Internship Report
LLM + RAGs for Text to SQL using Agents

Felipe Espinosa [1 2]

Abstract

This report presents a modular agent-based system that enables data analysts to interact with enterprise databases using natural language. Developed during a research internship at Capital Fund Management (CFM), the system translates user questions into executable SQL queries and returns structured results in the form of text, tables, or visualizations. The architecture is based on the Reasoning Without Observation (ReWoo) planning pattern, combining a structured planner, tool execution layer, retrieval-augmented generation, and a final answer synthesizer. The system integrates with internal tools, supports follow-up questions, and is deployed through a Streamlit interface with full LangSmith traceability. Evaluation shows strong performance across a variety of query types, with reduced latency and cost, highlighting its potential for real-world deployment in data-intensive workflows.

1. Introduction

Accessing structured data through natural language is a long-standing goal in human–computer interaction. In enterprise contexts, analysts often rely on SQL to retrieve insights from internal databases, but writing correct and efficient queries requires deep knowledge of schema structure, naming conventions, and domain-specific logic. This creates friction, slows down analysis, and limits access to insights for non-technical users.

Recent advances in large language models (LLMs) have made it possible to translate natural language into SQL with increasing accuracy. However, single-shot prompts that only include raw schema names without retrieval, examples, or tool feedback often fail in real-world settings where databases are large, inconsistently documented, or semantically ambiguous. Moreover, general-purpose solutions typically lack the domain grounding, traceability, and cost-efficiency required for production deployment in enterprise environments.

This work addresses these challenges by developing a conversational SQL agent for internal use at Capital Fund Management (CFM). The system lets analysts ask questions in plain language and returns text, tables, or charts. We rely on a practical stack: *LangChain* provides the building blocks for LLM apps (prompt templates, tool wrappers, typed I/O) to keep components modular (Chase & the LangChain team, 2023); *LangGraph* adds a state-machine on top of LangChain so tool calls follow explicit nodes and transitions, giving deterministic control flow (Team, 2023); *LangSmith* supplies observability and evaluation by logging traces and storing LLM-as-judge outcomes for debugging and iteration (LangChain Inc., 2025); *ChromaDB* is a local vector store where we keep embeddings of schema summaries, column descriptions, recent sample rows, and curated examples for similarity search (Chroma, 2025); and *Streamlit* is the lightweight web UI where analysts ask questions and view answers, and developers can inspect SQL and traces (Streamlit Inc., 2025).

On top of these components, we adopt a plan-first (ReWoo) architecture (Yao et al., 2023): the model produces one explicit tool plan (which tools to call, in what order, with which arguments), and an executor runs that plan deterministically. This reduces back-and-forth LLM calls, lowers latency, and makes outputs more predictable across runs, while improving traceability.

The key contributions of this work are:

- The design and implementation of a working SQL agent that allows analysts at CFM to query financial data in plain language.
- A retrieval-augmented generation pipeline for injecting enriched schema metadata and past examples into the SQLgeneration process.
- The delivery of a usable prototype with a web interface, logging, and feedback features, ready for further testing and integration.

The remainder of this report is organized as follows: Section 2 reviews relevant background and state-of-the-art methods. Section 3 present the motivation and problem statement. Section 4 describes the system architecture and methodol-

ogy. Section 5 presents evaluation results and analysis. Section 6 concludes with lessons learned and future directions.

2. Background and State of the Art

Natural language interfaces to databases (NLIDB) have gained increasing attention in recent years, especially with the rise of large language models (LLMs). These systems aim to bridge the gap between technical database query languages (e.g., SQL) and non-technical users, allowing them to retrieve structured information from databases using plain language. The goal is to reduce reliance on data specialists, speed up access to insights, and democratize data within organizations.

2.1. Text-to-SQL Systems and Benchmarks

Early text-to-SQL research used datasets like *WikiSQL*, which pairs ∼80k natural-language questions with SQL over ∼24k Wikipedia tables, but each query targets a *single table* and mostly simple clauses (e.g., `SELECT`, `WHERE`), limiting compositional difficulty and schema reasoning requirements (Zhong et al., 2017). The *Spider* benchmark addressed these gaps with 10,181 questions and 5,693 unique SQL queries across 200 multi-table databases spanning 138 domains, explicitly enforcing *cross-domain* generalization by separating train/test schemas and stressing harder SQL phenomena (joins, nesting, grouping, ordering, and `HAVING`) (Yu et al., 2018). Together, WikiSQL and Spider form the canonical progression from single-table pattern matching to multi-table, cross-schema reasoning.

However, real enterprise settings are tougher than Spider along several axes: database *scale* (dozens–hundreds of tables, thousands of columns), *messy metadata* (ambiguous names, partial documentation, schema drift), *temporal logic* (latest snapshots, dated compositions), *efficiency constraints* (queries must run fast on large stores), and *interaction* (follow-ups, disambiguation). The *BIRD* benchmark narrows this gap by evaluating 12,751 question–SQL pairs over 95 large, domain-rich databases (∼33.4 GB) and by introducing the *Valid Efficiency Score (VES)*, which rewards answers that are both *correct* and *execution-efficient*(Zhang et al., 2023). These datasets have become standard evaluation frameworks for comparing both neural architectures and more recent LLM-based approaches.

Modern systems typically fall into two categories: fine-tuned models and prompt-based LLMs. Fine-tuned models, such as SQLCoder (Letham & Team, 2023), have been trained on text-to-SQL datasets and demonstrate strong performance, particularly when adapted to a specific domain. Alternatively, prompt-based methods using models like GPT-4 can generalize well without training, but they require carefully constructed prompts and often depend on external schema context. While both approaches have made significant progress, they remain limited by brittleness, schema ambiguity, and a lack of robust context understanding (Wang et al., 2023).

2.2. Retrieval-Augmented Generation (RAG) for Databases

To mitigate the limitations of standalone LLMs, many systems now incorporate Retrieval-Augmented Generation (RAG) (Lewis et al., 2020). RAG pipelines retrieve relevant database schema information, documentation, or example queries at runtime and feed it into the model prompt. This allows LLMs to generate more accurate and context-aware SQL, especially in enterprise environments with evolving or poorly documented schemas.

In the context of database querying, RAG is typically used to inject relevant table definitions, column descriptions, or even user-generated metadata into the prompt. Tools like LangChain and open-source pipelines often use vector databases to store schema-related information and retrieve only the most relevant snippets based on the user's question. However, simple similarity-based retrieval can still return irrelevant matches, overflow the prompt with excessive context, or surface inconsistent metadata, which reduces reliability unless the retrieval process is carefully engineered (OpenAI, 2023).

2.3. Agentic Architectures and Tool Integration

Beyond retrieval, agentic systems offer a promising paradigm for complex or interactive query generation. Inspired by frameworks like ReAct (Yao et al., 2022) and Toolformer (Schick et al., 2023), these systems interleave reasoning steps with tool calls. Rather than generating a SQL query in a single pass, agents operate in loops: planning an action, executing a tool (e.g., a schema retriever or SQL runner), observing the result, and iterating as needed.

Open-source frameworks such as LangChain (Chase & the LangChain team, 2023) and LangGraph (Team, 2023) have generalized this design, allowing for multi-agent workflows, error recovery, and modular decision-making. These architectures are particularly relevant in enterprise contexts where user queries may involve vague terminology, ambiguous logic, or require clarification through dialogue or intermediate steps.

However, agentic approaches introduce new trade-offs. They tend to be more complex to orchestrate, harder to debug, and slower due to multiple LLM invocations. Misuse of tools, error propagation, or unintended loops are also common challenges. Despite these, they offer a path to more resilient and explainable systems when reliability is critical.

2.4. Real-World Tools and Applications

Several open-source frameworks and commercial platforms now offer robust text-to-SQL capabilities grounded in schema retrieval, agentic design, and few-shot LLM prompting.

- **Google's NL2SQL features with Gemini and BigQuery:** Google Cloud has integrated natural language to SQL conversion across products such as BigQuery Studio, Cloud SQL Studio, and AlloyDB AI. These tools rely on Gemini-based LLMs, schema-aware prompting, and internal RAG methods to generate SQL queries and visual results from user questions (Google Cloud, 2024).
- **Agent Development Kit (ADK) for BigQuery:** Google published an open-source Agent Development Kit that simplifies building conversational agents on BigQuery using LLMs like Gemini for text-to-SQL. It supports reusable tools, planner/executor structures, and retrieval pipelines—all applicable to agentic SQL workflows (Looney, 2024).
- **Vanna AI platform and OSS framework:** Vanna provides both an open-source Python RAG framework and a hosted enterprise-grade agent service that transforms natural language into SQL across databases such as BigQuery, Snowflake, Postgres, and Azure SQL. It includes few-shot learning, feedback loops, and data governance features, and is deployed by enterprises to allow analysts to "chat with their data" (Vanna AI, 2024).
- **Open-source frameworks:** LangChain and LangGraph continue to be foundational for building LLM-based agents and retrieval workflows; SQLCoder remains a competitive fine-tuned model designed for SQL generation. DSPy (Khattab et al., 2024) offers a modular programming paradigm emphasizing reliability in LLM applications.

