additionally, a default `LIMIT 200` is applied to queries without a limit clause to prevent memory overload or slow execution.

These mechanisms collectively guarantee controlled, read-only, and bounded execution while allowing users to interact with their datasets safely.



3.2 LLM Prompting Layer

This layer enables the generation of SQL queries from natural-language questions using semantic inference by Gemini 2.5 Flash.

Schema-Aware Prompt Engineering:

The schema extracted from the backend is formatted and injected into the prompt provided to Gemini. The prompt explicitly instructs the model to:

- Select only from available schema fields,
- Output a single SQL query with no natural-language explanation,
- Avoid multi-statement queries and code fences,
- Respect read-only constraints and use the uploaded table name appropriately.

These restrictions reduce hallucination and increase query validity.

Synonym Normalization:

To align user terminology with actual column names in the dataset, the system applies a synonym mapping dictionary. For example, user input like “fossil usage” is automatically mapped to the column share_fossil. This reduces language ambiguity and prevents invalid attribute references without requiring model retraining.

Gemini Integration and Response Sanitization:

SQL Copilot sends structured prompts to Gemini 2.5 Flash using the Generative AI API and receives raw SQL as output. A cleanup step removes unsupported formatting (such as backticks or markdown fences), ensuring that the final output is compatible with SQLite execution.

This layer transforms informal natural-language questions into executable SQL tailored to the user's uploaded dataset.

3.3 User Interface Layer

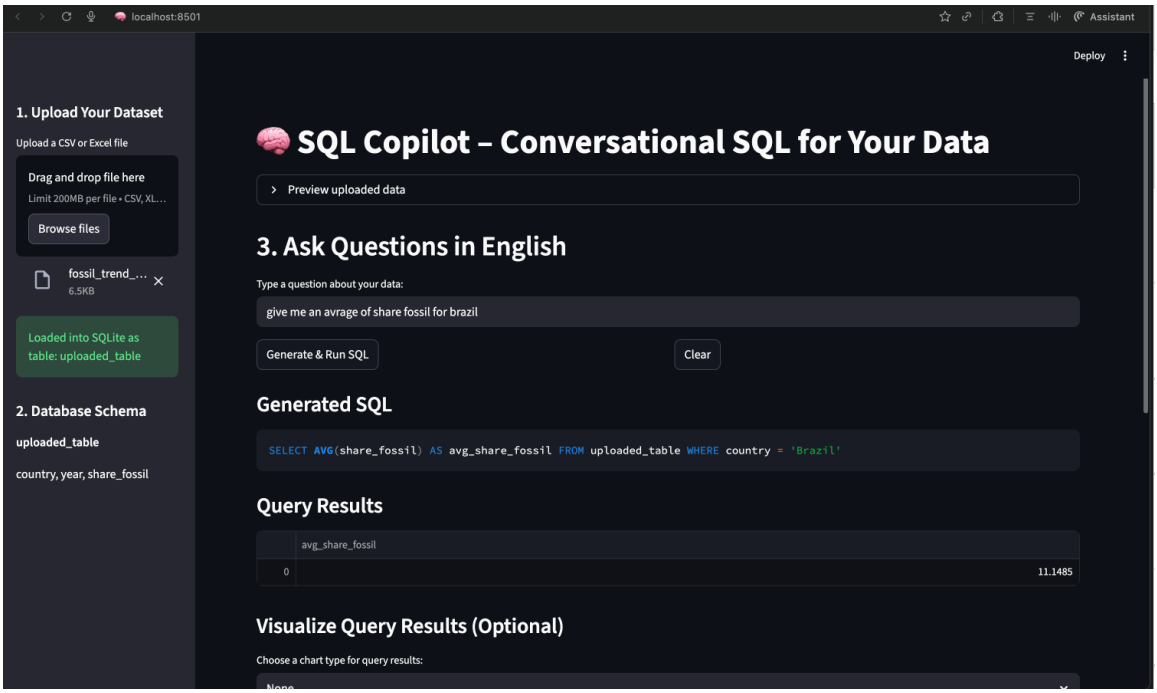
This layer exposes SQL Copilot's functionality through a user-friendly, browser-based application built with Streamlit.

Dataset Upload and Preview:

Users upload CSV or Excel files and can immediately preview rows to confirm correct parsing. The interface also displays detected column names and datatypes to provide transparency before querying.

