COMP4801 Final Year Project 2024-25

FYP24007 Gamified Code Learning Platform with NLP

Final Report

Supervisor: Dr YU Tao

2nd Examiner: Dr ZOU Difan

CHAN Ching Yee	3035930187	BEng(CompSc)
Nafis Ul ISLAM	3035835571	BEng(CompSc)
SO Ki Wai Grace	3035927116	BEng(CompSc)

Author:

SO Ki Wai Grace	3035927116	BEng(CompSc)

Date of submission: 21 Apr 2025

Abstract

The development of beginner-friendly coding education platforms faces significant barriers, including an overemphasis on syntax, a lack of systematic problem-solving techniques, and limited integration of state-of-the-art artificial intelligence (AI). This project aims to bridge the gap between natural language and executable code, addressing the challenges faced by users, particularly those with limited programming experience. By developing a robust platform that facilitates this translation, we seek to eliminate initial programming barriers and enhance accessibility. A comprehensive review and testing of AI-powered tools and benchmarks, including Spider 2.0, were conducted to explore and address issues like hallucination and context management. Our exploration of state-of-the-art models provided insights into their functioning and the methodologies that can improve their performance. Through this work, we aspire to significantly increase the accuracy of AI models in translating natural language to code, ultimately creating a more effective tool for users in diverse programming environments. This endeavour not only aims to empower users but also contributes to the ongoing advancements in the field of natural language processing and code generation.

Acknowledgement

We would like to express my deepest gratitude to our supervisor, Dr YU, Tao for his guidance and support throughout this project. Our gratitude is also extended to Mr. LEI, Fangyu for his time and patience in answering our questions regarding Spider 2.0. Lastly, we would like to thank Professor KAO, Benjamin C.M. and Professor HUANG, Zhiyi for giving us invaluable insight on teaching data structure and algorithms courses, which kickstarted our project.

Table of Contents

Abbreviations

Abbreviation	Full Description
AI	Artificial Intelligence
API	Application Programming Interface
CTE	Common Table Expression
DBMS	Database Management System
FYP	Final Year Project
LLM	Large Language Model
LM	Language Model
NLP	Natural Language Processing
RAG	Retrieval Augmented Generation
SOTA	State-Of-The-Art
SQL	Structured Query Language
TTS	Text-To-SQL

List of Tables

List of Figures

1 Introduction

1.1 Background

With the rise of generative artificial intelligence (GenAI) technology, there is an increasing demand for tech talents. However, breaking into the field of software engineering remains challenging as professionals not only need to possess programming knowledge, also be able to apply their skills to real-world projects. Many existing code learning platforms such as LeetCode [1], HackerRank [2], and CodeWars [3] mainly focus on teaching data structures and algorithms through competitive programming challenges. These challenges require participants to solve predefined problems within strict time limits, prioritizing algorithmic efficiency and computational thinking. However, these platforms often pose obstacles for beginners by assuming prior programming knowledge, which shifts the emphasis from grasping fundamental concepts to mastering syntax and technical proficiency—making it difficult for learners still developing their foundational skills.

Moreover, there is a disconnect between these platforms with larger scale projects and real-world software development practices. These platforms often emphasise on theoretical knowledge by offering standalone coding exercises which is inadequate in preparing for real-world projects. As a result, learners may struggle to apply their knowledge acquired from such platforms to practical scenarios, resulting in a steep learning curve when transitioning from isolated coding exercises to enterprise-scale environments.

In addition, insufficient support and guidance for beginners may make the subject seem overly intimidating and discouraging, potentially leading to many beginners give up on their learning journey. These platforms have yet to fully leverage state-of-the-art (SOTA) artificial intelligence to enrich the learning experience. A one-size-fits-all curriculum is often adopted in code learning platforms that fails to consider the idiosyncratic learning style and pace that the platform users may have. The lack of personalised curriculum and feedback may leave learners feeling isolated and frustrated as they require different methods and pace to grasp concepts and develop skills, further complicating the challenges they face in code learning [4].

1.2 Motivation

The deficiencies outlined above highlight a lack of accessible, beginner-friendly, and advanced AI-driven tools that emphasise conceptual understanding, promote problem-solving techniques, and provide tailored learning experiences. These gaps underscore three key needs in coding education:

1. A platform that enables students to test and learn coding concepts without prior programming knowledge (e.g. syntax)
2. Step-by-step, divide-and-conquer solutions that help learners understand and apply problem-solving techniques to a wide range of problems
3. The integration of AI into learning platforms to assist learners throughout the learning process and provide personalized reviews of their strengths and weaknesses

1.3 Objectives

This project has three objectives:

1. To eliminate initial programming barriers by developing a platform that translates natural language understanding into executable code.
2. To investigate how SOTA models perform natural language to code translation and identify methods that can facilitate this process.
3. To improve the accuracy of SOTA AI models in translating natural language to code.

1.4 Deliverables

This project will deliver two outcomes.

1. A detailed analysis of SOTA AI models running on the Spider 2.0 benchmark, targeting Text-to-SQL abilities.
2. A prompt framework that enhances the capabilities of SOTA AI models in Text-to-SQL generation.

1.5 Report Outline

This report is organized as follows: Section 2 presents the technologies we reviewed. Section 3 details the experiments and tests we conducted. Section **Error! Reference source not found.** describes the difficulties we faced during the project. Section 6 outlines the limitations of the project. Section 6 discusses the possible future works to expand this project. Finally, Section 7 concludes the report and the project.

2 Technology Review

2.1 AI-powered Tools

2.1.1 Overview

AI-powered tools like CursorAI [5]and OpenHands [6] are highly relevant to our FYP as they leverage SOTA AI models to assist developers in solving coding challenges and increasing efficiency. These platforms aim to provide intelligent coding assistance, automate tasks, and improve developer productivity by generating accurate and context-aware code suggestions. By reviewing these tools, we aim to understand how they address developer pain points, their strengths, and limitations, which will help us design a system that intelligently assists users in learning and demonstrates how effectively natural language-to-code generation works at its current stage. This review ensures our FYP aligns with current advancements and contributes meaningfully to the field.

2.1.2 OpenHands

OpenHands, an open-source, community-driven platform designed to develop AI agents that function similarly to human developers. These agents can write code, interact with command-line interfaces, and browse the web. OpenHands provides a safe interaction environment through sandboxed execution, coordination between multiple agents, and an evaluation framework with benchmarks. It supports a variety of tasks and is distributed under the permissive MIT license.

It is noticed that while simple tasks such as creating a basic HTML page with a centred element works well, more advanced prompts such as developing a restaurant booking system results in non-functional or partially functional code that either results in a complete failure or looping indefinitely in an attempt to correct the code, requiring user intervention.

2.1.3 Cursor AI

Cursor AI is a code editor that integrates powerful AI tools that integrates powerful AI tools to help developers write, debug and improve code efficiently. It features predictive auto-complete features that suggests edits to the code based on the existing code that was written, AI-powered chatbots that can also apply code suggestions directly into code, as well as multi-file context provision within the query prompts. It also features an agent that can complete coding tasks across an entire project.

Despite its advanced capabilities, it is noticed that in some highly technical applications such as mathematics and logic intensive code, i.e. linear-time-invariant motion model for a two-wheeled robot, it often fails to convert the systems of linear equations into appropriate matrices, despite the equations being provided in LaTeX formatting. This limitation likely stems from the complexity of parsing and interpreting mathematical notation into structured matrix representations that can be directly used in code implementations that may exceed the large language models' (LLMs) trained capabilities.

2.2 Retrieval Augmented Generation

2.2.1 Overview

Retrieval augmented generation (RAG) is a framework that combines retrieved information that is relevant to the user query from a database containing text questions with SQL answers with an LLM's generative capabilities to provide the LLM with grounded context. This framework is used to mitigate the possibility of the LLM hallucinating on information that it may not have and give accurate responses to user queries. The process involves 5 steps, which shall be discussed in the following subsections.

This framework is highly relevant to our project as it directly addresses one of the primary challenges in text-to-code generation: hallucination. Hallucination occurs when a model

generates plausible-sounding but incorrect outputs, which, in the context of text-to-code generation, can result in invalid or incorrect code logic. This can lead to syntax errors, logical errors, and reduced trust in the system. RAG mitigates hallucination by grounding the model's outputs in a reliable knowledge base, such as SQL-related questions and answers extracted from PDFs, text files, or code files. By parsing these inputs, RAG creates a structured repository of accurate Q&A pairs, which the system uses to retrieve relevant context when answering new questions. This ensures that the model generates accurate SQL queries or code by leveraging verified knowledge, significantly reducing the risk of hallucination. Furthermore, RAG's ability to process multiple input formats allows it to acquire knowledge from diverse sources, making it adaptable and versatile. For our FYP, this enables the system to generate accurate, grounded, and hallucination-free text-to-code outputs, thereby improving reliability, user trust, and overall performance.

Step 1: Parsing PDF document

Initially, a PDF document consisting of text questions on an SQL database and SQL answers is divided into small chunks such as paragraphs or sections using LangChain's [7] text splitter with overlap. This divides the document into smaller, manageable chunks while retaining links to other paragraphs. Parsing PDF documents, however, may introduce potential errors, such as incorrect text extraction, formatting issues, or missing data due to variations in PDF structures. In comparison, reading plain text files or structured formats could provide a more reliable and error-free input source.

Dividing the document in this manner serves two purposes. First, it improves the manageability of the document by breaking it down into digestible pieces, which facilitates easier processing and analysis. Second, by incorporating overlap between chunks, the text splitter ensures that context is preserved, allowing each chunk to maintain its connection to adjacent sections. This preservation of context is essential for maintaining the coherence and flow of information.

Step 2: Apply Agentic Chunking

After the initial PDF parsing, an agentic chunking approach is implemented that extends beyond simple text segmentation. The document undergoes analysis to determine purpose-driven segmentation, specifically focusing on the SQL question-answer structure present in the source material. Using the Spacy [8] natural language processing pipeline, the system performs semantic analysis to identify semantically related paragraphs and sections. This identification process examines the textual content, considering factors such as SQL-specific terminology, question-answer relationships, and topical coherence to establish meaningful boundaries between segments.

The chunking mechanism then implements a dynamic merging process, where the segmentation granularity is adjusted according to the specific use case requirements. For tasks such as retrieval, chunks are optimized to maintain complete question-answer pairs. For summarization purposes, related concepts are grouped together, while classification tasks may require different grouping strategies. Through this adaptive chunking process, each segment retains sufficient contextual information to be meaningful as a standalone unit, while

maintaining a size and structure appropriate for downstream processing tasks. The implementation ensures that the semantic integrity of the SQL content is preserved while optimizing for computational efficiency in subsequent processing stages.

Step 3: Further Combining Chunks Using Semantic Analysis

Following the agentic chunking phase, a secondary semantic analysis process is implemented using Spacy to identify deeper relationships between the existing chunks. This additional analysis layer examines the semantic structures and relationships that may not have been apparent during the initial chunking process. The system evaluates semantic overlap between chunks by analysing their semantic features, syntactic patterns, and SQL-related conceptual relationships. Chunks exhibiting strong semantic similarity or logical continuity are merged to form larger, more comprehensive units. This merging process preserves the thematic coherence of SQL concepts while consolidating related information into meaningful semantic blocks.

Throughout this combination process, careful attention is paid to maintaining an optimal balance between chunk granularity and contextual preservation. The resulting chunks are sized to facilitate efficient retrieval and processing while retaining sufficient context to represent complete SQL concepts and their relationships. This balance ensures that the final document structure remains both computationally manageable and semantically meaningful for subsequent processing stages.

Step 4: Generating Embeddings

Following the semantic combination phase, each refined chunk is transformed into a numerical representation using pre-trained embedding models. Specifically, the system employs OpenAI's text-embedding-ada-002 [9] and nomic-embed-text models [10] to generate dense vector embeddings that capture the semantic characteristics of the SQL content. These embedding models process each chunk, converting the textual SQL questions and answers into high-dimensional vectors. The resulting vector representations encode the semantic relationships and technical concepts present in the text, mapping similar SQL concepts to proximate locations in the vector space.

The embedding process preserves the semantic structure identified in the previous stages while transforming the content into a mathematical format suitable for computational analysis. This vector representation enables precise quantification of semantic similarities between different chunks of SQL content, facilitating efficient retrieval and comparison operations in subsequent processing stages.

Step 5: Building the Vector Database

The generated embeddings are indexed into a FAISS [11] vector database, which provides an optimized infrastructure for large-scale similarity search operations. FAISS, developed by Meta Research, is specifically designed for efficient similarity search and clustering of dense vectors, making it particularly suitable for managing high-dimensional embedding representations of SQL content. The selection of FAISS as the vector database solution is

driven by its efficient indexing structures and algorithms that enable fast approximate nearest neighbour search in high-dimensional spaces.

To store the generated embeddings into this vector database, each vector embedding is indexed alongside associated metadata, including section titles, page numbers, and other contextual information form the source material. This metadata enrichment ensures that retrieved vectors can be mapped back to their original context, providing comprehensive information during the retrieval process.

Enabling RAG

The configured FAISS vector database serves as the foundation for implementing the RAG pipeline. When a user submits a query, it is first transformed into a vector embedding using the same embedding models applied to the source content in section 3.1.4. The system then performs similarity searches within the vector database to identify and retrieve the most semantically relevant SQL chunks. These retrieved chunks serve as contextual information, providing relevant SQL examples, explanations and related concepts that enhance the response generation process.

The retrieved content is then integrated into the Large Language Model's prompt structure. This combination of the user query and relevant retrieved chunks enables the LLM to generate response that are both contextually accurate and grounded in the specific SQL knowledge base. As a result, the system provides precise, context-aware answers that draw upon the comprehensive SQL documentation while maintaining semantic coherence and technical accuracy.