These tools demonstrate how production-ready text-to-SQL systems are evolving from their academic foundations into robust platforms that support ongoing research in data management, particularly when combined with schema-aware retrieval and agent-based planning.

3. Motivation and Problem Statement

Despite recent progress in natural language interfaces to databases, bridging the gap between user intent and executable SQL remains a challenge in real-world settings. At Capital Fund Management (CFM), analysts routinely interact with complex internal databases containing financial and operational data. While these users are technically skilled, writing precise SQL queries often requires deep knowledge of database schemas, naming conventions, and historical usage patterns. This introduces friction, slows down data exploration, and can limit access for non-SQL experts.

Existing solutions based on prompt engineering or fine-tuned models are insufficient in this context. Prompt-only methods often fail when schema complexity increases or when contextual disambiguation is needed. On the other hand, domain-specific fine-tuning leads to rigid models that are costly to maintain and poorly suited to evolving data schemas. Moreover, most current systems do not support multi-step reasoning, error recovery, or traceability, key requirements in financial and enterprise environments.

The central problem addressed in this work is the design of an AI-powered SQL agent that enables natural language access to enterprise data while maintaining reliability, modularity, and explainability. Rather than relying on static prompts, the system dynamically retrieves and integrates schema information, decomposes user questions when needed, and generates candidate SQL queries through structured reasoning. It also supports execution-time feedback and presents final answers in a readable, contextualized format.

This project aims to build a system that is not only technically effective but also usable and maintainable by internal data teams. It must adapt to changing metadata, support diverse user queries, and allow developers to trace and improve its behavior over time. These constraints motivated a modular, agent-based architecture combining Retrieval-Augmented Generation (RAG), controlled query generation, execution, and final answer synthesis, each component designed for clarity, extensibility, and enterprise integration.

4. Materials and Methods

4.1. System Overview

The developed system operates as an agent-based chatbot that translates user queries into executable SQL statements, runs them against a developer-safe version of CFM's internal Oracle database, and returns results in the form of text, tables, or visualizations. The system also supports follow-up questions, clarification, and error correction, enabling multi-turn conversations and richer data exploration (overview in App. B, Fig. 5).

The architecture is modular and composed of specialized tools orchestrated via a LangGraph-based agent. Each tool performs a well-defined function within the pipeline, which includes the following key stages:

Input and Planning: The user submits a natural language question. A reasoning-capable LLM (GPT-o4-mini) gener-

ates a detailed action plan describing which tools should be used, in what order, and with which parameters. The plan is structured using Pydantic objects, ensuring a stable and machine-usable format for downstream steps.

Retrieval Tools:

- **Table Schema Retriever:** This tool works in two steps and is designed to handle common issues in real databases, such as missing or unclear documentation. Sometimes, tables and columns have no comments or only basic ones (e.g., `PRD_NAME TEXT -- name`), making it difficult for both users and language models to understand their meaning. Other times, names are short or unclear, like `PX_LST`, `DAEND`, or generic names like `HISTORY`, which do not clearly explain their purpose.

  Step 1 (table selection). The system compares the user's question to short, LLM-generated descriptions of each table (reviewed by a human). These summaries help find the most relevant tables, even if the original names or comments are not clear.

  Step 2 (detailed context). For each selected table, the tool collects the full schema, any available comments, improved LLM descriptions, and five sample rows. Seeing real data helps to understand what the columns mean and improves context for SQL generation.

  A concrete example of these issues and their resolution is given in App. D, and a diagram of the retrieval module can be found in App. C Figure 6.

- **Good Known Examples Retriever:** This tool retrieves the top-k (with $k = 5$) previously validated examples of natural language questions and their associated SQL queries. These examples are stored in a dedicated vector database and are selected based on semantic similarity. Specifically, the user's question is embedded and compared to all stored examples using LangChain's similarity function, which assigns a similarity score to each example. The five examples with the highest scores are returned as the most relevant. This retrieval improves the quality of generated queries by guiding the LLM with grounded examples and also contributes to building a dataset for future model fine-tuning. While this improves consistency and correctness, care must be taken to avoid bias toward specific query patterns.

- **Data Referential Search:** This tool connects to an internal service previously developed at the company, designed specifically for fast retrieval of financial product information using various identifiers (e.g., name fragments, product codes, market codes). It uses a Redis database (Redis), a type of high-performance vector store, to ensure quick access to product metadata. This service plays a crucial role in grounding the SQL generation process with accurate and up-to-date product information.

Query Tools:

- **Query Generation:** This module transforms natural language questions into executable SQL queries using a multi-strategy approach. It leverages two Chain-of-Thought (CoT) prompting methods inspired by the CHASE-SQL paper (Zhang et al., 2024): one for query decomposition, and one for query planning. Multiple candidate queries are generated in parallel and passed through an evaluation pipeline to select the most accurate result. Section 4.4 provides further details on these CoT methods. An overview of the query generation process is illustrated in Appendix E, Figure 7.

- **Query Verification:** This module acts as a security layer. If the chosen SQL query fails or produces an unexpected result, it is passed to a verification tool, which uses only the Query Plan CoT method to analyze the issue and propose a corrected version. The module supports retrying the updated query and re-evaluating its output.

- **Query Execution:** This tool executes the SQL queries against the Oracle database. It returns results when successful, issues warnings for empty responses, and forwards error messages to the verification module for further analysis.

Table Insight Tool: This tool aggregates all schema metadata, product details, and user requirements before query generation. It generates a concise summary of relevant tables, key columns, join paths, and query strategies, and can also enrich the user question when it is vague or underspecified. Providing this focused summary helps guide the LLM's attention to only the most relevant context, reducing the risk of hallucinated outputs and helping the model stay on track. By performing part of the schema analysis in advance, the tool also streamlines the SQL generation step, which can improve generation speed.

Visualization Tool: When users request visual output, this tool uses Plotly (Plotly) to generate executable Python code based on the SQL results and user instructions. The code is safely executed within the user interface to render the desired chart or graph. Visualizations can be edited interactively or regenerated with new parameters.

Organizer Tool: This final module compiles the final answer using the selected results and generated outputs. It presents the response in a readable, user-friendly format, including tables, text, SQL code, or visualizations. The tool

is explicitly instructed not to alter any data, only to format and structure it as per user specifications.

User Interface: A custom Streamlit-based interface was developed to support both users and developers (see App. F, Fig. 8). End users can input queries, view results, and provide binary feedback (positive or negative). Developers can inspect traces, trigger tool calls manually, and update vector stores used in the retrieval components. User feedback is logged and stored in a dedicated vector database, enhancing future retrieval and enabling human-in-the-loop improvement of the system.

4.2. Data Sources and Preparation

Enterprise Database (Developer Clone). All SQL executions target a read-only developer clone of CFM's internal Oracle database, which mirrors the production schema while avoiding side effects. The system focuses on 30 schemas corresponding to a subset of 30 core tables, primarily related to equities and index composition. Although some tables related to options are accessible, they were not heavily used during this project.

These tables contain a variety of data types, including dates, prices, categorical strings, numerical metrics, and binary indicators (e.g., `Y/N` flags). This diversity supports a wide range of user queries and natural language formulations.

Schema Extraction and Join Graph. We extracted structural metadata such as table names, column types, primary and foreign keys using Oracle dictionary views (`ALL_TABLES`, `ALL_TAB_COLUMNS`, `ALL_CONSTRAINTS`, `ALL_CONS_COLUMNS`). A Python script was developed for this task to ensure reproducibility and scalability as the number of tables evolves.

Metadata Enrichment Policy. We preserved all existing human-authored table and column comments in the database. For any table or column without documentation, we used a pipeline based on large language models (LLMs) to automatically generate missing descriptions.

- **Table summaries**: For each undocumented table, we asked the LLM to write a concise summary (80–150 words) using only the available information: the table name, its columns, their data types, and five sample rows of the table. This provided enough context for the LLM to infer the purpose of the table and describe its main contents, even if the original naming was ambiguous.
- **Column descriptions**: For columns lacking comments, the LLM generated short explanations based on the column name, data type, and sample values. Human-written comments were never overwritten; only missing fields were filled.

Every LLM-generated description was reviewed manually before being added to the schema, to ensure the information was accurate and consistent. This enrichment step is critical for downstream performance: LLMs use these improved comments and summaries as key context when generating SQL queries, so better documentation directly improves both the accuracy and reliability of query generation. In our case, approximately 50% of tables already had native descriptions and the other 50% were enriched by the LLM pipeline. After this process, all tables and columns in the system had complete documentation.

Sample Rows. For each table, we extracted five representative rows to aid disambiguation, especially for categorical or identifier columns. Rows were sampled based on the most recent date. While random sampling is a common approach to maximize variability and avoid bias, in this case many tables contained missing values in randomly selected rows. After manual inspection, sampling the first five rows by descending date proved more reliable, as these entries were typically more complete and informative. These examples were used during table enrichment and embedded alongside schema metadata for retrieval.