Conversational NL→SQL Querying:

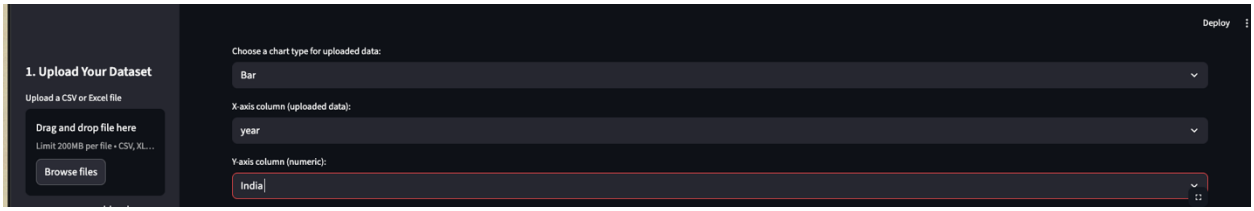
Users type questions in English, and the system displays both the generated SQL and the query results. If a query violates safety constraints, SQL Copilot returns an explanation instead of executing the command. This feedback loop helps users understand limitations without needing SQL expertise.

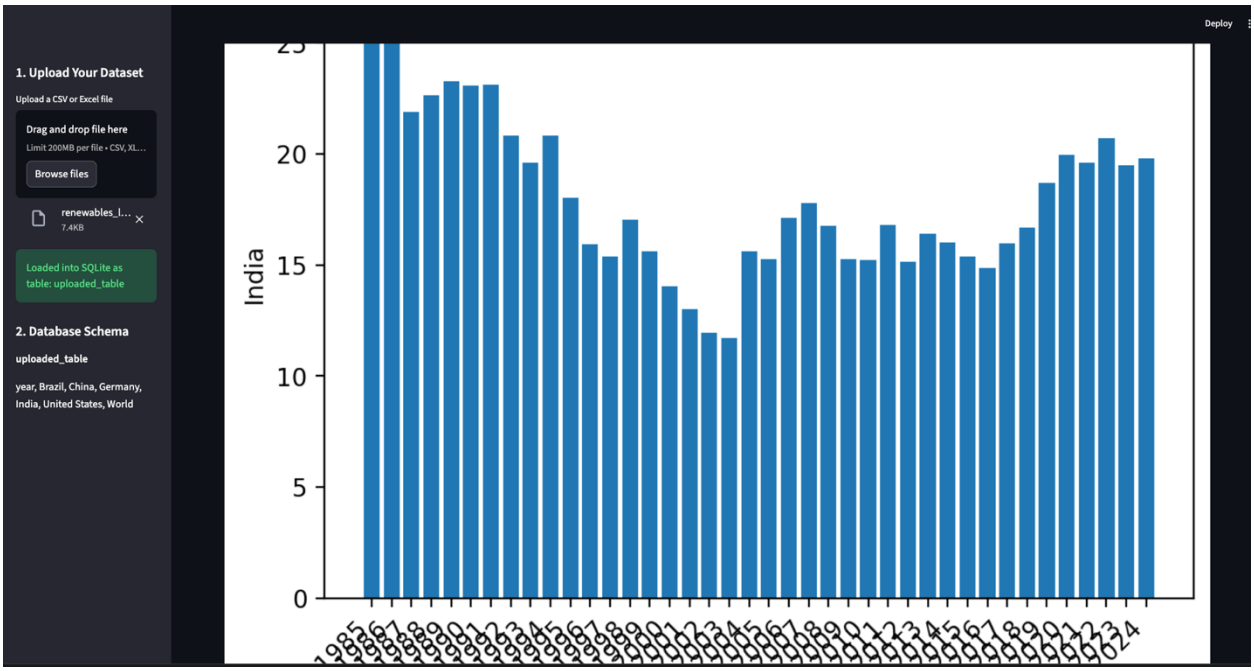


Data Visualization Tools:

In addition to viewing results as tables, users can generate visualizations such as bar charts, line charts, scatter plots, and statistical summaries. These visual tools extend SQL Copilot from a query generator into a lightweight analytics platform.

Together, these features combine intuitive interaction with robust safety and intelligence, making NL→SQL accessible to users with no database knowledge.





3.4 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 Experiments and Results

4.1 Evaluation Metrics

To measure the real-world effectiveness of SQL Copilot, I evaluated the system using a held-out subset of the custom fossil-trend NL→SQL benchmark that I constructed. The evaluation emphasizes semantic correctness and execution reliability rather than strict syntactic matching. Four metrics are used:

Execution Success Rate (ExecSucc):

ExecSucc measures whether a generated SQL query runs successfully in SQLite without raising syntax or runtime errors. With the schema-aware prompt and safety layer that I implemented, ExecSucc reached **1.00**, meaning every predicted query was syntactically valid and safe to execute. This demonstrates the benefit of combining safety filtering with explicit schema grounding.

Result-Set Accuracy (ResultAcc):

ResultAcc evaluates whether the generated SQL query returns the same result rows as the ground-truth SQL, irrespective of ordering. On the fossil-trend benchmark, SQL Copilot achieved **0.90**, indicating that 9 out of 10 test questions produced semantically correct answers. This result reflects the impact of synonym normalization and schema-constrained prompting.