2.3 Benchmarks

2.3.1 Overview

Benchmarks are standardized tests or evaluation methods used to measure and compare the performance, accuracy, and effectiveness of systems, tools, or models. Since the project focuses on natural language-to-code generation, one of the key tasks is to evaluate the effectiveness and correctness of the code generated by AI in solving the tasks described in the user prompts.

Therefore, it is crucial to understand how current benchmarks and technologies assess the accuracy and functionality of AI-generated code. By exploring widely used evaluation metrics and frameworks, we aim to identify reliable methods for measuring the quality, correctness, and practical utility of the generated code. This analysis will support the development of a robust evaluation framework for our project, ensuring that the output codes are executable and meet the intended requirements.

2.3.2 SWE-Bench

SWE-bench [12] is a benchmark dataset designed to evaluate the performance of LLMs on real-world software engineering tasks. Unlike previous benchmarks that focus on small-scale inputs and outputs, SWE-bench addresses larger, more complex scenarios, such as analysing GitHub issues linked to bugs or new features and their corresponding pull requests from 12 popular open-source Python repositories (e.g., Django, Flask). These pull requests often involve changes across multiple files, reflecting real-world development challenges.

The evaluation process involves providing the model with a GitHub issue and relevant files as input, prompting it to generate a patch file specifying code edits. The generated solution is tested by applying the patch to the codebase and running unit and system tests to evaluate correctness. SWE-bench is significant for training and evaluating LLMs on tasks requiring comprehension of large codebases and cross-file relationships. Current results highlight the difficulty of these tasks, with leading models achieving resolution rates as low as 45.20% (Gru, SWE-bench Verified) and 22.6% (Honeycomb, full dataset). These findings emphasize the importance of improving LLM performance on complex code generation tasks.

2.3.3 Spider-2.0

2.3.3.1 Overview

Spider 2.0 [13] is a benchmark designed to evaluate the capabilities of NLP models on generating Structured Query Language (SQL) queries. It includes two types of tasks: Text-to-SQL (TTS) tasks and Code Agent tasks.

TTS tasks are self-contained tasks where a parser generates a SQL query (S) based on a natural language question (Q), a database schema (D), and auxiliary documentation (E). The process can be represented as: $S = f(Q, D, E \mid \theta)$. Here, θ represents the parameters of the parser. Spider 2.0 includes two subsets, Spider 2.0-Lite and Spider 2.0-Snow, both of which consist exclusively of TTS tasks.

Code Agent tasks involve an iterative approach, where an agent modifies code (SQL/Python) (C) based on a natural language TTS question (Q), using a database interface (I) and a codebase (C) (including context, configuration, and documentation). The agent executes and refines the code iteratively until the final result (A) is achieved. This process can be represented as: $A = O_k$, where $O_k = \text{execute}(C, I, Q)$. Unlike the subsets, the full Spider 2.0 benchmark includes Code Agent tasks, making it more comprehensive.

2.3.3.2 Relevance to project scope

One of the ultimate goals or future visions of the project is to develop a reliable framework for text-to-code generation. However, for the scope of this FYP, focusing on Text-to-SQL tasks serves as a more practical and well-defined starting point. Text-to-SQL generation offers a smaller, more constrained scope because SQL is a domain-specific language with simpler

syntax and a well-defined problem space. Unlike general-purpose programming languages, SQL is specifically designed for querying and managing relational databases, which makes its functionality more predictable and limited. These characteristics make text-to-SQL generation more manageable and easier to evaluate within the development time constraints of the FYP.

Therefore, testing Spider 2.0 is essential as it mimics real-world enterprise scenarios, providing a practical framework to evaluate the capabilities of NLP models in generating SQL queries. This benchmark enables us to gain a deeper understanding of Text-to-SQL generation and identify frameworks or architectures that deliver more accurate results, ultimately helping us achieve the project's deliverables.

3 Experiment and Testing

3.1 Spider 2.0-Snow Text-to-SQL Experiment

3.1.1 Introduction

As a starting point, we experimented with Spider 2.0-Snow evaluation dataset, using different combinations of prompt frameworks and LLMs. While many SQL commands are commonly found across different relational database management systems (DBMS), SQL statements across system are often incompatible due to nuances in implementation. The differences form distinct SQL dialects specific to each DBMS. Spider 2.0 and Spider 2.0-lite consist of TTS tasks based on multiple databases, meaning that the LLM to be evaluated must adapt and juggle between dialects. On the other hand, Spider 2.0-Snow is hosted exclusively on Snowflake [14], a cloud-based data warehouse. This means that the LLM only needs to address the Snowflake SQL dialect, making Spider 2.0-Snow less complex and therefore ideal for initial testing in a relatively more controlled setting. Moreover, Spider 2.0-Snow is the only set that do not require any cost for running, making it the most ideal option for testing.

3.1.2 Methodology

3.1.2.1 Pre-processing

Following the instructions from the original repository, a pre-processing step was done on the tasks and their respective database. The info was extracted from all databases and merged into a singular file, where each database's schema components, including tables, columns, and their relationships, were preserved in a structured JSON format. As part of this pre-processing pipeline, all questions were tokenized using the en_core_web_sm-3.5.0 model by Spacy [15]. The questions underwent tokenization, part-of-speech tagging, and lemmatization before being consolidated into the merged file, ensuring consistent token representation across the dataset.

Then, schema linking was performed using a word embedding model Glove-42B [16] to lemmatize the tokens and establish meaningful connections between question elements and database schema components. This process utilized SpiderEncoderV2Preproc with specific

parameters (minimum frequency of 4, maximum count of 5000) to compute schema-column (sc) links that match question tokens with table/column names. The processor also maintains the option to compute column-value (cv) links while preserving the integrity of primary keys and foreign key relationships in the database schema.

3.1.2.2 Prompt frameworks

3.1.2.2.1 Prompt Framework 1: Zero-shot Prompting

Zero-shot prompting [17] is a prompt engineering technique where the model is asked to perform a task directly without any examples or demonstrations, relying solely on its pre-trained knowledge to understand and execute the request. This simulates the real-world scenario where user provides ambiguous question along with only limited context or without offering additional details similar to a database schema.

A simple zero-shot prompt (see Appendix B) was created using a baseline template that structures the input with three key components: the user's question, the database schema information showing available tables and columns, and the foreign key relationships between tables. This structured format helps the language model understand how to convert natural language questions into valid SQL queries while maintaining proper table relationships and database structure. The prompt is deliberately kept minimal and consistent, serving as a baseline to evaluate the model's inherent ability to generate SQL queries without any exemplars. This approach allows us to assess the model's fundamental capability in SQL generation before comparing it with more advanced prompting frameworks mentioned afterwards.

3.1.2.2.2 Prompt Framework 2: DIN-SQL

DIN-SQL [18] is a modular prompt-engineering framework that breaks down complex SQL generation into manageable, sequential stages. The complete process is separated into distinct modules, namely the schema linking module, the classification module, the SQL generation module, and the SQL debugging module. The high modularity enables the framework to handle increasingly complex queries while maintaining high accuracy and traceability at each step.

First, the schema linking module provides examples in addition to the query for the LLM to explain how it can generate the schema links:

1. The natural language question is accompanied by providing examples on how to identify key phrases (e.g., "Find the buildings which have rooms" becomes ["Find the buildings", "rooms with capacity"]) and mapping each phrase to potential database components through pattern matching.
2. Database schemas are also provided in the examples as a structured format where each table and its columns are clearly listed (e.g., "Table classroom, columns = [*,building,room_number,capacity]"), allowing the LLM to have a reference during the mapping process.

3. Foreign key relationships are presented as explicit connections between tables (e.g., "Foreign_keys = [section.building = classroom.building,section.room_number = classroom.room_number]"), enabling the LLM to understand table relationships.
4. Question component analysis is provided with the prompt in the form of guided by step-by-step instructions that prompt the model to:
    - Extract required columns ("we need column = [classroom.building]")
    - Identify necessary tables ("we need these tables = [classroom]")
    - Document required relationships ("we need these Foreign_keys = []")
    - List explicit values ("The set of possible cell values are = [50]")

Next, the classification module categorizes queries as:

- EASY: Single table operations (e.g., "Find all buildings")
- NON-NESTED (Medium): Multiple table operations requiring joins (e.g., "Find buildings with sections")
- NESTED (Hard): Operations requiring subqueries (e.g., "Find departments with highest average salary")

Then, the SQL Generation Module creates a prompt based on classification:

1. EASY queries are handled through direct template application:
    - SELECT template: "SELECT DISTINCT {columns} FROM {table}"
    - WHERE conditions: "WHERE {column} {operator} {value}"
    - Simple aggregations: "SELECT {agg_function}({column}) FROM {table}"
2. NON-NESTED (MEDIUM) queries are processed by:
    - Creating join chains based on foreign keys
    - Applying filtering conditions across joined tables
    - Handling aggregations with proper GROUP BY clauses
3. NESTED (HARD) queries are managed through:
    - Breaking down into subqueries
    - Generating each subquery independently
    - Combining through appropriate nested structures (IN, EXISTS, etc.)

Finally, the SQL debugging module uses a prompt-based approach where it feeds the query, schema information and validation guidelines to a language model for correction. It prepares a

comprehensive context before the LLM by gathering table definitions, column specifications, foreign key relationships and primary key information from the database schema. This information is combined with a set of specific debugging instructions:

1. Use the database values that are explicitly mentioned in the question.
2. Pay attention to the columns that are used for the JOIN by using the Foreign_keys.
3. Use DESC and DISTINCT when needed.
4. Pay attention to the columns that are used for the GROUP BY statement.
5. Pay attention to the columns that are used for the SELECT statement.
6. Only change the GROUP BY clause when necessary (Avoid redundant columns in GROUP BY).
7. Use GROUP BY on one column only.

This approach allows the LLM to be more context-aware during the correction stage.

The entire process is logged with timestamps and detailed information, which include the input question and parsed components, the generated schema links, the classification results, the intermediate SQL representations, the error messages and corrections, and the final validated SQL query. This enables systematic analysis of the framework's performance and identification of common error patterns for continuous improvement.

3.1.2.2.3 Prompt Framework 3: Iterative Prompt Framework - Tarantula

After running the tests on the previous frameworks, we observed the errors caused by the SQL generated in previous attempts. Based on this understanding, we created an iterative prompt framework, which we called Tarantula.

This framework first queries the LLM with specific instructions as follows:

1. Always put column names for a table in double quotes, such as "column_name" (to eliminate case-sensitivity issues)
2. Pay strict attention to the provided database schema (to bring attention to the column names and potentially prevent it from hallucinating)
3. Ensure proper schema linking by using the provided foreign key relationships (to prevent it from making invalid or incorrect foreign key relationships)
4. Match table and column names exactly as they appear in the schema (to avoid naming mismatches)
5. Output SQL directly without any text explanation or comments (to ensure easy extraction of the SQL)
6. Generate SQL based on the database names specified in the schema (to ensure compatibility with Snowflake database system)
7. Consider all table relationships defined in the foreign keys section (to leverage proper table connection and optimize query performance through appropriate join path)

If the generated SQL has an error upon executing on Snowflake (such as syntax errors, invalid column references, or incorrect schema references), the LLM is reprompted with:

1. The original question (to understand the intended query requirements and expected results)
2. The failed SQL query (to identify the problematic SQL syntax or structure causing the error)
3. The specific error message from Snowflake (to pinpoint the exact cause of query failure and guide the correction process)
4. The complete database schema (to verify table structures, column datatypes, and available fields for query construction)
5. The foreign key relationships (to ensure proper table joins and maintain referential integrity in the query)
6. Instructions to generate a new SQL query that fixes the specific error while maintaining the correct logic (to provide clear guidance for query modification while preserving the original query intent and functionality)

This iterative error correction process continues for up to 10 attempts, eliminating incorrect SQLs due to technical errors rather than the LLM's understanding of the question or schema structure. Each attempt is logged with detailed information including timestamps, error messages, and the generated SQL queries for analysis.

An example of the prompts used in Tarantula is listed in Appendix C.

3.1.2.3 Model selection

During this early stage of the project, we wanted to conserve the project budget of $3000 when choosing language models (LM) to evaluate on. Therefore, we started with exploring local LMs, using Ollama [19] and LM Studio [20] to obtain the models and serve as a server providing an Application Programming Interface (API) compatible with the OpenAI API format used in the original code. However, the hardware limitations of our available setups severely limited our options. Larger models such as Llama 3.3 70B Instruct [21] required an exhaustive amount of memory resources (See Figure 1). Even the most recent high-end consumer gaming GPU model from Nvidia, the GeForce RTX 5090, only has 32GB of memory [22] which is not sufficient for storing the whole model in its own memory. This results in the inability to run larger models locally.