Vectorization and Indexing. We constructed three collections using a vector database, which stores and retrieves documents based on the similarity of their vector embeddings rather than keywords. In our system, we use ChromaDB as the underlying vector database to efficiently manage and search through different types of indexed information:

- `description_index`: contains one document per table, with its name high-level LLM-generated summary, primary and foreign keys.
- `schemas_index`: stores full schemas, column descriptions, and sample rows for top-k retrieved tables (see App. G, Fig. 9 for a concrete chunk).
- `examples_index`: contains validated question–SQL pairs used for few-shot learning.

All documents are embedded using OpenAI's `text-embedding-3-large` model, and cosine similarity is used for nearest neighbor retrieval. Retrieval proceeds in two stages: first, the top 10 tables are selected from the `description_index` based on the user's question; then, corresponding documents from the `schemas_index` are attached, including full schema and sample data. At inference time, we also retrieve the top 5 validated examples from the `examples_index`. Average retrieval latency is approximately 0.3 seconds.

Validated Examples (Good-Known Examples). We maintain a growing collection of validated question–SQL pairs. Each item stores a natural language question and the associated SQL query. These examples serve as few-shot prompts to guide the LLM during SQL generation and also as seed data for potential future fine-tuning.

As of this writing, the collection contains 20 examples. During model testing, this index was disabled to avoid contamination and bias, as these examples overlapped with evaluation data. Once the system stabilized, the example collection was integrated into the retrieval process to improve generation reliability.

Product Referential Service. The system integrates with an internal referential service that returns detailed metadata about financial instruments (e.g., full name, IDs, market, code mappings). This tool is used to resolve vague or partial product references in user queries. The service typically responds in under one second and significantly improves SQL generation quality by reducing the need for exploratory subqueries.

Without this tool, the agent would need to generate two-step SQL plans (e.g., find product code, then build the real query), increasing error rates and prompting failures. By providing this data directly to the LLM at generation time, we avoid this indirection and reduce hallucination risk.

Governance, Privacy, and Reproducibility. All database access is performed using read-only credentials, and no production data are written or modified. Vector stores persist only schema descriptions, enriched metadata, and sanitized sample rows. Personally identifiable information (PII) is not stored in any retrievable chunk.

LangSmith is used to log tool traces, model inputs/outputs, and agent decisions for auditability and debugging. All major resources (schemas, embeddings, validated examples) are versioned, and this report refers to snapshot `2025-07-31`.

Appendix: Data Chunk Examples. Examples of the actual documents stored in the vector databases—including table summaries, schema chunks, and question–SQL pairs are provided in Appendix G. These illustrate the structure and formatting used in each collection.

4.3. Tooling and Frameworks

The system was implemented primarily in Python and leverages a set of open-source libraries for orchestration, generation, and retrieval. The following core components were used:

- **LangChain** (Chase & the LangChain team, 2023): for managing prompts, tool wrappers, memory, and integration with external services. LangChain provides high-level abstractions for building LLM applications in a modular fashion.
- **LangGraph** (Team, 2023): a state-machine extension of LangChain used to coordinate complex agent workflows. It enables robust execution of multi-step reasoning with explicit control over transitions, retries, and stopping criteria.
- **LangSmith**: used to log all model interactions, tool traces, and agent decisions. LangSmith was critical for debugging, testing, and performance evaluation throughout the project.
- **ChromaDB**: a local vector database used to store and retrieve embedded documents.
- **OpenAI APIs:** Multiple OpenAI models were used throughout the system, each selected based on its latency, cost, and reasoning capabilities for specific tasks:
  - **GPT-4o-mini**: Used exclusively as the *planning* model within the agentic architecture. It decomposes the user's natural language query into a structured tool execution plan using pydantic-style variable definitions. Its strong reasoning performance and fast response time made it ideal for this orchestration task.
  - **GPT-4.1**: Applied to tasks requiring deeper semantic understanding and complex reasoning, including:
    - *Table Insights Tool*: Generates comprehensive summaries of relevant schema components and query strategies.
    - *Query Generation*: Produces four candidate SQL queries using CoT strategies.
    - *Judging Module*: Scores each (SQL, result) pair to select the best answer.
    - *Verification Module*: Diagnoses and fixes incorrect SQL queries.
    - *LLM Organizer*: Formats and composes the final user-facing response.
  - **GPT-4.1-nano**: Used in the *Product Referential Tool* to filter multiple entity match candidates and select the most relevant based on the user's intent. This lightweight model was chosen for its low latency and adequacy for structured filtering tasks.
  - **text-embedding-3-large**: Employed for vectorization of user questions, table summaries, schemas, and validated examples. These embeddings powered three ChromaDB collections used in the system for similarity-based retrieval.

- **Streamlit**: the user interface was built with Streamlit, allowing users to submit questions, view responses, interact with visualizations, and provide feedback. A developer panel was also included for inspecting tables, examples and updating the vector databases.
- **Plotly**: used to generate Python code for data visualizations on demand. These plots are rendered directly in the UI based on SQL query outputs and user preferences.

All components were integrated using Python 3.11. The environment was managed with virtual environments and versioned through Git. No cloud infrastructure was required, and the system was deployed on local developer machines with access to the internal Oracle database.

4.4. Query Generation and Execution

The central objective of the system is to accurately translate natural language questions into executable SQL queries. To ensure robustness, the system generates multiple candidate queries in parallel and selects the best result through evaluation and scoring mechanisms.

Generation Strategy. Following the CHASE-SQL methodology (Zhang et al., 2024), our system uses two distinct Chain-of-Thought (CoT) prompting strategies to guide query construction: *Divide-and-Conquer CoT* and *Query Plan CoT*. Each is applied in two independent runs resulting in a total of four candidate queries per question.

- **Divide-and-Conquer CoT:** This strategy decomposes a complex natural language question into smaller subproblems. These are articulated as pseudo-SQL instructions, which help the model focus on individual components such as filters, joins, or aggregations. For example, in a multi-step query requiring a join across several tables and computation of market capitalization (number of shares multiplied by share price), the model first identifies relevant tables, extracts fields step-by-step, and constructs intermediate SQL fragments. These fragments are then aggregated and post-processed to form the final SQL query. An optimization step is applied to simplify redundant clauses and remove unnecessary joins. This method is particularly effective for complex, nested logic.
- **Query Plan CoT:** Inspired by traditional database execution plans, this approach leads the model through a structured SQL generation path. First, the model generates a human-readable query plan: which tables to query, which filters to apply, and how to join the data. Each decision point (e.g., selecting latest prices, identifying index members) is explicitly reasoned. The final SQL is constructed by mapping these steps into clauses in the FROM, WHERE, and SELECT blocks. This strategy performs well when precise reasoning over schema relationships and temporal constraints is required.

All four SQL queries are generated using GPT-4.1. Prompts follow structured templates and may include validated few-shot examples when retrieval is enabled. Otherwise, they rely solely on schema and metadata retrieved via semantic search.

Execution and Judging. The four candidate queries are executed in parallel on the developer database. Queries that fail (due to syntax errors, type mismatches, or invalid joins) or return empty results are excluded.

Remaining candidates are passed to a judging module, also implemented using GPT-4.1. Each `(SQL, result)` pair is evaluated along three dimensions:

- Relevance to the original question
- Correctness and consistency of the returned data
- Completeness and clarity of the answer

The top-ranked answer is returned to the user.

Fallback and Verification. If all generated queries fail or none meet quality thresholds, the input is passed to a verification module. This module uses a CoT-based prompt, specifically, the Query Plan strategy to reason through potential causes of failure and regenerate a corrected SQL. This choice was motivated by its ability to explicitly analyze failure points step-by-step, making it effective for identifying missing joins, improper filters, or logical mismatches. The regenerated query is re-executed and re-evaluated using the same judging procedure.

Design Decisions. Rather than relying on traditional tool-based agent loops (e.g., LangChain agents), which require multiple reasoning and tool-execution steps, we opted for stateless CoT prompting to induce reasoning directly within a single LLM call. This was driven by empirical observations: without guidance, LLMs often produce SQL that appears syntactically correct but fails upon execution or yields irrelevant results. CoT prompts, by contrast, yield more interpretable reasoning traces and help prevent logical errors early on. This approach strikes a practical balance between latency, cost, and correctness, without the need for fully agentic control flows.

4.5. Agent Workflow and Tool Integration

The system's reasoning and execution follow a structured architecture inspired by the "Reasoning Without Observation" (ReWoo) pattern, as proposed in (Yao et al., 2023) and adopted in the LangGraph framework (Team, 2023). This architecture was selected for its efficiency, predictability, and suitability for chat-based systems, where latency, cost, and clarity are critical.

Architecture Overview. The agent operates through three core modules:

- **Planner:** An LLM (GPT-o4-mini) generates a detailed plan to answer the user's question. The plan consists of a sequence of tool calls, each labeled with variables (e.g., #E1, #E5) and accompanied by instructions or arguments. The output format is a structured object parsed using Pydantic to ensure type safety and compatibility with downstream execution. An example agent plan is provided in Appendix H.
- **Workers:** Once the plan is created, the system invokes each tool sequentially in the order specified. Execution is strictly plan-driven, there is no intermediate LLM reasoning. Tools are selected dynamically depending on the user query and may include schema retrieval, referential search, query generation, or others. The system does not rely on static tool nodes; the actual pipeline varies across user queries.
- **Solver:** Instead of passing all tool outputs to the final answer generator, only selected responses (e.g., from query execution or referential search) are sent to a dedicated "organizer" LLM. This model composes the final user-facing answer in a concise and structured format (text, table, or visualization). Filtering unnecessary tool outputs reduces token usage and prevents noisy generations.