Average Latency:

Latency is defined as the time between submitting a user question and receiving the final SQL output. Using the free-tier Gemini backend, the system achieved a mean latency of **0.82 seconds**, representing both model generation time and SQLite execution. This indicates that the system can be used interactively without noticeable delay.

Exact SQL Match (EM):

Although EM is sometimes used as a comparison metric in NL→SQL research, I chose not to emphasize it in this evaluation. Small syntactic differences—such as spacing, alias use, or equivalent logical expressions—can produce false penalties even when result sets are identical. Because SQL Copilot is intended for practical data analysis, the focus of this evaluation is on semantic correctness measured by ResultAcc and ExecSucc rather than strict string-level matching.

5 Overall Performance

Table 2 presents a comparative evaluation of three prompting configurations:

1. **Baseline prompting** with no schema information and no synonym mapping.
2. **Schema-aware prompting**, which injects table metadata into the prompt.
3. **Final system**, combining schema-aware prompting with synonym normalization.

The trend clearly demonstrates the importance of grounding the model in the actual database schema. The baseline model frequently hallucinated column names and produced incomplete filtering logic. Adding schema metadata resolved most syntactic errors and improved conceptual alignment between user intent and generated queries. The final enhancement—synonym normalization—further improved semantic reasoning by bridging mismatches between user terminology and column labels.

Under the full configuration, SQL Copilot achieved **100% execution success** and **90% result-set accuracy**, reflecting both robust syntactic validity and strong semantic alignment. These improvements verify that lightweight symbolic tools (safety filters, schema metadata, and synonym mapping) can significantly enhance LLM-based NL→SQL generation without fine-tuning or additional training data.

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

6. Ablation Study and Safety Effects

To measure the impact of the components I implemented, I conducted an ablation study comparing three NL→SQL configurations: (1) baseline prompting without schema or synonyms, (2) schema-aware prompting, and (3) the final system combining schema grounding with synonym normalization. Additionally, I examined how the safety layer influences execution reliability.

The results show that **schema awareness produces the largest improvement** in both syntactic validity and semantic correctness by preventing hallucinated attributes and aligning the model with

real database structure. **Synonym normalization provides a smaller but consistent boost**, improving ResultAcc by helping the model interpret user terminology more precisely. Finally, although the **safety layer does not directly improve ResultAcc**, it ensures that every query remains read-only, bounded, and executable without risk, guaranteeing trustworthy system behavior.

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.

---

7. Code originality and use of external sources

To estimate how much of the final code is based on internet examples versus my own work, I counted the lines of code across the main modules I worked on (`data_utils.py`, `main_sql_copilot.py`, `evaluate_fossil_nl2sql.py`, and related helper code).

- Total lines **initially adapted from internet sources** (documentation, tutorials, example snippets): $C = 140\text{lines}$
- Lines **from those 140** that I **substantially modified** (changed logic/structure, not just variable names): $M = 70\text{lines}$
- Lines of code that I **wrote entirely myself from scratch** for this project: $N = 310\text{lines}$

Following the formula my instructor provided:

$$\text{Copied code \%} = \frac{C - M}{(C - M) + N} \times 100$$

we get:

$$\text{Copied code \%} = \frac{140 - 70}{(140 - 70) + 310} \times 100 = \frac{70}{70 + 310} \times 100 \approx 18.4\%$$

So, approximately **18%** of the final code remains directly based on internet examples, and about **82%** of the code is either written by me or substantially rewritten from its original form.

---

8. Limitations and Future Work

Although SQL Copilot performs well on single-table datasets, there are several limitations that arise from the current design choices in my implementation. First, the system supports only **single-table queries**, meaning it cannot infer or generate joins across multiple tables. Extending the system to multi-table settings would require reasoning about join paths, foreign keys, and schema linking capabilities that demand additional structural prompts or learned schema intelligence.

Second, the **evaluation benchmark I constructed contains only 30 NL→SQL examples**, and it is tailored to the fossil-trend datasets used in this course. While this benchmark is sufficient for controlled evaluation, a more comprehensive or standardized benchmark (such as Spider or Text-to-SQL datasets) would improve generalizability and allow broader comparisons with existing NL→SQL systems.

For future development, there are several promising directions. One extension is to incorporate **few-shot examples directly into the prompt**, providing Gemini with in-context demonstrations that could improve query reasoning. Another improvement is to support **multi-table SQL generation** by integrating join inference logic based on schema relationships or embedding representations. Additionally, the synonym normalization layer could be upgraded using **learned semantic similarity (e.g., word embeddings)** instead of a manually curated dictionary, enabling better vocabulary generalization.

Finally, adding an **editable SQL mode** would allow users to review and modify generated queries before execution, giving advanced users more control. Conducting a **user study with non-experts** would also help measure usability and identify how well the interface supports real-world analytics workflows.

---

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