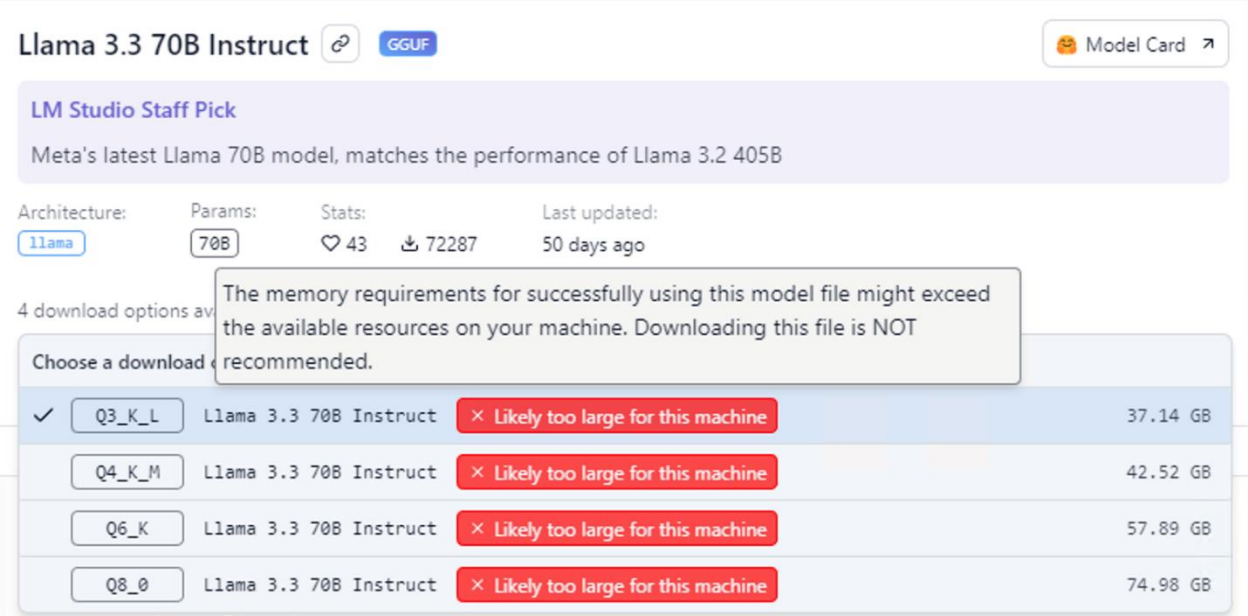


Figure 1 Llama 3.3 70B Quantized version requires at least 37.14GB of memory, which exceeds the memory resources of most single commercial GPUs

Medium sized models, such as Codestral that has 22B parameters [23], could be successfully run with our resources. However, running one task required over 2 hours, which is over 4 times slower than running the tasks on Llama 3.2-3B [24]. Given that Spider 2.0-Snow contains 547 tasks, it would have taken us over 45 days to fully run all tasks, and therefore not feasible to repeatedly experiment using Codestral. Due to the circumstances, Llama 3.2-3B was chosen for its efficiency rather than its abilities or quality in terms of training parameter size.

After running Llama 3.2-3B, the results were not satisfactory nor to our level of expectation, which resulted in us trying to run the experiment using LLMs hosted by OpenAI and Anthropic. To bypass the geographical restriction imposed by OpenAI, a virtual private network (VPN) was used. We were able to access GPT-4o mini which has a score of 87.2% on HumanEval [25], a benchmark for text-to-code generation, and is expected to be high performing in text-to-SQL tasks, given the similar nature of both tasks. However, GPT-4o's strict rate limit of 3 requests per minute (RPM), combined with a daily limit of 200 requests and 40,000 tokens per minute quickly proved insufficient for the dataset size and iterative error-correction approach [26]. The token-heavy prompts containing database schemas and foreign key relationships, often consuming 5,000-10,000 tokens per request, alongside multiple correction attempts per query, made it impractical to process the full dataset under these constraints.

Claude-3.5-Sonnet by Anthropic was also in our consideration, given their 92.0% score on HumanEval [27] which is higher than that of GPT-4o mini. However, since Anthropic requires a model phone number not from Hong Kong which we do not possess nor have the means to, we were not able to access Claude models directly from Anthropic with their official API. As a result, the API to Claude-3.5-Sonnet provided by Poe was used as a last resort. The context window of the aforesaid model is constraint by Poe to approximately 100,000 characters per message [28], which is significantly smaller than Claude's native 200,000-token context

window [29], limiting the amount of schema and error context that could be included in each prompt.

The final two models selected were Llama 3.5-3B and Claude-3.5-Sonnet through Poe.

3.1.2.4 Postprocessing

After getting the output SQLs from the LLMs, a postprocessing script from the original repository was executed. The purpose of this script is to transform the table names in the generated SQL to match the dialect and requirement of the target databases. The schema that is provided as a part of the prompt do not include the information about how these schemas are loaded into the database when executing the SQL for evaluation. This discrepancy is irrelevant to the understanding and generation ability of the LLM as it depends on how the database is setup and is therefore corrected in order to successfully execute the generated SQL.

For example, the way that Snowflake was set up required the table names in the format of fully qualified table names, which contains more information than just the table name. In Figure 2, the LLM generated a SQL that references the table HISTORY_DAY, which is the table name given in the schema. This table name is then converted into GLOBAL_WEATHER_CLIMATE_DATA_FOR_BI.STANDARD_TILE.HISTORY_DAY, as shown in Figure 3.

3.1.2.5 Evaluation

The evaluation metric used in Spider 2.0-Snow is the Execution Accuracy (EX), taken from Spider 1.0 paper [30]. To obtain the EX score of each task, the first step is executing the generated SQL statements and the corresponding gold statements on Snowflake. The execution results are then compared. Only columns that are essential to fulfil the task requirement are compared, which differs for each task. This is because the results from the generated SQL might include additional columns that is not required but also does not affect the overall function of the SQL. If the value of these columns match, it means that the generated SQL statement can satisfy the requirements of the questions, and a score of 1 is therefore given for that task. 0 marks are given if the results do not match. Note that identical results do not guarantee that the intermediate methods and steps in the generated SQL is the same as the gold SQL statement, meaning that the optimization and efficiency of the SQL statements are not evaluated.

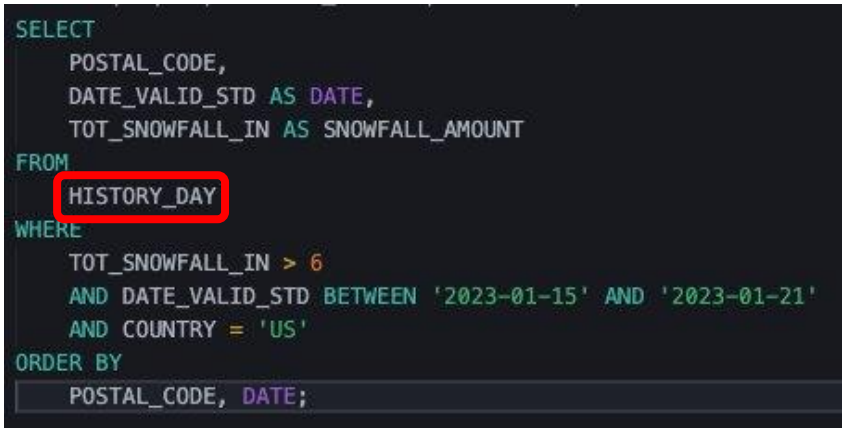


Figure 2 SQL before postprocessing

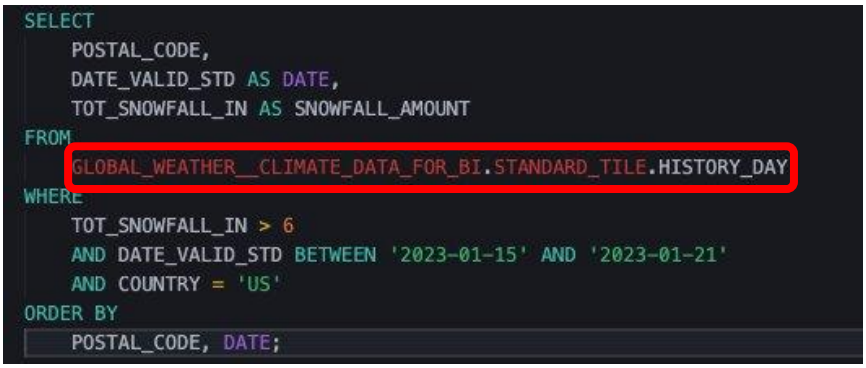


Figure 3 SQL after postprocessing

3.1.3 Results

Upon evaluating the full set of Spider 2.0-Snow tasks generated by Llama 3.2-3B, disappointing results were observed, with scores of 0 regardless of the prompt framework used (see Table 1). Therefore, we isolated and evaluated the easy tasks only, as an attempt to pinpoint the reason for the observed results. There is a total of 47 out of 547 tasks that are labelled as easy in Spider 2.0-Snow tasks.

Table 1 Execution Accuracy (EX) of Llama3.2-3B with different prompt frameworks on all Spider 2.0-Snow tasks

LLM	Zero-shot	DIN-SQL	Tarantula
Llama 3.2-3B	0%	0%	0%

Given that the easy tasks are a subset of the Spider 2.0-Snow tasks, it is not surprising that Llama 3.2-3B showed the same results as the previous attempt, with a score of 0 across the all prompt frameworks (see Table 2).

An improvement on Claude 3.5 Sonnet using the Tarantula prompt framework was observed. Out of 47 tasks, one task successfully passed the test and scored 2.1%, as compared to the scores of 0 using either Zero-shot prompting or DIN-SQL and Llama 3.2-3B. Although the improvement is minimal, it suggests that the Tarantula framework tackles some of the tests successfully due to the iterative error handling, which will be discussed in section 3.1.3.2.

The capability of the model, usually proportional to the parameter size and architecture, plays an important role in the accuracy of the generated results, as indicated by the difference in EX for Claude 3.5 Sonnet and Llama 3.2-3B despite using the same Tarantula prompt framework.

Table 2 Execution Accuracy (EX) of different LLMs and prompt frameworks on Spider 2.0-Snow easy tasks (total = 47)

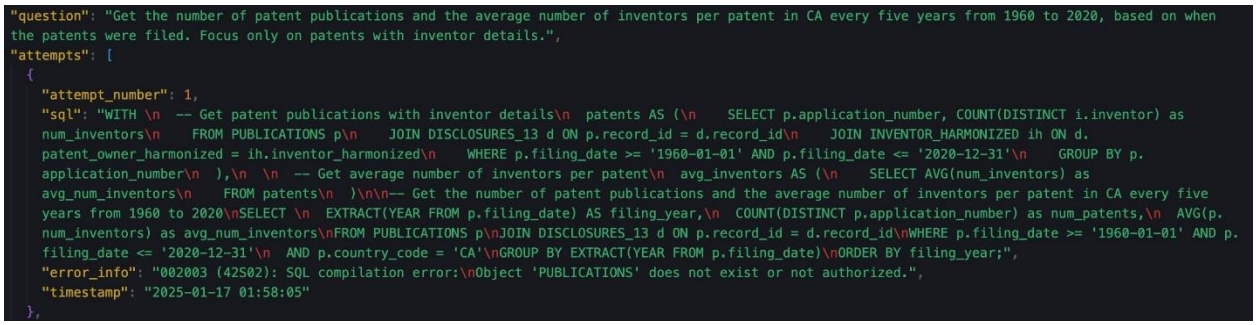
LLM	Zero-shot	DIN-SQL	Tarantula
Llama 3.2-3B	0%	0%	0%
Claude 3.5 Sonnet	0%	0%	2.1%

Error Analysis

3.1.3.1 Error 1: Object Authorization/Existence errors

Upon inspection of the error logs generated during evaluation, the most common error that occurred was: "Object 'X' does not exist or not authorized"

This suggests incorrect table names due to LLM hallucination as the context window of the LLM was significantly less than the required number of tokens, causing it to assume the table name.



```
"question": "Get the number of patent publications and the average number of inventors per patent in CA every five years from 1960 to 2020, based on when the patents were filed. Focus only on patents with inventor details.",
"attempts": [
  {
    "attempt_number": 1,
    "sql": "WITH \n  -- Get patent publications with inventor details\n  patents AS (\n    SELECT p.application_number, COUNT(DISTINCT i.inventor) as num_inventors\n    FROM PUBLICATIONS p\n    JOIN DISCLOSURES_13 d ON p.record_id = d.record_id\n    JOIN INVENTOR_HARMONIZED ih ON d.patent_owner_harmonized = ih.inventor_harmonized\n    WHERE p.filing_date >= '1960-01-01' AND p.filing_date <= '2020-12-31'\n    GROUP BY p.application_number\n  ),\n  \n  -- Get average number of inventors per patent\n  avg_inventors AS (\n    SELECT AVG(num_inventors) as avg_num_inventors\n    FROM patents\n  )\n\n-- Get the number of patent publications and the average number of inventors per patent in CA every five years from 1960 to 2020\nSELECT \n  EXTRACT(YEAR FROM p.filing_date) AS filing_year,\n  COUNT(DISTINCT p.application_number) as num_patents,\n  AVG(p.num_inventors) as avg_num_inventors\nFROM PUBLICATIONS p\nJOIN DISCLOSURES_13 d ON p.record_id = d.record_id\nWHERE p.filing_date >= '1960-01-01' AND p.filing_date <= '2020-12-31'\n  AND p.country_code = 'CA'\nGROUP BY EXTRACT(YEAR FROM p.filing_date)\nORDER BY filing_year;",
    "error_info": "002003 (42S02): SQL compilation error:\nObject 'PUBLICATIONS' does not exist or not authorized.",
    "timestamp": "2025-01-17 01:58:05"
  },
```


Figure 4 Example of Object Authorization/Existence error

3.1.3.2 Error 2: Syntax errors

Another common error that caused issues in executing the generated SQL statements is syntax errors. The syntax errors prevented the SQL being executed by Snowflake, causing failures in evaluation.

Figure 5 shows an example of a syntax issue in the generated SQL for task sf_bq029. The proper syntax for the ORDER BY clause in descending order is "ORDER BY <column> DESC". The SQL in Figure 5, however, has the order of "DESC ORDER BY <column>" instead. One can argue that the line break, as indicated by the "\n" after DESC, suggests that the descending command is associated to the previous keyword GROUP BY. However, the GROUP BY keyword does not have the argument DESC, which also results in a syntax error.