Why ReWoo Architecture? We chose this planning-centric structure over more flexible alternatives like ReAct or Plan-and-Execute due to strict latency and cost constraints. ReWoo minimizes LLM calls: reasoning is consolidated into a single planner invocation, followed by deterministic tool execution. This ensures faster response times, lower token consumption, and easier traceability.

In practice, most generation failures stem from tool misuse or incorrect variable parsing, not from the planning step itself. As such, concentrating reasoning upfront has shown to improve consistency and makes failure diagnosis more straightforward.

Execution Flow. Each conversation turn begins with a user question. The planner outputs a tool sequence using labeled variables. The tool executor parses this plan and invokes each tool in order. Tool outputs are stored in a python dictionary that keep track of all the messages (inputs, outputs, results) of the agent. Once execution is complete, the relevant outputs are passed to the LLM-based organizer tool, which formats the final response.

This agent design results in three stable LangGraph nodes: `plan`, `tool`, and `solve`. The planner is only invoked once per turn. LangSmith is used to log the execution trace, including planner decisions, tool inputs/outputs, and final answers.

Modularity and Adaptability. New tools can be added to the system without modifying the core agent logic. Since the planner dynamically selects tools based on the user question, the agent can flexibly adapt to different task types. This design supports future extensions, such as integrating feedback loops, dynamic memory, or summarization modules.

4.6. Answer Construction and Visualization

Once all relevant tools have been executed, the final user-facing response is constructed through a dedicated LLM-based module referred to as the **organizer**. Unlike traditional agent architectures where the LLM generates the answer based on all prior outputs, this system selects only the necessary tool results (e.g., SQL query result, referential data, visualization instructions) to reduce token usage and prevent information overload.

Answer Assembly. The organizer tool receives a curated subset of tool outputs, determined by the system's internal logic. For example, if the user question required a SQL query and a referential lookup, only the corresponding results are passed to the organizer. This filtering step is critical to maintain generation efficiency and accuracy, especially for long or multi-step queries.

The organizer is prompted to produce a coherent, readable, and well-structured answer that matches the format expected by the user. Depending on the query, the final response may include:

- **Plain text summaries**, when the result is short or explanatory
- **Tables**, when structured data is returned (e.g., top 10 companies, composition of an index)
- **SQL code**, when the user explicitly requests it or when transparency is beneficial
- **Visualizations**, when trends or comparisons are involved

Visualization Pipeline. If the user requests a chart (or if the planner infers that a visual representation would be useful), the visualization tool is invoked. It receives the SQL query result, user instructions, and context, and generates Python code using the `plotly` library. The generated code is validated and executed within the UI to produce the final chart. Users may modify the chart parameters or regenerate it based on new inputs.

User Interface. The interface, built with Streamlit, presents the full interaction pipeline. Users can:

- Enter free-text questions
- View results (text, tables, plots)
- See the underlying SQL query if desired
- Provide binary feedback (positive/negative) on the answer

Developers also have access to a control panel within the UI that allows for updating and maintain vector databases (tables schemas, tables descriptions, good known examples).

5. Evaluation Setup

The evaluation aimed to measure the performance of the proposed text-to-SQL agent under different prompting strategies and retrieval configurations. The primary goal was to assess accuracy, latency, token consumption, and execution cost across representative query scenarios.

Dataset. A curated evaluation set of 48 natural language questions was used. These questions were specifically designed to cover a diverse range of SQL query patterns and data access scenarios present in the developer database. Each question was paired with a validated *gold SQL query* and a *reference answer table*, both generated on August 1, 2025. The underlying database snapshot includes data up to July 31, 2025.

This explicit date cutoff is essential for reproducibility: because the database is updated daily, answers to the same question could change over time as new data appears. By restricting evaluation to data available up to July 31, 2025, we ensure that all evaluation results are stable and consistent, and can be fairly compared or reproduced in the future. When running new agent queries after this date, any value that matches the expected structure and the reference answer (from data before August 1) is considered correct, even if there are newer records in the database.

Evaluation Methodology. Correctness was measured using an *LLM-as-a-judge* approach, implemented with GPT-4.1. The judging prompt was explicitly instructed to:

- Ignore differences in data post–August 1, 2025.
- Accept extra columns or minor row differences, provided the core result matches the reference.
- Focus on semantic equivalence rather than byte-by-byte matching.

For each test run, the agent produced a final SQL query (or direct retrieval answer, for non-SQL cases), executed it, and returned the result table. The judge received:

1. The original user question.
2. The *gold* answer.
3. The agent's predicted answer.

It responded with `CORRECT` or `INCORRECT`.

Metrics Collected. For each configuration, the following metrics were measured:

- **Accuracy:** Percentage of questions marked CORRECT by the judge.
- **Latency:** Average end-to-end execution time per question, including planning, retrieval, query generation, execution, and judging.
- **Tokens:** Average number of tokens consumed across all model calls.
- **Cost:** Approximate USD cost per question, based on OpenAI API pricing for GPT-4.1, GPT-o4-mini, and GPT-4.1-nano.

5.1. Test Configurations

Four experimental configurations were evaluated. They were chosen to isolate the contribution of each prompting strategy and to measure the impact of example retrieval.

1. **Full System – No Examples:** Combines two candidates from the Divide-and-Conquer (DC) CoT strategy and two from the Query Plan (QP) CoT strategy. *Goal:* Serve as a baseline for the full multi-strategy system without retrieval bias from few-shot examples.
2. **Full System – With Examples:** Same as above, but retrieves relevant *few-shot examples* from the validated examples DB during schema retrieval. *Goal:* Measure the added value of example-based grounding in improving SQL correctness.

3. **Divide-and-Conquer Only:** Generates four SQL candidates exclusively with the DC strategy, no examples. *Goal:* Evaluate the standalone performance of DC prompts, which excel in decomposing complex, multi-step queries.
4. **Query Plan Only:** Generates four SQL candidates exclusively with the QP strategy, no examples. *Goal:* Assess the effectiveness of QP prompts for schema reasoning and join-path discovery without influence from DC.

By comparing these configurations, we can (i) quantify the benefit of combining prompting strategies versus using them in isolation, and (ii) determine the impact of including retrieved examples on overall performance.

5.2. Quantitative Results

We evaluated the four configurations from Section 5.1 on the 48-question benchmark, computing mean values and confidence intervals for accuracy, latency, token usage, and monetary cost. Confidence intervals are calculated using bootstrap resampling to estimate variability across the evaluation set.

Accuracy

Figure 1 shows the accuracy distribution per configuration. The Full System + Examples configuration achieved the highest mean accuracy at 87.50%, representing a +4.17 percentage point improvement over the Full System – No Examples (83.33%).

This gain is directly attributable to the inclusion of schema-grounded, curated few-shot examples in the generation stage. These examples act as contextual priors, reducing the hypothesis space the LLM must explore when generating SQL. By anchoring the reasoning process with known table access patterns and join strategies, the planner reduces the risk of both structural errors (invalid SQL) and semantic errors (incorrect joins, filters, or aggregations).

Among isolated strategies, Divide-and-Conquer outperforms Query Plan (81.25% vs. 77.08%), indicating that explicit decomposition of sub-tasks can improve correctness in queries requiring multiple join conditions or conditional aggregations. However, without the complementary global structure that the other approach provides, decomposition alone cannot reach the accuracy of the combined system.

Latency

Figure 2 presents total end-to-end agent latency. Full System + Examples is also the fastest configuration, with a mean of 51.20 s. This reduction relative to the No Examples variant (54.37 s) can be explained by the use of targeted few-shot examples in the prompt: these examples guide the LLM more directly toward the correct SQL patterns, narrowing the range of possible outputs and helping the model generate answers more efficiently.

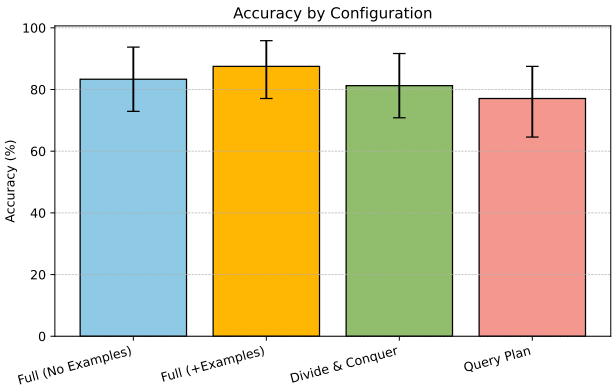

Figure 1. Accuracy by configuration with confidence intervals.