A possible countermeasure for this error could be the implementation of a feedback loop for errors, which is the aim of Tarantula. The active correction of execution errors during the iterations improves the accuracy of the generated SQL statements, as supported by the improvement shown by Claude 3.5 Sonnet with the Tarantula framework.

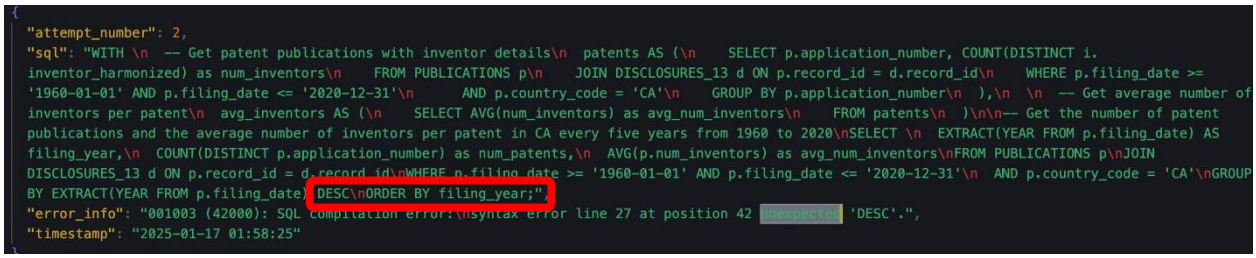


Figure 5 Example of syntax issue in the generated SQL, for task sf_bq029

3.1.3.3 Error 3: Context window length issue

The prompt to some tasks exceeded the maximum token count of the models. For example, in Figure 6, the prompt to the task that involves the database CENSUS_BUREAU_ACS_2 consists of 477539 tokens. However, the model in use, Llama 3.2-3B, only supports a maximum of 131072 tokens (See Figure 7). The model was unable to generate a SQL for that task and therefore failed the evaluation of those tasks. 13 of such cases were found in the Spider 2.0-Lite dataset.

```
[2025-01-12 20:09:17][INFO][LM STUDIO SERVER] Running chat completion on
conversation with 1 messages.
[2025-01-12 20:09:18][DEBUG] sampling:
logits -> logit-bias -> penalties -> greedy
generate: n_ctx = 70720, n_batch = 512, n_predict = 1000, n_keep = 477539

[2025-01-12 20:09:19][DEBUG][LLM Engine bindings] PredictWorker::Execute -
caught exception: Trying to keep the first 477539 tokens when context the
overflows. However, the model is loaded with context length of only 70692
tokens, which is not enough. Try to load the model with a larger context
length, or provide a shorter input
```

Figure 6 Error log of generating the task associated with the database CENSUS_BUREAU_ACS_2

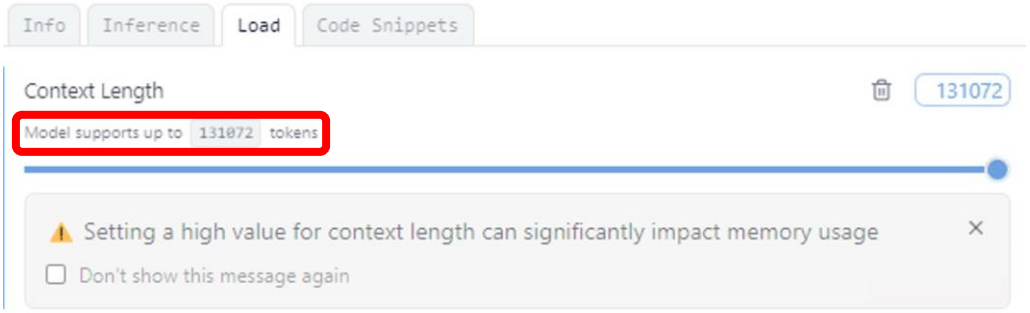


Figure 7 Maximum token count of Llama 3.2-3B

3.1.3.4 Error 4: Lost in the middle issue

It has been reported that LLM perform worse when it has to access the relevant information within a long context [31]. In our experimentation with Spider 2.0-Snow tasks, we discovered this issue to be relevant and possibly impacted the performance of the models.

Figure 8 shows the complete database schema that is sent to the LLM as a part of the context for task sf001. However, among the three tables in the schema, only the table HISTORY_DAY is needed to fulfill the requirements of the question (see Figure 9), meaning that the information of the tables CLIMATOLOGY_DAY and FORECAST_DAY is redundant. Furthermore, out of the extensive number of columns in HISTORY_DAY (over 50 columns), the query only requires four specific columns: country, postal_code, date_valid_std, and tot_snowfall_in. Under the current processing system, there is no filter to eliminate such redundant information, as a result, important information (i.e. columns of HISTORY_DAY) may be lost in the middle of the long context.

A possible method to tackle this issue is to create an intermediary step to identify only the required schema and the columns before the generation of the final SQL statement using keywords from the user query. This approach could help with the focus of LLM by reducing the schema context from over 50 columns to just the 4 required ones, and from 3 tables to 1 with the additional benefit of shortening the required context window, which potentially eliminating the issue mentioned in section 3.1.3.3 and contributes to a more efficient and accurate output.

3.1.3.5 Potential error

When the correction step is sent to the LLM using Tarantula, it often sends the postprocessed SQL containing the fully qualified table format (i.e. <Database>.<Table>.<Schema>.<Column> format), but such information was not sent to the LLM originally since it is converted in the postprocessing step. This might cause the LLM to hallucinate due to its inability to associate the table name to the fully qualified table name.

One way to tackle this issue is to convert the table name to the fully qualified format in the pre-processing stage according to the hosting database (i.e. Snowflake for Spider 2.0-Snow). An alternative is to associate the table name to the fully qualified table name in the iterative error correction section in Tarantula.

```
Database Schema:
Table CLIMATOLOGY_DAY, columns = [AVG_OF__DAILY_MAX_TEMPERATURE_WETBULB_F,AVG_OF__DAILY_AVG_TEMPERATURE_AIR_F,AVG_OF__DAILY_AVG_PRESSURE_SURFACE_MB,
STD_OF__DAILY_MAX_HUMIDITY_RELATIVE_PCT,STD_OF__POS_DAILY_TOT_SNOWFALL_IN,POSTAL_CODE,FRQ_OF__POS_DAILY_TOT_SNOWFALL_IN_PCT,STD_OF__DAILY_MAX_WIND_SPEED_10M_MPH,
AVG_OF__DAILY_MIN_TEMPERATURE_AIR_F,FRQ_OF__DAILY_TOT_PRECIPITATION_ZERO_IN_PCT,STD_OF__POS_DAILY_SNOWDEPTH_IN,AVG_OF__DAILY_MIN_CLOUD_COVER_TOT_PCT,
AVG_VEC_OF__DAILY_AVG_VEC_WIND_DIRECTION_10M_DEG,AVG_OF__DAILY_AVG_CLOUD_COVER_TOT_PCT,AVG_OF__POS_DAILY_TOT_PRECIPITATION_IN,
AVG_OF__DAILY_MAX_PRESSURE_SURFACE_MB,AVG_OF__DAILY_MAX_RADIATION_SOLAR_TOTAL_WPM2,STD_OF__DAILY_MAX_CLOUD_COVER_TOT_PCT,
PCT_FRQ_OF__DAILY_TOT_SNOWFALL_250_GTR_IN_PCT,STD_OF__DAILY_MIN_PRESSURE_MEAN_SEA_LEVEL_MB,AVG_OF__DAILY_AVG_RADIATION_SOLAR_TOTAL_WPM2,
STD_OF__DAILY_MAX_HUMIDITY_SPECIFIC_GPKG,PCT_FRQ_OF__DAILY_TOT_SNOWFALL_025_050_IN_PCT,FRQ_OF__DAILY_TOT_PRECIPITATION_250_500_IN_PCT,
PCT_FRQ_OF__DAILY_TOT_SNOWFALL_100_150_IN_PCT,STD_OF__DAILY_MAX_TEMPERATURE_FEELSLIKE_F,STD_OF__DAILY_AVG_CLOUD_COVER_TOT_PCT,
AVG_OF__DAILY_AVG_HUMIDITY_RELATIVE_PCT,AVG_OF__DAILY_MIN_TEMPERATURE_DEWPOINT_F,FRQ_OF__DAILY_TOT_PRECIPITATION_001_010_IN_PCT,
STD_OF__DAILY_MIN_TEMPERATURE_FEELSLIKE_F,STD_OF__DAILY_MIN_PRESSURE_SURFACE_MB,AVG_OF__DAILY_AVG_WIND_SPEED_10M_MPH,STD_OF__DAILY_AVG_WIND_SPEED_10M_MPH,
STD_OF__DAILY_AVG_PRESSURE_SURFACE_MB,FRQ_OF__POS_DAILY_TOT_PRECIPITATION_IN_PCT,STD_OF__DAILY_AVG_TEMPERATURE_WETBULB_F,STD_OF__DAILY_MAX_TEMPERATURE_DEWPOINT_F,
STD_OF__DAILY_MIN_TEMPERATURE_DEWPOINT_F,FRQ_OF__DAILY_TOT_PRECIPITATION_500_GTR_IN_PCT,AVG_OF__POS_DAILY_TOT_SNOWFALL_IN,
AVG_OF__DAILY_AVG_HUMIDITY_SPECIFIC_GPKG,AVG_VEC_OF__DAILY_AVG_VEC_WIND_DIRECTION_100M_DEG,STD_OF__DAILY_AVG_TEMPERATURE_DEWPOINT_F,
STD_OF__DAILY_MAX_PRESSURE_SURFACE_MB,AVG_OF__DAILY_MIN_WIND_SPEED_10M_MPH,AVG_OF__DAILY_MAX_HUMIDITY_RELATIVE_PCT,STD_OF__DAILY_MAX_TEMPERATURE_AIR_F,
PCT_FRQ_OF__DAILY_TOT_SNOWFALL_150_250_IN_PCT,STD_OF__DAILY_MAX_RADIATION_SOLAR_TOTAL_WPM2,FRQ_OF__DAILY_TOT_PRECIPITATION_025_050_IN_PCT,
AVG_OF__DAILY_MAX_TEMPERATURE_FEELSLIKE_F,AVG_OF__DAILY_MAX_PRESSURE_MEAN_SEA_LEVEL_MB,PCT_FRQ_OF__DAILY_TOT_SNOWFALL_ZERO_IN_PCT,
AVG_OF__DAILY_TOT_RADIATION_SOLAR_TOTAL_WPM2,STD_OF__DAILY_AVG_TEMPERATURE_FEELSLIKE_F,STD_OF__POS_DAILY_TOT_PRECIPITATION_IN,
STD_OF__DAILY_AVG_HUMIDITY_RELATIVE_PCT,AVG_OF__DAILY_AVG_TEMPERATURE_WETBULB_F,STD_OF__DAILY_AVG_HUMIDITY_SPECIFIC_GPKG,
STD_OF__DAILY_MAX_PRESSURE_MEAN_SEA_LEVEL_MB,AVG_OF__DAILY_AVG_TEMPERATURE_DEWPOINT_F,AVG_OF__DAILY_MIN_TEMPERATURE_FEELSLIKE_F,
AVG_OF__DAILY_AVG_TEMPERATURE_FEELSLIKE_F,AVG_OF__DAILY_MIN_PRESSURE_MEAN_SEA_LEVEL_MB,COUNTRY,STD_OF__DAILY_MIN_CLOUD_COVER_TOT_PCT,
STD_OF__DAILY_MIN_WIND_SPEED_10M_MPH,AVG_OF__DAILY_MAX_WIND_SPEED_10M_MPH,FRQ_OF__DAILY_TOT_PRECIPITATION_100_250_IN_PCT,STD_OF__DAILY_MAX_TEMPERATURE_WETBULB_F,
AVG_OF__DAILY_MIN_RADIATION_SOLAR_TOTAL_WPM2,STD_OF__DAILY_AVG_PRESSURE_MEAN_SEA_LEVEL_MB,STD_OF__DAILY_MIN_HUMIDITY_SPECIFIC_GPKG,DOY_STD,
AVG_OF__DAILY_MIN_PRESSURE_SURFACE_MB,AVG_OF__DAILY_MAX_HUMIDITY_SPECIFIC_GPKG,AVG_OF__DAILY_AVG_PRESSURE_MEAN_SEA_LEVEL_MB,
FRQ_OF__DAILY_TOT_PRECIPITATION_050_100_IN_PCT,FRQ_OF__DAILY_TOT_PRECIPITATION_010_025_IN_PCT,PCT_FRQ_OF__DAILY_TOT_SNOWFALL_010_025_IN_PCT,
AVG_OF__POS_DAILY_SNOWDEPTH_IN,STD_OF__DAILY_TOT_RADIATION_SOLAR_TOTAL_WPM2,AVG_OF__DAILY_MIN_HUMIDITY_RELATIVE_PCT,STD_OF__DAILY_MIN_HUMIDITY_RELATIVE_PCT,
STD_OF__DAILY_AVG_RADIATION_SOLAR_TOTAL_WPM2,AVG_OF__DAILY_MAX_TEMPERATURE_AIR_F,AVG_VEC_OF__DAILY_AVG_VEC_WIND_DIRECTION_80M_DEG,
AVG_OF__DAILY_MIN_HUMIDITY_SPECIFIC_GPKG,STD_OF__DAILY_AVG_TEMPERATURE_AIR_F,STD_OF__DAILY_MIN_TEMPERATURE_WETBULB_F,
PCT_FRQ_OF__DAILY_TOT_SNOWFALL_050_100_IN_PCT,AVG_OF__DAILY_MIN_TEMPERATURE_WETBULB_F,AVG_OF__DAILY_MAX_CLOUD_COVER_TOT_PCT,STD_OF__DAILY_MIN_TEMPERATURE_AIR_F,
STD_OF__DAILY_MIN_RADIATION_SOLAR_TOTAL_WPM2,PCT_FRQ_OF__DAILY_TOT_SNOWFALL_001_010_IN_PCT,AVG_OF__DAILY_MAX_TEMPERATURE_DEWPOINT_F]
Table FORECAST_DAY, columns = [MIN_TEMPERATURE_DEWPOINT_2M_F,POSTAL_CODE,AVG_PRESSURE_MEAN_SEA_LEVEL_MB,PROBABILITY_OF_SNOW_PCT,AVG_WIND_SPEED_80M_MPH,
AVG_TEMPERATURE_WINDCHILL_2M_F,AVG_TEMPERATURE_FEELSLIKE_2M_F,AVG_CLOUD_COVER_TOT_PCT,MAX_PRESSURE_2M_MB,MAX_TEMPERATURE_HEATINDEX_2M_F,DATE_VALID_STD,
AVG_RADIATION_SOLAR_TOTAL_WPM2,TIME_INIT_UTC,MAX_HUMIDITY_RELATIVE_2M_PCT,MAX_TEMPERATURE_DEWPOINT_2M_F,MIN_WIND_SPEED_10M_MPH,AVG_PRESSURE_2M_MB,
MIN_WIND_SPEED_100M_MPH,AVG_WIND_DIRECTION_10M_DEG,MAX_HUMIDITY_SPECIFIC_2M_GPKG,TOT_RADIATION_SOLAR_TOTAL_WPM2,MAX_CLOUD_COVER_TOT_PCT,
MIN_RADIATION_SOLAR_TOTAL_WPM2,MIN_TEMPERATURE_AIR_2M_F,AVG_HUMIDITY_RELATIVE_2M_PCT,AVG_TEMPERATURE_AIR_2M_F,AVG_WIND_SPEED_100M_MPH,MIN_PRESSURE_2M_MB,
MAX_WIND_SPEED_10M_MPH,MAX_PRESSURE_MEAN_SEA_LEVEL_MB,MIN_CLOUD_COVER_TOT_PCT,AVG_TEMPERATURE_WETBULB_2M_F,MAX_TEMPERATURE_FEELSLIKE_2M_F,MAX_WIND_SPEED_80M_MPH,
MAX_RADIATION_SOLAR_TOTAL_WPM2,PROBABILITY_OF_PRECIPITATION_PCT,MIN_TEMPERATURE_WETBULB_2M_F,MIN_TEMPERATURE_WINDCHILL_2M_F,DOY_STD,AVG_WIND_DIRECTION_100M_DEG,
AVG_TEMPERATURE_HEATINDEX_2M_F,AVG_WIND_DIRECTION_80M_DEG,MIN_TEMPERATURE_FEELSLIKE_2M_F,MAX_TEMPERATURE_WETBULB_2M_F,MIN_HUMIDITY_RELATIVE_2M_PCT,
TOT_SNOWFALL_IN,AVG_TEMPERATURE_DEWPOINT_2M_F,MAX_TEMPERATURE_WINDCHILL_2M_F,TOT_PRECIPITATION_IN,MAX_WIND_SPEED_100M_MPH,MIN_TEMPERATURE_HEATINDEX_2M_F,
MAX_TEMPERATURE_AIR_2M_F,MIN_PRESSURE_MEAN_SEA_LEVEL_MB,AVG_WIND_SPEED_10M_MPH,MIN_HUMIDITY_SPECIFIC_2M_GPKG,AVG_HUMIDITY_SPECIFIC_2M_GPKG,MIN_WIND_SPEED_80M_MPH,
COUNTRY]
Table HISTORY_DAY, columns = [MIN_HUMIDITY_RELATIVE_2M_PCT,AVG_HUMIDITY_SPECIFIC_2M_GPKG,AVG_PRESSURE_MEAN_SEA_LEVEL_MB,MIN_TEMPERATURE_WETBULB_2M_F,
MAX_TEMPERATURE_HEATINDEX_2M_F,COUNTRY,AVG_WIND_DIRECTION_80M_DEG,TOT_PRECIPITATION_IN,MIN_TEMPERATURE_FEELSLIKE_2M_F,AVG_TEMPERATURE_WINDCHILL_2M_F,
AVG_WIND_SPEED_10M_MPH,MAX_WIND_SPEED_80M_MPH,MIN_WIND_SPEED_10M_MPH,TOT_SNOWDEPTH_IN,MAX_RADIATION_SOLAR_TOTAL_WPM2,MAX_PRESSURE_TENDENCY_2M_MB,TOT_SNOWFALL_IN,
AVG_PRESSURE_TENDENCY_2M_MB,MIN_RADIATION_SOLAR_TOTAL_WPM2,MIN_HUMIDITY_SPECIFIC_2M_GPKG,AVG_HUMIDITY_RELATIVE_2M_PCT,AVG_RADIATION_SOLAR_TOTAL_WPM2,
AVG_WIND_SPEED_80M_MPH,DATE_VALID_STD,MAX_CLOUD_COVER_TOT_PCT,MAX_WIND_SPEED_100M_MPH,MIN_TEMPERATURE_DEWPOINT_2M_F,AVG_TEMPERATURE_FEELSLIKE_2M_F,
AVG_TEMPERATURE_HEATINDEX_2M_F,DOY_STD,MIN_PRESSURE_2M_MB,MAX_TEMPERATURE_WETBULB_2M_F,AVG_TEMPERATURE_AIR_2M_F,MIN_WIND_SPEED_100M_MPH,
MAX_TEMPERATURE_WINDCHILL_2M_F,MAX_TEMPERATURE_FEELSLIKE_2M_F,MAX_TEMPERATURE_AIR_2M_F,AVG_TEMPERATURE_WETBULB_2M_F,AVG_WIND_SPEED_100M_MPH,
MIN_WIND_SPEED_80M_MPH,MAX_PRESSURE_2M_MB,AVG_WIND_DIRECTION_100M_DEG,POSTAL_CODE,MIN_PRESSURE_MEAN_SEA_LEVEL_MB,MAX_PRESSURE_MEAN_SEA_LEVEL_MB,
MIN_PRESSURE_TENDENCY_2M_MB,MIN_TEMPERATURE_WINDCHILL_2M_F,MIN_TEMPERATURE_HEATINDEX_2M_F,MAX_TEMPERATURE_DEWPOINT_2M_F,AVG_CLOUD_COVER_TOT_PCT,
MAX_HUMIDITY_RELATIVE_2M_PCT,MIN_CLOUD_COVER_TOT_PCT,MAX_HUMIDITY_SPECIFIC_2M_GPKG,MAX_WIND_SPEED_10M_MPH,AVG_PRESSURE_2M_MB,AVG_TEMPERATURE_DEWPOINT_2M_F,
MIN_TEMPERATURE_AIR_2M_F,TOT_RADIATION_SOLAR_TOTAL_WPM2,AVG_WIND_DIRECTION_10M_DEG]
```

Figure 8 Complete database schema for task sf001

```sql
WITH timestamps AS
(
    SELECT
        DATE_TRUNC(year,DATEADD(year,-1,DATE '2024-08-29')) AS ref_timestamp,
        LAST_DAY(DATEADD(week,2 + CAST(WEEKISO(ref_timestamp) != 1 AS INTEGER),ref_timestamp),week) AS end_week,
        DATEADD(day, day_num - 7, end_week) AS date_valid_std
    FROM
    (
        SELECT
            ROW_NUMBER() OVER (ORDER BY SEQ1()) AS day_num
        FROM
            TABLE(GENERATOR(rowcount => 7))
    )
)
SELECT
    country,
    postal_code,
    date_valid_std,
    tot_snowfall_in
FROM
    GLOBAL_WEATHER__CLIMATE_DATA_FOR_BI.standard_tile.history_day
NATURAL INNER JOIN
    timestamps
WHERE
    country='US' AND
    tot_snowfall_in > 6.0
ORDER BY
    postal_code,date_valid_std
;
```

Figure 9 Gold SQL for task sf001, requiring data only from the table history_day

3.1.4 Summary

The evaluation results clearly indicate that the inherent abilities of LLMs alone are insufficient for handling TTS tasks in convoluted environments. A robust framework is essential to address the challenges posed by these tasks effectively. Furthermore, the capabilities of the base model play a critical role in ensuring success in TTS tasks, emphasizing the importance of selecting reliable models.

The complexity of database environments, often characterized by extensive schemas and large tables, introduces a significant challenge in managing context. The excessively long context can result in a "lost in the middle" issue, where key information becomes inaccessible or overlooked during processing. The limitations of the model's context window also hinder their performance. These problems indicate the need for breaking down tasks into smaller, more manageable subtasks to enhance accuracy and efficiency in handling intricate queries.

While the Tarantula framework was developed to address specific issues such as syntax errors and schema linking issues, it has proven inadequate in resolving more intricate errors. This underscores the necessity of developing a more sophisticated and comprehensive error-handling framework that can effectively tackle the diverse challenges encountered in TTS tasks.

As an attempt to better support the LLMs ability to navigate complex database environments and produce more accurate results, we continued to the next experiment incorporating advanced strategies to a refined framework, Spider-Agent.

3.2 Spider-Agent Experiment

3.2.1 Introduction

As a part of our goal discussed with our supervisor, we set up an advanced experiment playing around with the Spider-Agent code agent framework, as an extension of the previous experiment. In each run, we adopted different workflows and evaluated the generated SQLs, in hopes of finding ways to improve the abilities of LLM on TTS tasks through the help of an agent. Given that Spider-Agent+Claude-3.5-Sonnet scored 25.00% on the easy subset of Spider 2.0-Snow, much higher than the 2.1% from Tarantua+Claude-3.5-Sonnet, Spider-Agent is expected to help us overcome the limitations observed in previous approaches.

3.2.1.1 Spider-Agent

Spider-Agent is a code agent framework developed by the authors of the Spider 2.0 benchmark. This framework adheres to ReAct, a prompting framework created by Yao, et al. [32]. Instead of relying on the LLM to generate reasoning through a chain of thought, which is less reliable and more prone to hallucinations [32], the prompt framework in Spider-Agent models decompose complex problems into manageable steps of reasoning (Reasoning), action (Action), and observation (Observation). These steps are repeated iteratively, each turn allowing the agent to observe the state after each action, such that the reasoning in the next iteration is based

on the observation (see Figure 10). The agent is permitted to choose an action from a limited set of actions called action space (see Table 3), designed to maintain a focus on database interactions though command line interfaces.

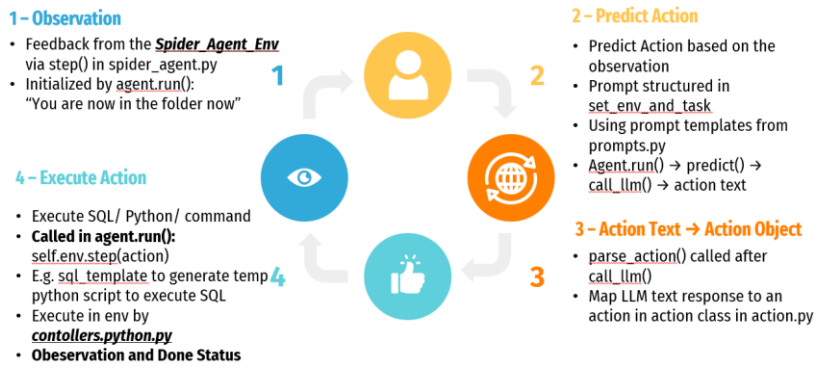


Figure 10 Illustration of workflow of each Spider-agent iteration. Based on the given observation of the current state, the agent prompts the LLM for the next action. Upon receiving the action, the action is executed. The agent observes the current state and ends current iteration and starts next iteration if no terminate action is called.

Table 3 The action space of Spider-Agent, a list of permitted actions [13, Table 19].

Action	Description
BASH	Executes shell commands, such as checking file information, running code, or executing DBT commands.
CreateFile	Creates a new file with specified content.
EditFile	Edits or overwrites the content of an existing file.
ExecuteSQL	Executes a SQL query on BigQuery/Snowflake, with an option to print or save the results.
GetTables	Retrieves all table names and schemas from a specified BigQuery/Snowflake dataset.
GetTabInfo	Retrieves detailed column information for a specific table in BigQuery/Snowflake.
SampleRows	Samples a specified number of rows from a BigQuery/Snowflake table and saves them as JSON.
FAIL	Agent decides the task is infeasible.
Terminate	Agent decides the task is finished.

3.2.2 Methodology

The full set of Spider 2.0-Snow tasks were run and evaluated, which consisted of 547 tasks. The maximum number of iterations allowed for each task is 20, following the code from the original repository.

3.2.2.1 Baseline

We started with running the original code for Spider-Agent, using different LLMs to power the agent each time. This vanilla run will act as our baseline to compare the effectiveness of the

different workflows we created on top of the original code. See Appendix D for the full system prompt from the original Spider 2.0 repository.

3.2.2.2 Workflow 1

From running the baseline and the previous experiment, we noticed that there is a portion of error that stems from syntax errors. The tasks we were running were exclusively based on the Snowflake database, meaning that the LLM only needed to take care of a singular SQL dialect. Despite of this, syntax errors still occurred and multiple iterations were wasted to fix the syntax errors. As an attempt to reduce syntax errors and to eliminate dialectal problems, we created a new action called GetDocs.