In contrast, Divide-and-Conquer incurs the highest latency (62.60 s). This is mainly due to the design of its prompts, which encourage the LLM to generate more verbose, step-by-step reasoning within a single call. As a result, the model outputs more tokens for each candidate query, increasing total processing time.

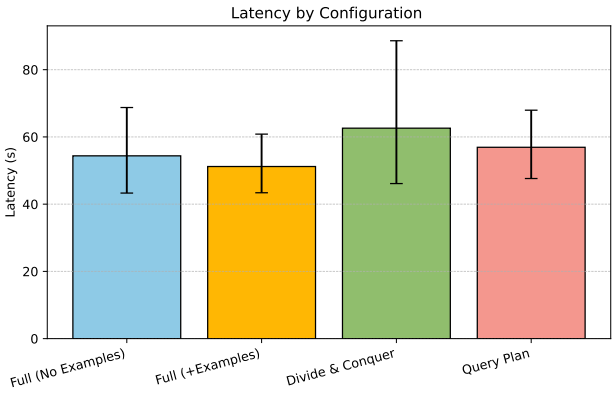

Figure 2. Latency by configuration with confidence intervals.

Token Consumption

Token usage (Figure 3) closely mirrors latency trends. Divide-and-Conquer has the highest mean token count (162.41 k) because its prompt format encourages the LLM to generate detailed, step-by-step reasoning in the output, resulting in longer responses. Query Plan uses the fewest tokens (146.83 k), as it produces a more concise, structured plan in a single output, with less intermediate explanation. Differences in token consumption across configurations are primarily a result of the prompt structure and the verbosity

of the model's generated answers, rather than any difference in the number of model calls.

The Full System variants fall between these extremes, with the +Examples configuration slightly higher (152.60 k) due to the extra context tokens from example insertion. However, this token overhead is proportionally small compared to the accuracy gains and latency reduction it enables.

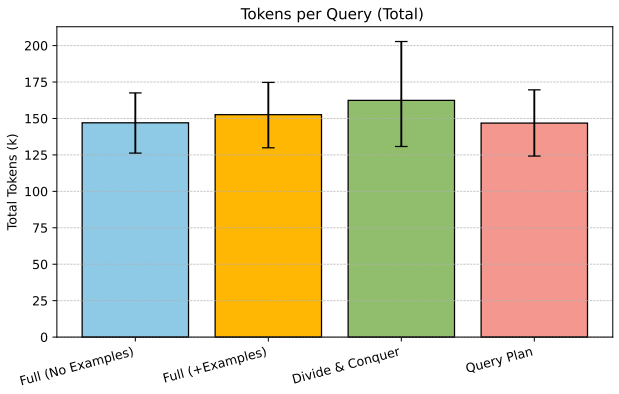

Figure 3. Total tokens per query with confidence intervals.

Cost per Query

Figure 4 reports the monetary cost per query, which is a linear function of total tokens processed and the model's pricing schedule. Query Plan is the cheapest ($0.2230) due to its compact reasoning trace. Divide-and-Conquer is the most expensive ($0.2547), reflecting its high token footprint.

Full System + Examples incurs a modest cost increase over the no-examples baseline (+$0.0162/query) but delivers significant accuracy and latency benefits, which in production may outweigh the marginal expense.

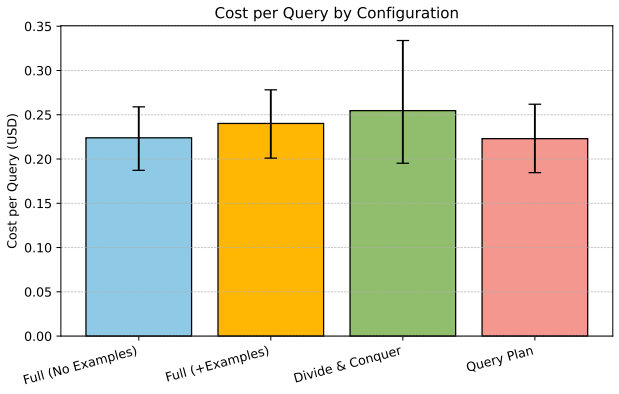

Figure 4. Cost per query by configuration with confidence intervals.

Technical Interpretation

From a systems perspective, these results reveal several operational trade-offs:

1. Grounded few-shot prompting improves both accuracy and latency, even when it increases input length. This indicates that reducing search complexity in the LLM's reasoning outweighs the marginal cost of processing more tokens.

2. Divide-and-Conquer and Query Plan each have strengths: Divide-and-Conquer can handle more complex queries with detailed logic, while Query Plan tends to be more efficient. However, the overall differences between the two are moderate, and neither method is clearly superior across all cases.

3. The combined Full System works better because it takes advantage of both strategies: the Query Plan approach helps the model choose the right tables and joins for the overall structure, while the Divide-and-Conquer approach focuses on getting the details right for complex or multi-step requirements. Using both together reduces the chance of missing key logic or making mistakes in the SQL. Adding few-shot examples further helps by giving the model concrete patterns to follow.

In summary, the Full System + Examples configuration offers the most robust and balanced performance, making it a practical default choice for production environments.

Full Metrics Table in Appendix. The detailed numerical results, including all metric means and confidence intervals, are provided in Table 1 in Appendix A, where they can be displayed at full size for clarity.

5.3. Robustness and Error Taxonomy

This section summarizes where errors come from and how the system handles them. The analysis is based on execution traces, agent logs, and the aggregated results in Section 5.2 (Figures 1, 2, 3, 4).

Entity and Identifier Resolution User questions often use partial names or legacy codes. Without grounding, this creates ambiguous joins or empty results. The Product Referential Service resolves names and codes before SQL generation, which reduces two-step exploratory plans. This design choice is consistent with the higher accuracy and lower latency of the Full System + Examples configuration: better grounding reduces the search space and the number of retries.

Join Path Selection Picking the wrong join keys or an unnecessary bridge table is a common failure mode. The two-stage schema retrieval (high-level table descriptions, then full schema + sample rows) plus the Table Insight Tool narrows the candidate join graph and surfaces primary/foreign keys early. Divide-and-Conquer helps on complex multi-table queries by making constraints explicit, which matches its relatively strong accuracy when used alone. The trade-off is seen in Figure 2 and Figure 3: more intermediate reasoning means more time and tokens.

Temporal Predicates Questions about "latest value", "composition on a date", or windowed aggregates fail when the plan under-specifies time filters. The Query Plan prompt makes these steps explicit (e.g., MAX(date) joins, BETWEEN windows). Few-shot examples further bias the planner toward stable date patterns. This helps explain why adding examples improves both accuracy and latency despite slightly higher prompt length.

Syntactic vs. Semantic Success Valid SQL can still be wrong (empty sets, mismatched filters) or misleading (partial joins). The verification module inspects errors and reprompts with a structured plan, which corrects a share of failures without human input. This contributes to the Full System's balance between accuracy and speed.

Planner Mis-specification and Non-determinism With a ReWoo-style planner, a poor initial plan can lead workers down a suboptimal path deterministically. Verification mitigates this but cannot fix every case. Stable prompts, consistent planning variables (pydantic), and curated examples reduce plan variance.

6. Discussion

This section summarizes what worked well, what did not, key trade-offs, limitations, and why the system is useful for Capital Fund Management (CFM).

6.1. What worked well

Few-shot grounding with examples Adding a small set of curated, schema-aware examples improved both accuracy and latency in the Full System. Examples act like contextual priors and reduce the search space for the planner, which leads to more direct SQL generation and fewer retries.

Two-stage retrieval and Table Insight Tool Splitting retrieval into high-level table summaries and then full schemas with sample rows helped the model pick correct join paths and key columns. The Table Insight Tool aggregates this context and gives the generator a clear picture of the tables to use and how to join them.

Planner–Worker–Solver (ReWoo) structure A single planning call followed by deterministic tool execution reduced unnecessary LLM calls. This made traces easier to inspect and control in LangSmith, and simplified debugging compared to agent loops with frequent back-and-forth reasoning.

Verification and judging The verification step recovered a share of failing runs by analyzing error messages and regenerating queries with explicit plan steps. The LLM judge supported semantic evaluation when output values changed after the snapshot date.

6.2. What did not work as well

Agentic RAGs I tested agentic RAG setups that route between multiple retrievers and tools at run time. They are powerful when documents are long and self-contained, because tool selection can bring in the exact chunk that answers the question. In our case the "documents" are short: table schemas, column lists, and a few sample rows. The extra routing added latency and tokens without adding much signal. A simpler two-step RAG works better. This keeps prompts small and makes the planner's job easier.

Sending everything A tempting idea is to send all schemas for all tables and let the model decide. This seemed fine for very simple questions, but it failed on harder ones: the model produced spurious joins, missed temporal predicates, and sometimes returned empty results. It also inflated tokens and latency. The system is more robust when we aggressively filter: rank tables by the question, keep only the top candidates, and pass a compact, ordered context to each step. The quantitative results reflect this: methods that reduce search space are both more accurate and faster.

Reasoning-optimized models Reasoning models can solve tricky cases, but their execution time is unstable. In my tests, the same prompt could run in seconds or stretch to several minutes, with no guarantee of a better answer. For example, query generation with a reasoning-heavy setup occasionally took 4–5 minutes in exploratory runs outside the benchmark. This is not acceptable for interactive workflows.