There are two parts to this workflow. First, a Python script was created to crawl the documentation of Snowflake database [33] to get relevant information of accepted commands. Each command has its own page which listed the exact syntax required for each command, parameters and usage. To reduce the context length, only a summary of such information was scrapped and saved to a json file. However, to prevent critical information being left out in the summary, the link to the pages of each command is also saved to the json file. This way, the agent could gather all the specifics of the command they are looking for directly from the website if additional information is needed, instead of receiving a long list of unnecessary information from unrelated commands.

Then a new action, GetDocs, was created. By using this action, the agent first search for a command in the json file, and from the top search results, corresponding information is then appended to observation. As mentioned in section 3.2.1.1, observation is the prompt for the next iteration, meaning that the information of the commands are sent as context for the next iteration. (See Figure 11 for an illustration of the complete workflow)

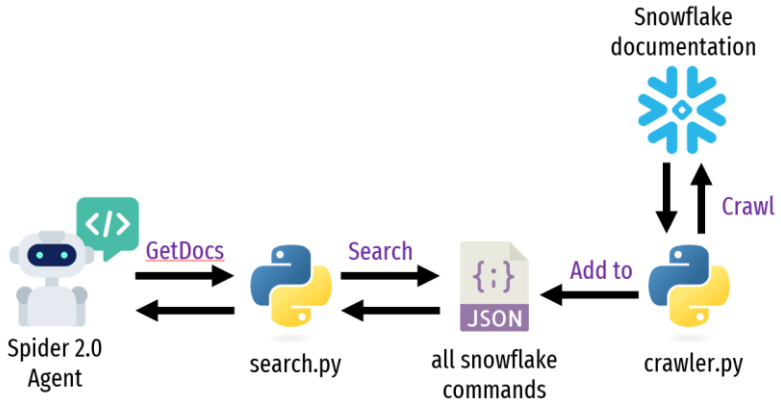


Figure 11 Illustration of workflow 1

To allow the agent to utilize the GetDocs action, a new instruction was also added to the system prompt to indicate when to use GetDocs (See Figure 12). This new instruction told the agent to search the json file for the command that has been repeatedly causing SQL errors.

```
7. If you keep encountering repeated SQL errors, consult the documentation using the GetDocs action. Limit
your search to one or two keywords to narrow down results, then use a specific link search with
verbose=False to gather more detailed information.
```

Figure 12 New instruction in the prompt for workflow 1

The complete system prompt for workflow 1 is available in Appendix E.

3.2.2.3 Workflow 2

In this workflow, we expanded the possibilities of workflow 1 by exploring the abilities of grounding. Grounding is a technique to seek for external knowledge from the internet. Such information could be past the training cut-off date of the LLM. For example, SQL commands could be modified throughout the years in different versions. Grounding could avoid errors due to usage of unsupported older versions of SQL commands by always fetching updated information from the internet. Grounding also enables the access to forums such as Stack Overflow where people might have encountered similar issues and gotten answers to fix those issues. Previous studies also found that grounding helps reduce bias and hallucinations [34] of the LLM output, leading to a more reliable result.

To achieve grounding, we used Gemini 2.0 Flash as an intermediate LLM, for its grounding with Google Search [35] function. Figure 13 illustrates workflow 2, where upon GetDocs action is called by the agent, the intermediate LLM then uses grounding to search the internet and returns the search result to the agent. The search result is added to the observation of next iteration.

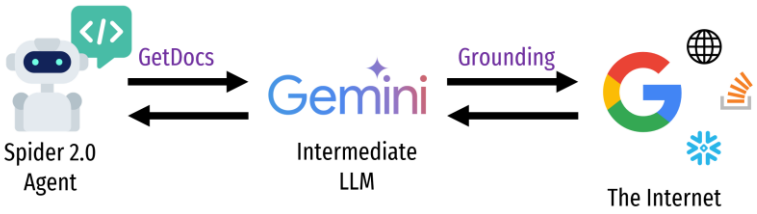


Figure 13 Illustration of workflow 2

The system prompt from workflow 1 was also modified. One limitation we found from workflow 1 is in the ambiguity of the prompt we added. We told the agent to consult the documentation whenever there was repeating SQL errors (See Figure 12). However, we overlooked the fact that SQL errors is not exclusive to syntax errors. As a result, we noticed that the agent has been consulting the documentation even when it encounters errors that involved table or column names. Such errors is classified as schema linking errors in the original Spider 2.0 paper [13], as there is likely a mismatch between the name used and what is actually in the database of the task. When encountering such errors, documentation of the available Snowflake SQL commands is not useful, and the schema documents of the task should be consulted instead. To potentially fix this issue, we refined the prompt to add in several more specific instructions. For example, Figure 14 showed an instance where a specific example of when to use GetDocs is given, which includes the exact error message received when syntax errors occurred.

Another modification to the prompt is the addition of a tip to encourage the usage of GetDocs action and Google Search (See Figure 15), in hopes of guiding the agent to find solutions from reliable external knowledge, instead of solely relying on its own limited knowledge base.

The Appendix F contains the full system prompt of workflow 2.

```
- When you get an error like: "001003 (42000): SQL compilation error: syntax error line 1 at position 706 unexpected '<EOF>'"
  Use: GetDocs(search_query="Snowflake SQL error unexpected EOF debugging examples", verbose=True)
```

Figure 14 Specific example of syntax error message and solution to fix the error

```
3. When encountering syntax errors or other problems, immediately use GetDocs to search for solutions.
Don't hesitate to use Google search via GetDocs whenever you're uncertain about syntax or encounter errors.
```

Figure 15 Tip added to encourage using GetDocs and grounding when encountering errors

3.2.2.4 Workflow 3

In workflow 3, we built on top of workflow 2 by adding dynamic prompting and self-critique mechanisms (see Figure 16)

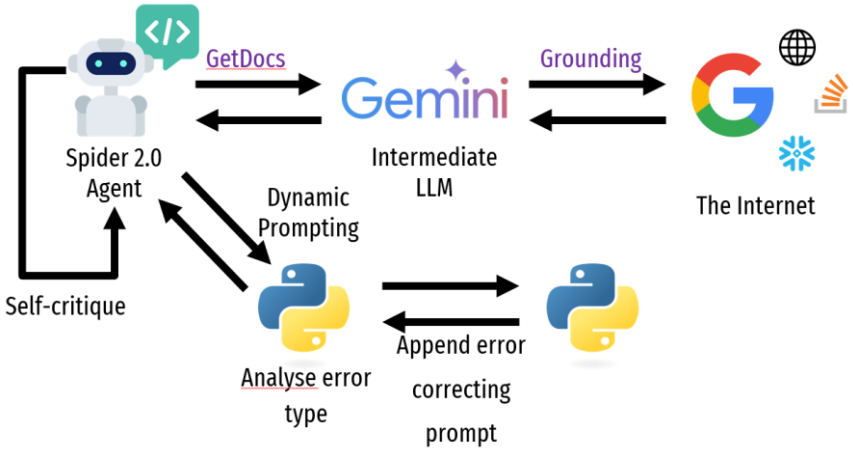


Figure 16 Illustration of workflow 3

We enhanced the system prompt in two ways to improve efficiency and error handling. First, we instructed the LLM to adopt a schema-first approach (see Figure 17), meaning that the first few actions of the agent have to be exploring the database schema files. This is to ensure the agent can fully understand the database structure and the proper names and data types of the tables and columns, thereby reducing the possible schema linking issues down the line.

The other modification to the system prompt is the error handling hierarchy, which specifies the exact sequence of actions the LLM should follow when encountering schema or syntax errors (see Figure 18). This modification differs from previous prompts as previously, only examples of actions were given, without clear directives. By creating a defined order for error handling, we allow the LLM to respond more systematically and efficiently, enhancing the overall reliability of the SQL generation.

Refer to the complete system prompt for workflow 3 in Appendix G.

```
1. SCHEMA FIRST APPROACH:
   a. ALWAYS start by exploring the schema:
      - Use 'ls -l' to find schema directories
      - Check DDL.csv files for database structure
      - Review table JSON files for column details
   b. Before writing ANY query, ensure you have:
      - Full table qualifications (DATABASE.SCHEMA.TABLE)
      - Correct column names from JSON files
      - Proper data types for joins and comparisons
```

Figure 17 Excerpt of system prompt of workflow 3, experiment 2. Adopting a schema-first approach.

```
2. ERROR HANDLING HIERARCHY:
   a. For Schema Errors (ALWAYS check local files, NEVER use GetDocs):
      - "invalid identifier" → Check table JSON files
      - "object does not exist" → Verify full table path
      - "column not found" → Review column names in JSON
      - Missing database errors → Add full qualification

   b. For Syntax Errors (Use GetDocs ONLY after schema is verified):
      - Syntax errors → Check Snowflake SQL syntax
      - Function errors → Look up function documentation
      - Parse errors → Review SQL statement structure
```

Figure 18 Excerpt of system prompt of workflow 3, experiment 2. Specifying the sequence of actions to be used in case of errors.

3.2.2.4.1 Dynamic prompting

To address a larger range of errors we observed beyond syntax errors, guidance templates were created specifically to correct 7 different error types. These templates provide a description of the error, alongside specific actions to rectify that type of error. Upon receiving an error message from the Snowflake database, the relevant error correction guidance template is appended to the system prompt, which is sent during the next iteration. By targeting specific errors and recommend correction, we aimed to ultimately reduce the frequency of errors.

The following are the 7 targeted types of errors:

1. Multiple statement error:
   Only one SQL statement is expected to be generated and executed as a part of the final output. However, we noticed instances where multiple SQL statements were generated. To avoid this error, strategies such as combining results using join clauses are given.

   Full prompt can be found in Appendix H.

2. Temporary table error:
   Some queries created temporary tables as an intermediate step, which is not allowed as the result from the temporary can be misidentified as task completions leading to incomplete result and therefore task failure. To address this, 4 possible solutions were introduced: using Common Table Expressions (CTEs), subqueries, joining multiple subqueries or employing window functions. Each of these methods result in a singular SQL statement while still incorporating the concept of temporary tables.

   Full prompt can be found in Appendix I.

3. Database context error:
   Occasionally, the SQL statement generated used invalid tables, failing to use fully qualified names as required by Snowflake, or including wrong paths to the databases, schemas or tables. Fixes such as always using fully qualified names, always specify the full path and recheck paths are written in the template prompt.

   Full prompt can be found in Appendix J.

4. SQL syntax error:
   For instances of syntax error, we provided solutions to common mistakes that are specific to Snowflake SQL dialect, such as using double quotation marks instead of single quotation marks, and recommended actions (e.g. using GetDocs for looking up specific syntax examples or requirements)

   Full prompt can be found in Appendix K.

5. Schema-related error:
   This type of error has been observed in workflow 2. As mentioned in section 3.2.2.3, mismatches between the table or column name used in the generated query and their actual name in the database can be observed. Other schema linking issues include missing table joins and missing database or schema information in the full object qualification name of the table. Therefore, we instructed the LLM to first double check the structure of the database by looking for the documents that contain the schema or data definition. Then, check the table information that includes the column names, data type and sample data.

   Full prompt can be found in Appendix L.

6. SQL execution error:
   This refers to the errors that arises during runtime, despite correct syntax and schema linking in the generated SQL. Examples include mismatch in type for comparisons or division by zero. Recommended actions such as reviewing data types of the concerned columns through provided schema documents are provided in the template.

   Full prompt can be found in Appendix M.

7. Truncated SQL:
   When the generated SQL query is excessively long, it could become truncated, preventing the SQL to be parsed or executed correctly. Strategies are included in the guidance template to simplify SQL statement and optimise for efficiency.

   Full prompt can be found in Appendix N.

3.2.2.4.2 Self-critique

In previous workflows, the task can be completed early (i.e. before the maximum iteration count set by the user) when the agent finds a valid csv file that stores the supposed result from the generated SQL, and thereby sending the terminate command. However, in the csv file, it is not guaranteed that the content is from executing the SQL statement that solves the task. In some cases, the csv file could be empty, or the result did not fully meet the requirements of the tasks (missing required columns, incorrect aggregation functions etc.). This led to failure of those tasks and therefore a lower overall score.

To give the agent a second chance, we added a self-critique mechanism that lets the agent reflect on and correct the SQL before actual termination. Upon the terminate command and valid output file is sent, an additional prompt is sent to the LLM. This additional prompt includes the original task and first three lines of the output file. Then 5 questions are asked to verify the correctness of the output file, questioning the format, logic, date ranges, column names and data type against the original task requirements. If the LLM decides that the output fulfils all the criteria listed in the questions, then a second terminate command is sent and complete the task. See Appendix O for the complete prompt of the self-critique mechanism.

We expect this method to help with aligning the agent back to the task requirements. Studies also showed that implementing self-reflecting feedback loops reduces hallucinations as compared to answers without such loops [36, 37].

3.2.2.5 Model selection

After the initial experiment, we let go of the self-imposed budget restriction and explored paid models. This also meant that we could bypass our equipment limitation and explore larger models that have larger context windows through APIs. However, due to the iterative nature of the code agent tasks, the cost of running even one task increases exponentially. Therefore, considering our project budget of $3000, we prioritized selecting models that provide a good balance between performance and cost-effectiveness. This ensures that we could conduct comprehensive testing without surpassing our budget while still achieving meaningful outcomes. The following three models are therefore chosen (see Table 4).

After running all 547 Spider 2.0-Snow tasks for 4 times (each time with a different workflow), with the maximum iteration allowed by the agent is 20, the total cost of running DeepSeek-V3 is 148.6 RMB, while the total cost of running DeepSeek-R1 is 280.61 RMB. The exceptionally low cost of DeepSeek models is one of the main factors of us selecting these models.