6.3. Trade-offs

Accuracy vs. speed and cost More reasoning and more context usually raise token usage and latency. Few-shot examples are an exception: they add tokens but often reduce total reasoning depth. The best trade-off for production is the Full System with examples.

Simplicity vs. robustness The ReWoo planner simplifies control flow and reduces LLM calls, but a poor initial plan can still send tools down the wrong path deterministically.

Generalization vs. specialization Examples and enriched schemas make the system strong on the covered domain. Porting to a new schema or vendor requires re-extraction of constraints, re-enrichment, and fresh examples.

6.4. Limitations

Evaluation scope The test set covers 48 questions on a focused subset of enterprise tables. Wider coverage may need new examples and prompt tuning.

Judge reliability To check the reliability of the LLM judge, its decisions were manually reviewed for all evaluation examples in this study. This was feasible because the dataset was small. For larger datasets, it becomes impractical to manually verify every answer, so there is a risk that the LLM judge may mis-score some cases. Overall, while LLM judges are useful for automatic evaluation, some errors can go unnoticed, especially when results are not double-checked by humans.

Prompt and retrieval sensitivity Performance depends on prompt templates, example quality, and how up-to-date the embeddings are. Poorly curated examples or outdated embeddings can lower accuracy and increase cost. To ensure results are reliable and reproducible, all prompts, example sets, and vector database snapshots should be saved and versioned over time.

Latency and cost variance The pipeline includes multiple model calls and external tools. Mean performance is strong, but tail latency can still occur.

System coupling The implementation is tuned to Oracle SQL, current date idioms, and internal naming conventions. Porting to other databases requires new constraint extraction and updates to canonical examples.

Planner trade-offs A wrong initial plan can propagate through deterministic tools. Keeping plans simple, grounding with examples, and monitoring verification loops are necessary to reduce this risk.

7. Conclusion and Future Work

7.1. Conclusion

This project delivered a robust SQL agent capable of translating natural language questions into accurate, executable SQL queries. The final system integrates a planning-first architecture, a two-stage schema retrieval pipeline, curated few-shot examples, and a verification loop with semantic judging. Each component was selected after evaluating alternative approaches, discarding those that added complexity or latency without improving results, such as agentic RAGs, unfiltered schema dumps, and reasoning-heavy models with unpredictable execution times.

Quantitative experiments showed that the Full System with curated examples achieved the best balance of accuracy, latency, and cost on a 48-question benchmark. Targeted retrieval using high-level table descriptions, combined with schema-grounded examples, allowed the LLM to converge to correct queries faster and with fewer retries. The planner–worker–solver structure reduced unnecessary LLM calls, improved traceability, and simplified debugging, while verification and judging increased robustness against certain error classes.

For the company, this system lowers the barrier for non-expert users to access data, reduces onboarding time by embedding schema knowledge into the retrieval process, and ensures safety through read-only execution and trace logging. Its modular design, strong observability, and controlled trade-offs between speed, accuracy, and cost make it ready for internal deployment, with a clear path for continuous improvement through example curation, retrieval updates, and enhanced verification.

7.2. Future Work

Future improvements will focus on three main areas. First, enhancing retrieval quality through hybrid search (dense and lexical), better handling of schema changes, and improved ranking of relevant tables and columns. Second, refining example management by rotating and curating few-shot examples to maintain diversity, avoid overfitting, and ensure they remain aligned with the current schema. Third, strengthening verification and safety with faster error recovery, clearer fallback messages, and lightweight checks on generated SQL before execution.

Additional opportunities include fully deploying the SQL agent and making it available to real users, integrating it with company tools. Strengthening multilingual support, especially for English and French would improve accessibility. Evaluating alternative models to reduce cost and latency while maintaining accuracy remains a priority, further enhancing robustness and usability for scaling across teams and workloads.

References

Chase, H. and the LangChain team. Langchain: Building applications with llms through composability, 2023. https://github.com/langchain-ai/langchain.

Chroma. Chromadb documentation: Introduction, 2025. URL https://docs.trychroma.com/docs/overview/introduction.

Google Cloud. Introducing nl to sql in bigquery and gemini. https://cloud.google.com/blog/products/data-analytics/nl2sql-with-bigquery-and-gemini, 2024. Accessed: 2025-08-07.

Khattab, O., Ré, C., Zhang, T., and Hashimoto, T. B. Dspy: Programming large language models. *arXiv preprint arXiv:2402.19114*, 2024.

LangChain Inc. Langsmith: Observability for llm applications, 2025. URL https://docs.langchain.com/langsmith/home.

Letham, B. and Team, D. Sqlcoder: Text-to-sql in context. *arXiv preprint arXiv:2306.04759*, 2023.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Kulkarni, M. J., et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 2020.

Looney, J. Build conversational sql agents with the agent development kit for bigquery. https://medium.com/google-cloud/agent-development-kit-nl2sql-with-bigquery-bd0f0cd0c06b, 2024. Accessed: 2025-08-07.

OpenAI. Best practices for retrieval-augmented generation with openai models. 2023. Available online: https://platform.openai.com/docs/guides/retrieval.

Plotly. Plotly. https://plotly.com/. Online; accessed 31 August 2025.

Redis. Redis documentation. https://redis.io/docs/latest/. Online; accessed 31 August 2025.

Schick, T., Dwivedi-Yu, D., Schütze, H., et al. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

Streamlit Inc. Streamlit documentation, 2025. URL https://docs.streamlit.io/.

Team, L. Langgraph: State machines for llm agents, 2023. https://github.com/langchain-ai/langgraph.

Vanna AI. Vanna ai – rag-based open source llm framework for sql generation. https://vanna.ai, 2024. Accessed: 2025-08-07.

Wang, J., Zhang, R., Li, L., Lin, X., et al. A survey of text-to-sql models and datasets. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2023. To appear.

Yao, S., Zhao, J., Yu, D., Yu, T., Park, K., Guu, K., and Gao, J. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Yao, S., Zhang, D., Zhao, Z., Zhan, R., Jiang, Z., and Yu, Z. Reasoning without observation improves tool-usage for large language models. *arXiv preprint arXiv:2305.18323*, 2023.

Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, Q., Yao, W., Li, S., Tan, Y., et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

Zhang, L., Wei, F., Tan, H., et al. Bird: Benchmarks for intelligent retrieval and database qa. *arXiv preprint arXiv:2306.05006*, 2023.

Zhang, T., Wu, L., Yang, Y., Li, Y., Deng, W., Tang, X., Li, H., Zhang, J., and Liu, X. Chase-sql: Chain-of-thought prompting elicits synergistic reasoning for text-to-sql. *arXiv preprint arXiv:2410.01943*, 2024.

Zhong, V., Xiong, C., and Socher, R. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.

A. Full Quantitative Metrics

Configuration	Accuracy (%)	Latency (s)	Tokens (k)	Cost (USD)
No_Examples	83.33 [72.92–93.75]	54.37 [43.30–68.75]	147.02 [126.24–167.53]	0.2240 [0.1872–0.2590]
With_Examples	**87.50** [77.08–95.83]	**51.20** [43.39–60.83]	152.60 [129.88–174.73]	0.2402 [0.2010–0.2782]
D_And_C	81.25 [70.83–91.67]	62.60 [46.12–88.61]	**162.41** [130.76–202.80]	**0.2547** [0.1952–0.3339]
Query_Plan	77.08 [64.58–87.50]	56.92 [47.63–67.95]	146.83 [124.20–169.64]	0.2230 [0.1846–0.2619]

Table 1. Mean performance metrics with 95% confidence intervals in brackets, computed over $n = 48$ questions. Tokens are reported in thousands (k).

B. System Workflow Diagram

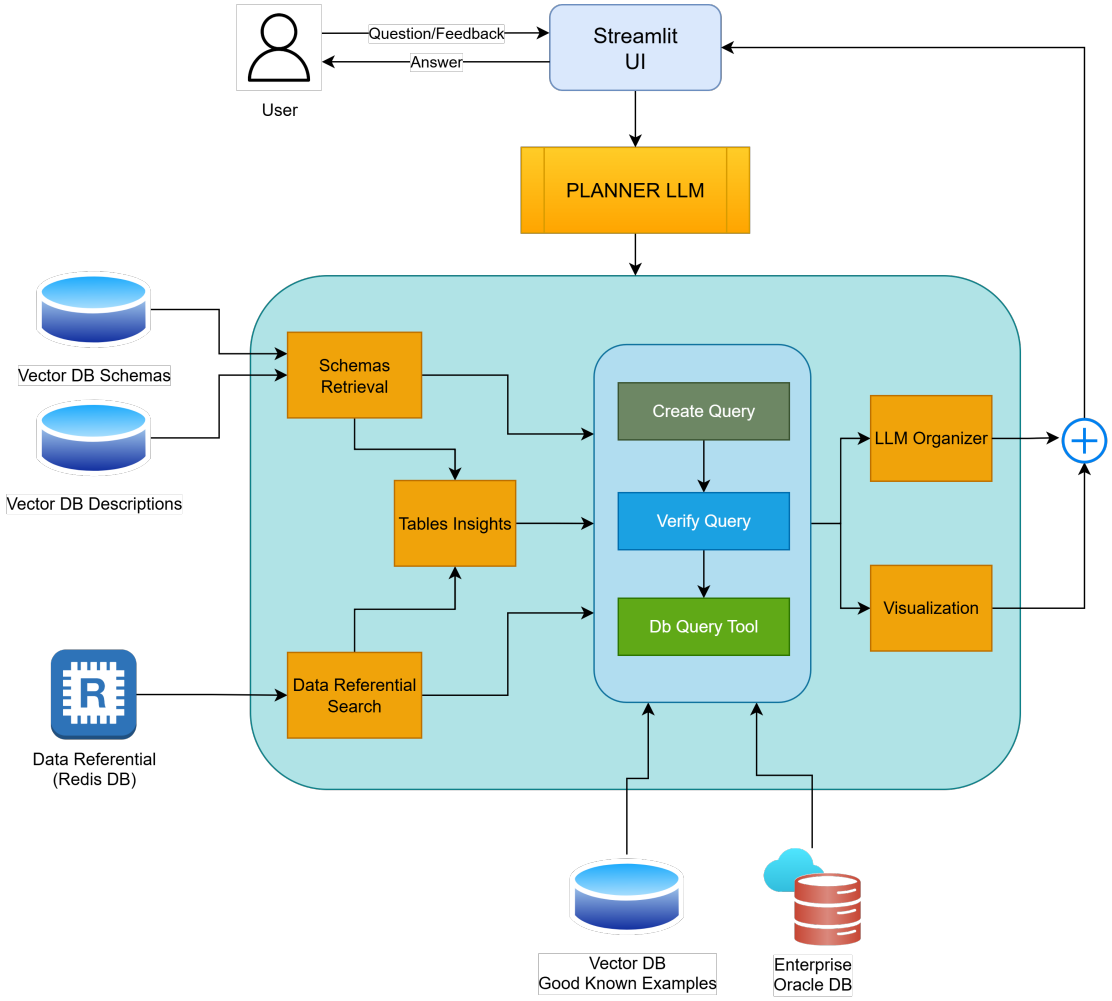

Figure 5. High-level workflow of the SQL Agent system, showing the planner, worker components, and data sources.

C. Retrieval Module

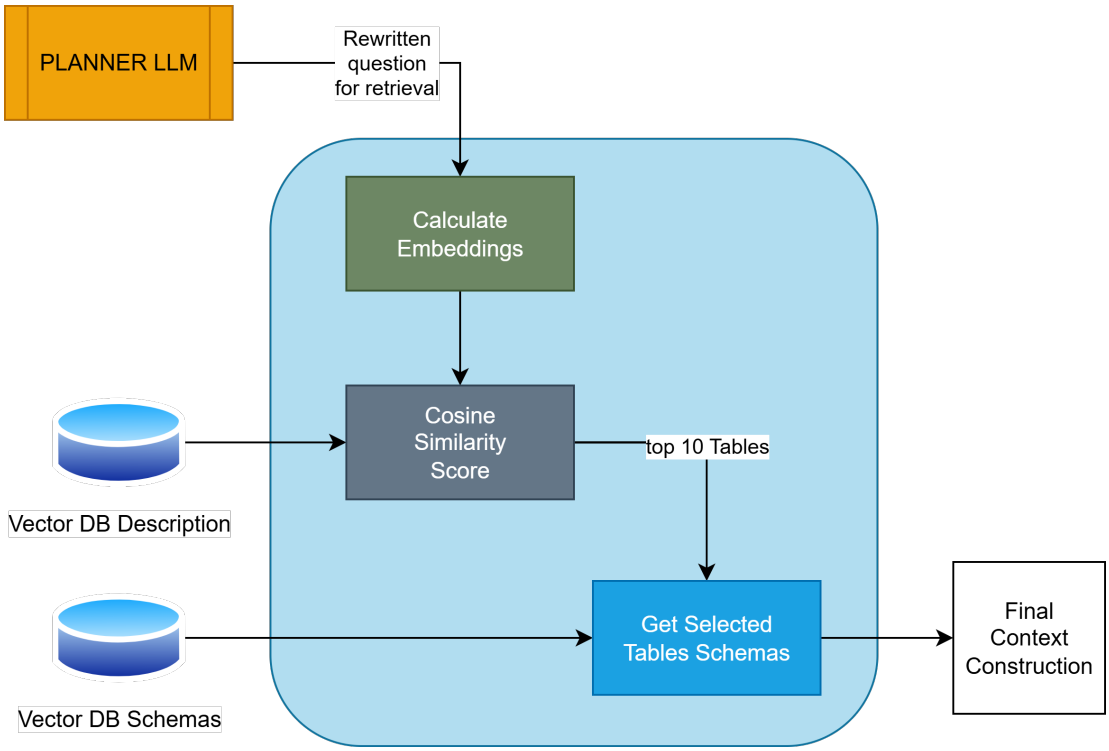

Figure 6. Retrieval pipeline: The planner rewrites the user question for retrieval, embeds it, and selects relevant tables using vector databases for descriptions and schemas.

D. Example: Poorly vs. Well-Documented Table

A. Poorly Documented Table Example

```
CREATE TABLE PRODUCTS (
   PRD_ID INTEGER,
   PRD_NAME TEXT,
   PX_LST FLOAT,
   DAEND DATE,
   HISTORY TEXT
);

-- PRD_NAME TEXT -- name
-- PX_LST FLOAT --
-- DAEND DATE --
-- HISTORY TEXT --
```

This table provides almost no explanation for what the columns mean. For example, `PX_LST` is just a short code, `DAEND` is not explained, and comments are missing or uninformative.

B. Well-Documented Table Example (after LLM-based enrichment)

```
CREATE TABLE PRODUCTS (
   PRD_ID INTEGER, -- Unique identifier for each product
   PRD_NAME TEXT, -- Full product name as used in official listings
   PX_LST FLOAT, -- Last traded price for the product
   DAEND DATE, -- End date of the product's trading period
   HISTORY TEXT -- Description of historical changes or notes
);
```

```
-- The PRODUCTS table contains all available products with identifiers, descriptive names,
    current prices, and trading period end dates. The HISTORY column tracks notable
   updates or changes for each entry.
```

Here, each column is clearly described, ambiguous names are clarified (e.g., `PX_LST` as "Last traded price"), and a table-level summary provides context for users and LLMs.

E. Query Generation Module

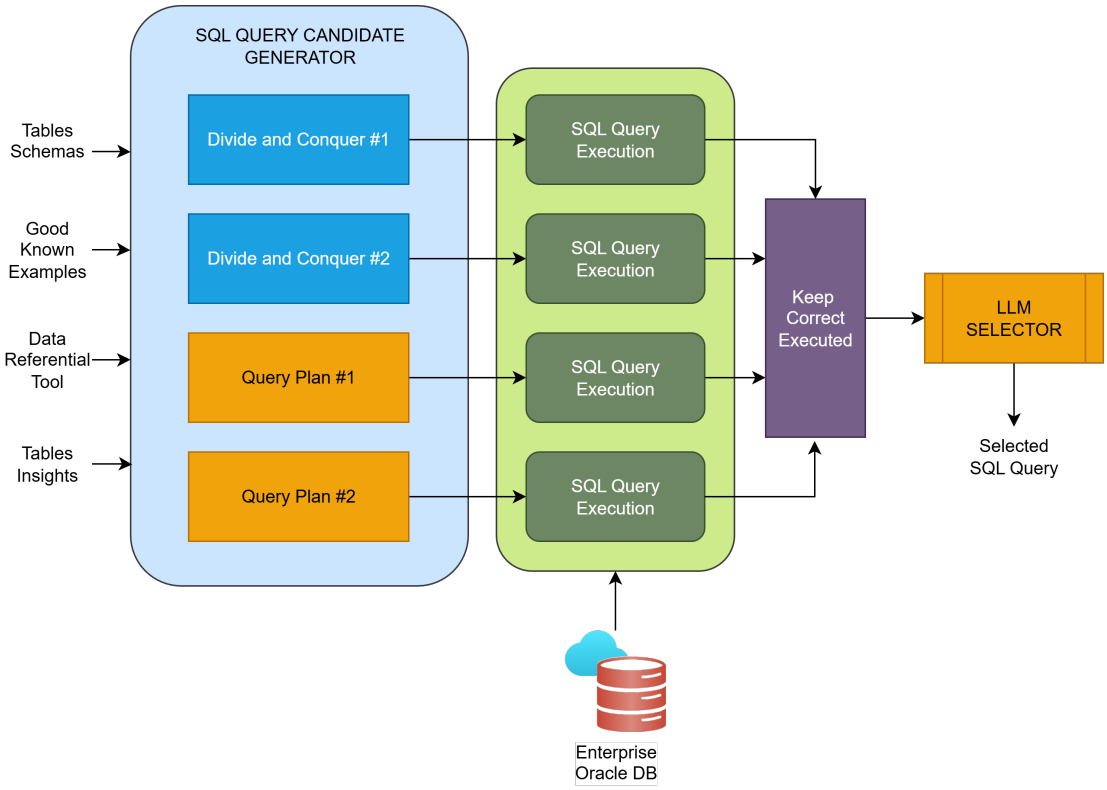

Figure 7. Query generation module: Multiple candidate SQL queries are produced by two prompt strategies, executed, and filtered before selection by the LLM-based judge.