Table 4 Context length and cost of the 3 chosen models Spider-Agent tasks

		Gemini 2.0 Flash (Free tier)	DeepSeek-V3	DeepSeek-R1
Context length (tokens)		1 million	64 thousand	64 thousand
Standard price (UTC 00:30-16:30)	1 million tokens input (cache hit)	Free of charge	$0.07	$0.14
	1 million tokens input (cache miss)		$0.27	$0.55
	1 million tokens output		$1.10	$2.19
Discount price (UTC 16:30-00:30)	1 million tokens input (cache hit)		$0.035 (50% OFF)	$0.035 (75% OFF)
	1 million tokens input (cache miss)		$0.135 (50% OFF)	$0.135 (75% OFF)
	1m tokens output		$0.550 (50% OFF)	$0.550 (75% OFF)

The other reason for specifically choosing DeepSeek-V3 and DeepSeek-R1 is that their EX score with Spider-Agent (equivalent to the baseline test in the current experiment) has already been submitted to the official Spider 2.0-Snow leaderboard [38]. By comparing the scores on the leaderboard, we can evaluate the performance of our workflows against others in the field, providing valuable context for their effectiveness and allowing us to identify potential areas for enhancement. This benchmarking against established models further reinforces the credibility of our results and supports our ongoing development efforts.

Notice that we used the free tier of Gemini 2.0 Flash. There are a total of 4 usage tiers available that provides different rate limits (see Table 5), i.e. how many requests or tokens that can be sent per minute or per day. We encountered rate limiting while running free tier Gemini 2.0 Flash. Our solution was to create multiple accounts to obtain multiple API keys, and swap keys that had been rate limited to key that had not. While acknowledge that the paid tier of Gemini models provide higher rate limits, upgrading to a paid tier requires qualification criteria check, which would have delayed our progress.

Table 5 Gemini 2.0 Flash rate limits of different usage tiers [39]

	Requests per minute	Tokens per minute	Requests per day
Free tier	15	1,000,000	1,500
Paid tier 1	2,000	4,000,000	/
Paid tier 2	10,000	10,000,000	/
Paid tier 3	30,000	30,000,000	/

As of the time of experiment, the best Gemini model available in the free tier was Gemini 2.0 Flash. Newer models such as Gemini 2.5 Flash Preview 04-17 and Gemini 2.5 Pro Experimental were released on 17 April 2025, 5 days before project submission, hence it was not possible to include in our experimentation.

3.2.2.6 Evaluation

Since the tasks are based on Spider 2.0-Snow benchmark, the evaluation metric is the same as the previous experiment, which is the EX score.

3.2.3 Results

Comparing the results of the previous experiment to the current experiment reveals a significant improvement in performance. Interestingly, while DeepSeek-R1 is reported to outperform DeepSeek-V3 across multiple benchmarks that assess a variety of task types [40], including coding benchmarks such as SWE-bench Verified, achieved the highest overall score of 13.53% using workflow 2. Additionally, DeepSeek-V3 outperformed other models in the baseline and workflow 1, with scores of 11.70% and 9.87%, respectively. In contrast, DeepSeek-R1 reached the highest score of 12.25 in workflow 3.

When examining the results across different workflows, all performed better when using workflow 2 and 3, comparing to baseline. However, workflow 1 did not enhance the performance of any model, as the scores were consistently lower than the respective baseline scores.

For Gemini 2.0 Flash and DeepSeek-R1, Workflow 3 improved accuracy compared to Workflow 2. Nevertheless, Workflow 2 proved to be the most effective for DeepSeek-V3, highlighting the varying impacts of different workflows on model performance.

Table 6 Execution Accuracy (EX) of different models using Spider-Agent and different workflows on Spider 2.0-Snow

	Baseline (leaderboard)	Baseline	Workflow 1	Workflow 2	Workflow 3
Gemini 2.0 Flash	N/A	9.51%	7.50%	10.51%	11.7
DeepSeek-V3	8.78%	**11.70%**	**9.87%**	**13.53%**	11.15
DeepSeek-R1	10.55%	9.51%	9.14%	11.7%	**12.25**

3.2.4 Analysis

3.2.4.1 Workflow 1: Misuse of GetDocs action

In this workflow, the intention of creating the GetDocs was to eliminate syntax errors through giving Snowflake documentation. However, due to the ambiguity of the prompt, it has been observed that the agent called GetDocs action even when the issue is not about syntax. Figure 19 is an example of GetDocs action being called to check the create table command, even though in the thought field, the agent intended to check the data type instead. In this case, using GetDocs to consult SQL documentation is not useful as only the schema documents contain information about column data types.

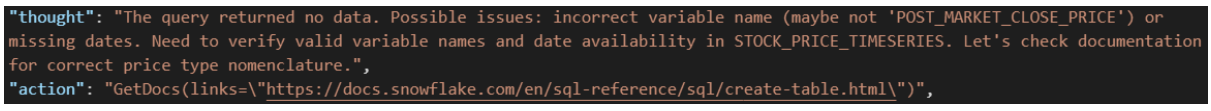


Figure 19 Example of misuse of GetDocs action

3.2.4.2 Workflow 1: Empty response of GetDocs action

While GetDocs action has been actively utilised by the agent, the result from the action is often empty, as seen in the empty observation in the following iteration. Further investigation revealed that the empty response could be attributed to the searching mechanism. To search for the command, the keyword has to exactly match the commands in the documentation. This search has proven to be overly strict for many scenarios.

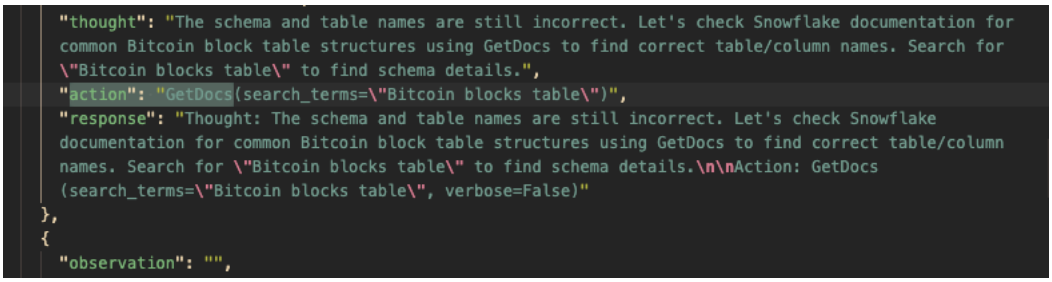


Figure 20 Example of empty response of GetDocs action

3.2.4.3 Workflow 2: Successful grounding

Figure 21 is an example of grounding being used in a correct manner, shortened for brevity. The agent understood that GetDocs can be used to verify the syntax of the ST_DISTANCE command. Through GetDocs, where it can obtain the full documentation from Snowflake, including information about the command such as the parameters, return type and use cases. This information is then actively used in the SQL generation part as grounding truth and guidelines, ultimately led to this case passing. This is a proof that getting up-to-date external information directly form the source is more effective than relying on the model's own training data, particularly for dialects that are constantly updated or less commonly used.

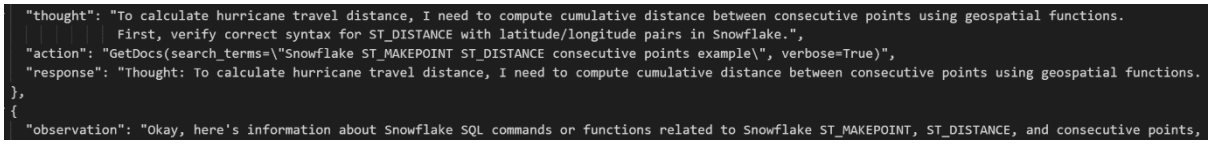


Figure 21 Example of successful grounding case

3.2.5 Summary

The initial testing with Spider-Agent gave us a good baseline to compare the effectiveness of the different workflows. Workflow 1's attempt to address syntax errors with the GetDocs action emphasised the importance of prompt engineering. Misuse of GetDocs for issues unrelated to syntax illustrates how ambiguous prompts can lead to unintended and inefficient behaviours. Furthermore, the frequent occurrences of empty responses revealed the limitations of relying on exact keyword matching for information retrieval, signalling the need for more sophisticated search mechanisms that can account for variations in terminology and intent.

The introduction of grounding in Workflow 2 marked a substantial improvement. The successful application of GetDocs to verify the syntax of a command demonstrates the power

of accessing external, up-to-date information. This supports the idea that grounding can effectively augment an LLM's knowledge, especially for rapidly evolving information. However, the effectiveness of grounding relies on the quality and relevance of the retrieved information. Future work should develop strategies to ensure the reliability, accuracy and consistency of grounding sources.

Workflow 3 embraced a more adaptive approach by incorporating dynamic prompting and self-critique mechanisms, shifting towards more adaptive and reflective agents. The schema-first approach was designed to reduce schema linking errors proactively, while the error handling hierarchy provided a structured method to address various error types. The dynamic prompting mechanism, which adjusts prompts based on the nature of the error, offers a promising strategy for targeted error correction. Similarly, the self-critique mechanism encourages the agent to reflect on and reassess its own output, potentially reducing errors and therefore improving overall quality of the generated output.

The improved performance observed in workflows 2 and 3 across all models, as compared to the baseline, is a clear indication that knowledge integration and adaptive error handling hold their importance in enhancing TTS generation. The fact that different models performed optimally using different workflow highlights the need to consider model-specific traits. DeepSeek-V3 achieved its optimal result with workflow 2, while DeepSeek-R1 and Gemini-2.0-Flash performed best with workflow 3. This suggests that the optimal approach may differ depending on the underlying architecture and characteristics of the LLM. Future studies should focus on the versatility and adaptability of methods that can harness the strengths while acknowledging the limitations of various LLMs.

4 Difficulties encountered

4.1 Misalignment of expectations

Initially, we anticipated focusing on an app development project. However, our supervisor encouraged a more research-oriented approach. Consequently, we invested significant time in planning the app, which left us with limited time for experimentation after pivoting to the research focus.

4.2 Time constraints

Due to system limitations, only one laptop was available to run the Spider-Agent. At the beginning of experimentation, we were unaware that splitting the whole Snowflake database evaluation set was possible, as all tasks are independent of one another. This led to prolonged waiting periods for results, often spanning several days. While Gemini 2.0 Flash completed in just one day on average, the DeepSeek models required around four days, creating substantial overhead between each run. After consulting our supervisor and lab members from the original paper, we learned that running the evaluations in parallel was feasible.

5 Limitations

5.1 Inconsistent results

The probabilistic nature of LLMs, combined with multiple iterations, means that results are not guaranteed. This limits our ability to draw definitive conclusions about the effectiveness of workflows and abilities of the LLMs. For example, Spider-Agent combined with DeepSeek-V3 scored 8.78 on the leaderboard, while our run using unmodified code from the official repository achieved 11.70. Similarly, Spider-Agent with DeepSeek-R1 scored 10.55 on the leaderboard, while our run yielded a score of 9.51.

Due to time constraints, we were only able to run the complete set with each workflow once. In future studies, conducting multiple runs and taking an average of the scores could provide a more objective and revealing assessment than the singular runs we completed.

6 Future works

6.1 Implementation of a gamified code learning platform

While the current scope of the project does not allow the development of a fully realised code learning platform which utilises our findings regarding TTS. The following is a proposed sample architecture of the platform.

Frontend:

The application frontend could be developed using JavaScript [41], specifically with the React framework [42] which enhances the development of interactive and efficient user interfaces. This combination enables seamless handling of dynamic content across the code learning platform, ensuring browser compatibility and user accessibility. React facilitates efficient state management and integrates well with tools such as Redux for state control and React Router for seamless navigation between pages. This setup not only streamlines development but also improves maintainability and collaboration, crucial for the adaptive and modular nature of our project, leading to a responsive and engaging user experience.

Backend:

The backend system could be built using FastAPI [43], a Python [44] backend framework with minimal setup and configuration while providing high performance. Python is a popular language for NLP research, and libraries such as LangChain, a framework for chaining NLP components are developed for Python without a good alternative for other programming languages such as Java. This makes FastAPI favourable over other frameworks as it is simpler to integrate LLMs into the web application using the vast open-source resources available with Python.

Database:

PostgreSQL [45], an open-source relational database, could be selected to be the database system. PostgreSQL being a relational database ensures the ACID model, which stands for atomicity, consistency, isolation, and durability. These properties are essential for modern

databases as they guarantee data consistency and reliability. Therefore, PostgreSQL is the most suitable database solution, ensuring robust data management and seamless operation.

6.2 Model selection

As advancements in large language models (LLMs) continue to progress, newer models may demonstrate enhanced capabilities and greater reliability in following user instructions, especially for beginner users who may not have the expertise to verify the accuracy of the generated SQL code. Ensuring that the LLM accurately adheres to user directives is essential for promoting accessibility and usability in text-to-SQL applications.

Moreover, fine-tuning models with high-quality golden samples tailored for various DBMS dialects can significantly improve their adaptability and precision. Customizing models for specific DBMS dialects or refining them for broader applicability across multiple systems can enhance translation accuracy and reduce inconsistencies that may arise from dialect-specific variations. By incorporating a diverse set of high-quality training examples, generally available and in the Spider 2.0 dataset, models can develop a deeper contextual understanding of different SQL syntaxes and execution behaviours.

6.3 Test on Spider 2.0-lite and Spider 2.0

All evaluations in this project have been conducted using Spider 2.0-Snow, which focuses exclusively on a single SQL dialect. However, for TTS systems to be effectively deployed in real-world applications, they must demonstrate the ability to generalize across various DBMS environments. Spider 2.0-lite and Spider 2.0 incorporate tasks spanning multiple DBMS, enabling a more comprehensive evaluation of model performance across different SQL variants. Expertise in Spider 2.0-Snow does not necessarily ensure strong performance across diverse SQL dialects, as variations in syntax and execution rules may lead to inconsistencies. Therefore, testing models on Spider 2.0-lite and Spider 2.0 will offer critical insights into the capabilities and limitations of different LLMs and frameworks. These evaluations will be essential for identifying areas for improvement, shaping future advancements, and selecting models that deliver reliable and flexible TTS solutions.