F. Streamlit UI Snapshot

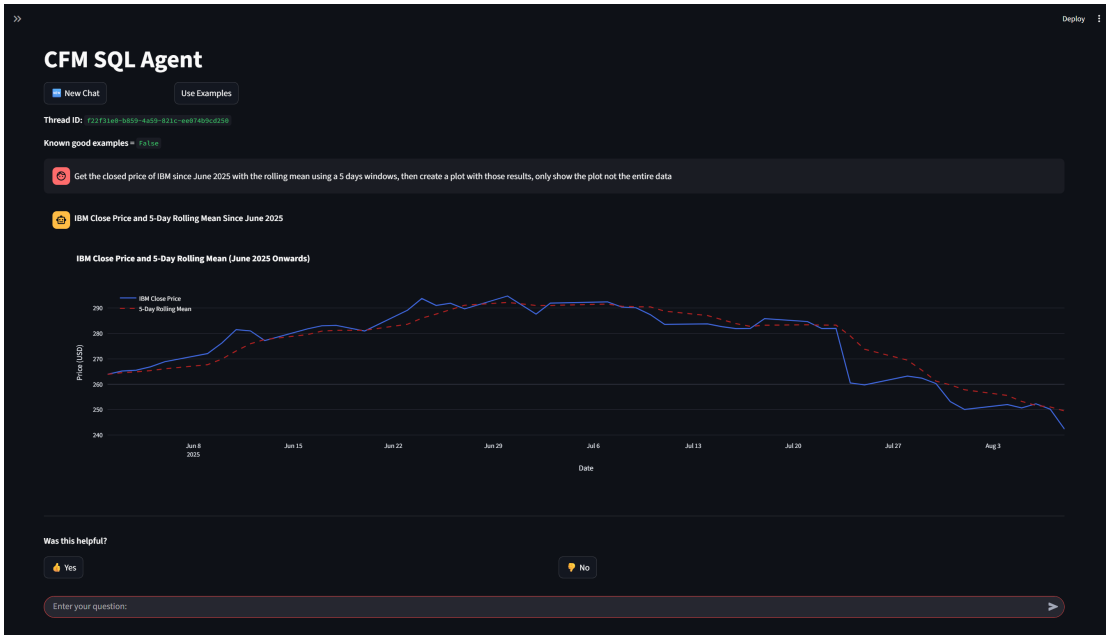

Figure 8. Streamlit interface of the SQL Agent, showing query input, generated SQL, and final answer display.

G. Schema Chunk Sample

Example schema chunk stored in the vector database:

```
CREATE TABLE DATA.markets (
   MKT_ID REAL NOT NULL -- Market Unique ID,
   MKT_CODE TEXT -- Market Code,
   LIMAR TEXT -- Market name,
   MKT_TYPE TEXT -- Market type (MARKET, COMPOSITE, SEGMENT, MTF, ...),
   CTR_ID REAL -- Country Unique ID,
   ZONE TEXT -- Zone,
   ZNE_ID REAL -- Zone Unique ID,
   DABEGIN DATETIME -- Beginning date,
   DAEND DATETIME -- End date,
   PRIMARY KEY (MKT_ID),
   FOREIGN KEY(CTR_ID) REFERENCES COUNTRY(CTR_ID),
   FOREIGN KEY(ZNE_ID) REFERENCES TZONE(ZNE_ID),
   FOREIGN KEY(ZONE) REFERENCES MARKET_ZONE(ZONE)
);
-- The DATA.markets table stores records of various financial markets,
-- linking each market to countries and zones to ensure consistent relational data.
-- Example first rows:
MKT_ID | MKT_CODE | LIMAR | MKT_TYPE | CTR_ID | ZONE | ZNE_ID | DABEGIN | DAEND
--- | --- | --- | --- | --- | --- | --- | --- | ---
15399 | XPAE | PALESTINE SECURITIES EXCHANGE | MARKET | 1293 | A | 124 | 1900-01-01
    00:00:00 | 3000-01-01 00:00:00
15062 | XBUD | BUDAPEST STOCK EXCHANGE | MARKET | 654 | E | 244 | 1900-01-01 00:00:00 |
    3000-01-01 00:00:00
15127 | XSIB | SIBERIAN STOCK EXCHANGE | MARKET | 400 | E | 136 | 1900-01-01 00:00:00 |
    3000-01-01 00:00:00
15559 | XMOL | MOLDOVA STOCK EXCHANGE | MARKET | 1274 | E | 273 | 1900-01-01 00:00:00 |
    3000-01-01 00:00:00
16141 | BURG | BURGUNDY NORDIC MTF | MTF | 401 | E | 269 | 2009-04-09 00:00:00 |
    3000-01-01 00:00:00
```

Figure 9. Example of a schema chunk stored in the vector database, including the SQL DDL, a textual description, and a preview of table rows.

H. Example Agent Plan for a Complex User Request

User Question:

> Give me the prices of IBM, Tesla, Microsoft, Nvidia, Alphabet and Apple since 1st January 2025 along with the rolling mean prices of each of them using a 10 days window and a 5 days window, one column per company and next to each company its corresponding rolling mean columns.

Agent's Query Plan:


```
{
  "steps": [
    {
      "description": "First, retrieve product information for IBM, Tesla, Microsoft, Nvidia,
          Alphabet and Apple from the database to obtain their identifiers.",
      "name": "#E1",
      "tool_name": "Dataref_search",
      "tool_input": "IBM, Tesla, Microsoft, Nvidia, Alphabet, Apple"
    },
    {
      "description": "Retrieve the schemas for tables containing equity price data,
         including product code, price date, close price, currency information, and fields
         needed to compute rolling means.",
      "name": "#E2",
      "tool_name": "Tables",
      "tool_input": "Retrieve schemas for tables containing equity price data including
         prd_code, price date, close price, and currency information"
    },
    {
      "description": "Organize and analyze the retrieved tables and the referential product
          information to identify key columns and joins.",
      "name": "#E3",
      "tool_name": "llm_tables_insights",
      "tool_input": "user question: Give me the prices of IBM, Tesla, Microsoft, Nvidia,
         Alphabet and Apple since 1st january 2025 along with the rolling mean prices of
         each of them using a 10 days windows and a 5 days windows one column per company
         and next to each company its corresponding rolling mean columns use apropiate
         names for the columns find the currency of the prices and indicate it in the
         column name Tables schemas: #E2 Referential db information: #E1"
    },
    {
      "description": "Create an Oracle SQL query to fetch the price history for each
         company since 1st January 2025 along with 10-day and 5-day rolling means, pivoted
         so that each company and its two rolling mean columns appear side by side, with
         currency indicated in the column names.",
      "name": "#E4",
      "tool_name": "Create_query",
      "tool_input": "user question: Give me the prices of IBM, Tesla, Microsoft, Nvidia,
         Alphabet and Apple since 1st january 2025 along with the rolling mean prices of
         each of them using a 10 days windows and a 5 days windows one column per company
         and next to each company its corresponding rolling mean columns use apropiate
         names for the columns find the currency of the prices and indicate it in the
         column name Tables schemas: #E2 Tables Insights: #E3 Referential db information: #
         E1"
    },
    {
      "description": "Execute the SQL query generated to retrieve the initial dataset.",
      "name": "#E5",
      "tool_name": "db_query_tool",
      "tool_input": "#E4"
    },
    {
```

```
    "description": "Verify that the query execution and results correctly match the user
        request, fixing any errors if necessary.",
    "name": "#E6",
    "tool_name": "verify_query",
    "tool_input": "user question: Give me the prices of IBM, Tesla, Microsoft, Nvidia,
        Alphabet and Apple since 1st january 2025 along with the rolling mean prices of
        each of them using a 10 days windows and a 5 days windows one column per company
        and next to each company its corresponding rolling mean columns use apropiate
        names for the columns find the currency of the prices and indicate it in the
        column name Tables schemas: #E2 Tables Insights: #E3 Referential db information: #
        E1 Previous generated query: #E4 Result: #E5"
  },
  {
    "description": "Execute the verified or corrected query to obtain the final accurate
        results.",
    "name": "#E7",
    "tool_name": "db_query_tool",
    "tool_input": "#E6"
  },
  {
    "description": "Generate a user-friendly answer including the SQL query used and the
        resulting pivoted table with prices and rolling means.",
    "name": "#E8",
    "tool_name": "llm_organizer",
    "tool_input": "The user wants the price history since 1st january 2025 for IBM, Tesla,
         Microsoft, Nvidia, Alphabet and Apple along with their 10-day and 5-day rolling
        mean prices in a pivoted format with currency indicated in column names this is
        the SQL query #E6 and its result #E7"
  }
 ]
}
```