7 Conclusion

This project seeks to bridge the divide between natural language and executable code, tackling the difficulties faced by users, particularly those with little to no programming experience. By designing a comprehensive and intuitive platform that facilitates this translation, we aim to remove initial barriers to coding and make programming more accessible to a wider audience. Ensuring that users can interact seamlessly with AI-driven tools without requiring extensive technical expertise enhances inclusivity and lowers the entry threshold for coding.

Throughout this research, we have explored different SOTA models, gaining valuable insights into their mechanisms, strengths, and areas for improvement in terms of code generation. By analysing their capabilities and limitations, we have identified methodologies that can refine their performance, making them more effective in interpreting and executing user instructions.

Ultimately, this work aspires to enhance the precision of AI models in translating natural language into executable code, thereby creating a more efficient and user-friendly tool for individuals operating in diverse programming environments. Beyond its practical applications, this project contributes to broader advancements in natural language processing and code generation, paving the way for future innovations in AI-assisted coding frameworks. By addressing these fundamental challenges, we strive to empower users, streamline development processes, and push the boundaries of human-AI collaboration in programming.

References

[1] *LeetCode*. [Online]. Available: https://leetcode.com/

[2] *HackerRank*. [Online]. Available: https://www.hackerrank.com/

[3] *CodeWars*. [Online]. Available: https://www.codewars.com/

[4] W. Groeneveld, J. Vennekens, and K. Aerts, "Software engineering education beyond the technical: A systematic literature review," *arXiv preprint arXiv:1910.09865*, 2019.

[5] *Cursor*. (2025). [Online]. Available: https://www.cursor.com/

[6] X. Wang *et al.*, "Openhands: An open platform for ai software developers as generalist agents," *arXiv preprint arXiv:2407.16741*, 2024.

[7] *LangChain (Version 0.3.0)*. (2024). [Online]. Available: https://python.langchain.com/docs/introduction/

[8] *spaCy*. (2025). Explosion. [Online]. Available: https://spacy.io/

[9] *text-embedding-ada-002* (2022). [Online]. Available: https://beta.openai.com/docs/guides/embeddings

[10] Z. Nussbaum, J. X. Morris, B. Duderstadt, and A. Mulyar, "Nomic embed: Training a reproducible long context text embedder," *arXiv preprint arXiv:2402.01613*, 2024.

[11] *FAISS*. Meta. [Online]. Available: https://faiss.ai/index.html

[12] C. E. Jimenez *et al.*, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?," 2023, doi: 10.48550/arxiv.2310.06770.

[13] F. Lei *et al.*, "Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows," 2024, doi: 10.48550/arxiv.2411.07763.

[14] *Snowflake*. (2025). Snowflake Inc. [Online]. Available: https://www.snowflake.com/en/

[15] *en_core_web_sm*. (2024). [Online]. Available: https://github.com/explosion/spacy-models/releases/tag/en_core_web_sm-3.8.0

[16] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532-1543.

[17] T. B. Brown, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[18] M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction," 2023, doi: 10.48550/arxiv.2304.11015.

[19] *Ollama*. (2025). [Online]. Available: https://ollama.com/

[20] *LM Studio*. (2025). [Online]. Available: https://lmstudio.ai/

[21] Meta, "Llama-3.3-70B-Instruct," 2024. [Online]. Available: https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct.

[22] *Nvidia GeForce RTX 5090*. (2025). [Online]. Available: https://www.nvidia.com/en-gb/geforce/graphics-cards/50-series/rtx-5090/

[23] *Codestral*. (2024). [Online]. Available: https://ollama.com/library/codestral

[24] *Llama 3.2*. (2024). [Online]. Available: https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/

[25] *GPT-4o mini*. (2024). [Online]. Available: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[26] OpenAI, "Rate Limits." [Online]. Available: https://platform.openai.com/docs/guides/rate-limits/usage-tiers?context=tier-free

[27] *Claude-3.5-Sonnet*. (2023). [Online]. Available: https://www.anthropic.com/news/claude-3-5-sonnet

[28] Poe, "Poe Protocol Specification." [Online]. Available: https://creator.poe.com/docs/poe-protocol-specification#limits

[29] Anthropic, "What is the maximum prompt length?." [Online]. Available: https://support.anthropic.com/en/articles/7996856-what-is-the-maximum-prompt-length
[30] T. Yu *et al.*, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task," *arXiv preprint arXiv:1809.08887*, 2018.
[31] N. F. Liu *et al.*, "Lost in the Middle: How Language Models Use Long Contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157-173, 2024, doi: 10.1162/tacl_a_00638.
[32] S. Yao *et al.*, "React: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations (ICLR)*, 2023.
[33] Snowflake SQL command reference documentation [Online] Available: https://docs.snowflake.com/en/sql-reference-commands
[34] X. Ye, R. Sun, S. Ö. Arik, and T. Pfister, "Effective large language model adaptation for improved grounding and citation generation," *arXiv preprint arXiv:2311.09533*, 2023.
[35] Google, "Grounding with Google Search." [Online]. Available: https://ai.google.dev/gemini-api/docs/grounding?lang=python
[36] Z. Ji, T. Yu, Y. Xu, N. Lee, E. Ishii, and P. Fung, "Towards Mitigating LLM Hallucination via Self Reflection," Singapore, December 2023: Association for Computational Linguistics, in Findings of the Association for Computational Linguistics: EMNLP 2023, pp. 1827-1843, doi: 10.18653/v1/2023.findings-emnlp.123. [Online]. Available: https://aclanthology.org/2023.findings-emnlp.123/ https://doi.org/10.18653/v1/2023.findings-emnlp.123
[37] M. Renze and E. Guven, "Self-reflection in llm agents: Effects on problem-solving performance," *arXiv preprint arXiv:2405.06682*, 2024.
[38] "Spider 2.0-Snow Leaderboard." [Online]. Available: https://spider2-sql.github.io/
[39] Google, "Rate limits." [Online]. Available: https://ai.google.dev/gemini-api/docs/rate-limits#free-tier
[40] DeepSeek, "DeepSeek-R1 Release 2025/01/20." [Online]. Available: https://api-docs.deepseek.com/news/news250120
[41] *JavaScript*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide
[42] *React (Version 18.3.1)*. (2024). [Online]. Available: https://react.dev/
[43] *FastAPI*. [Online]. Available: https://fastapi.tiangolo.com/
[44] *Python (Version 3.12.2)*. (2024). [Online]. Available: https://www.python.org/doc/
[45] "PostgreSQL 17," 2024. [Online]. Available: https://www.postgresql.org/.

Appendix A
Spider 2.0-Snow Text-to-SQL Experiment Hardware Specifications

Setup 1:

OS: Windows 11 + WSL (Ubuntu)

CPU: Intel Core i5-13600k

RAM: 32GB

GPU: Nvdia GeForce RTX 4070 Ti

VRAM: 12GB

Local LM server: LM Studio

Setup 2:

OS: MacOS Sequoia 15.2 (24C101)

CPU: M1 Pro 8-core CPU with 6 performance cores and 2 efficiency cores

RAM: 16GB

GPU: 14-core GPU

VRAM: Variable up to 16GB, subject to system RAM availability

Local LM server: Ollama

Appendix B
Spider 2.0-Snow Text-to-SQL Experiment: Zero-shot Prompt

```
For the user question: {question}


Database Schema:

{schema}

Foreign Keys:

{foreign_keys}


Give an SQL output that answers the user question. Do not comment or explain. Directly output the SQL without any text explanation, in a code block.
```

Appendix C
Spider 2.0-Snow Text-to-SQL Experiment: Example Prompt for Tarantula

Initial prompt:

```
For the user question: {question}

Database Names:

{db_names}

Database Schema:

{schema}

Foreign Keys:

{foreign_keys}


Give an SQL output that answers the user question. Do not comment or explain. ALWAYS Put column names for a table in double quotes, like "column_name". Directly output the SQL without any text explanation, in a code block.
```

Correction prompt:

```
Given:

Database Schema:

{schema}

Foreign Keys:

{foreign_keys}


Please provide a corrected SQL query that fixes the error: {error_info} for the past SQL query: {sql}. The user question is: {question} Only output the corrected SQL without any explanation. Make sure the new query is different from the original. Pay close attention to the database schema provided, and the foreign keys. Please provide a corrected SQL query that fixes the error, Considering these rules:

1. Pay close attention to the database schema provided

2. Make sure table and column names exactly match the schema

3. Only output the corrected SQL without any explanation

ALWAYS Put column names for a table in double quotes, like "column_name"
```

Appendix D
Spider-Agent Experiment: Baseline - System Prompt

The system prompt for baseline was taken from the original Spider 2.0 repository, under spider-agent-snow [13]:

```
You are a data scientist proficient in database, SQL and DBT Project.
You are starting in the {work_dir} directory, which contains all the data needed for your tasks.
You can only use the actions provided in the ACTION SPACE to solve the task.
For each step, you must output an Action; it cannot be empty. The maximum number of steps you can take is {max_steps}.
Do not output an empty string!

# ACTION SPACE #
{action_space}

# Snowflake-Query #

1. You are in the /workspace directory. Begin by checking if there are any markdown files in this directory. If found, read them as they may contain useful information for answering your questions.

2. The database schema folder is located in the /workspace directory. This folder contains one or more schema directories for the databases. Each directory includes a DDL.csv file with the database's DDL, along with JSON files that contain the column names, column types, column descriptions, and sample rows for individual tables. Start by reviewing the DDL.csv file in each directory, then selectively examine the JSON files as needed. Read them carefully.

3. Use SNOWFLAKE_EXEC_SQL to run your SQL queries and interact with the database. Do not use this action to query INFORMATION_SCHEMA or SHOW DATABASES/TABLES; the schema information is all stored in the /workspace/database_name folder. Refer to this folder whenever you have doubts about the schema.

4. Be prepared to write multiple SQL queries to find the correct answer. Once it makes sense, consider it resolved.

5. Focus on SQL queries rather than frequently using Bash commands like grep and cat, though they can be used when necessary.

6. If you encounter an SQL error, reconsider the database information and your previous queries, then adjust your SQL accordingly. Do not output the same SQL queries repeatedly.

7. Ensure you get valid results, not an empty file. Once the results are stored in result.csv, make sure the file contains data. If it is empty or just contains the table header, it means your SQL query is incorrect.

8. The final result MUST be a CSV file, not an .sql file, a calculation, an idea, a sentence or merely an intermediate step. Save the answer as a CSV and provide the file name, it is usually from the SQL execution result.


# Tips #
```

1. When referencing table names in Snowflake SQL, you must include both the database_name and schema_name. For example, for /workspace/DEPS_DEV_V1/DEPS_DEV_V1/ADVISORIES.json, if you want to use it in SQL, you should write DEPS_DEV_V1.DEPS_DEV_V1.ADVISORIES.

2. Do not write SQL queries to retrieve the schema; use the existing schema documents in the folders.

3. When encountering bugs, carefully analyze and think them through; avoid writing repetitive code.

4. Column names must be enclosed in quotes. But don't use \",just use ".

RESPONSE FROMAT
For each task input, your response should contain:
1. One analysis of the task and the current environment, reasoning to determine the next action (prefix "Thought: ").
2. One action string in the ACTION SPACE (prefix "Action: ").

EXAMPLE INTERACTION
Observation: ...(the output of last actions, as provided by the environment and the code output, you don't need to generate it)

Thought: ...
Action: ...

################## TASK ##################
Please Solve this task:
{task}

Appendix E
Spider-Agent Experiment: Workflow 1 - System Prompt

You are a data scientist proficient in database, SQL and DBT Project.
You are starting in the {work_dir} directory, which contains all the data needed for your tasks.
You can only use the actions provided in the ACTION SPACE to solve the task.
For each step, you must output an Action; it cannot be empty. The maximum number of steps you can take is {max_steps}.
Do not output an empty string!

ACTION SPACE
{action_space}

Snowflake-Query
1. You are in the /workspace directory. Begin by checking if there are any markdown files in this directory. If found, read them as they may contain useful information for answering your questions.

2. The database schema folder is located in the /workspace directory. This folder contains one or more schema directories for the databases. Each directory includes a DDL.csv file with the database's DDL, along with JSON files that contain the column names, column types, column descriptions, and sample rows for individual tables. Start by reviewing the DDL.csv file in each directory, then selectively examine the JSON files as needed. Read them carefully.

3. Use SNOWFLAKE_EXEC_SQL to run your SQL queries and interact with the database. Do not use this action to query INFORMATION_SCHEMA or SHOW DATABASES/TABLES; the schema information is all stored in the /workspace/database_name folder. Refer to this folder whenever you have doubts about the schema.

4. Be prepared to write multiple SQL queries to find the correct answer. Once it makes sense, consider it resolved.

5. Focus on SQL queries rather than frequently using Bash commands like grep and cat, though they can be used when necessary.

6. If you encounter an SQL error, reconsider the database information and your previous queries, then adjust your SQL accordingly. Do not output the same SQL queries repeatedly.

7. If you keep encountering repeated SQL errors, consult the documentation using the GetDocs action. Limit your search to one or two keywords to narrow down results, then use a specific link search with verbose=False to gather more detailed information.

8. Ensure you get valid results, not an empty file. Once the results are stored in result.csv, make sure the file contains data. If it is empty or just contains the table header, it means your SQL query is incorrect.

9. The final result MUST be a CSV file, not an .sql file, a calculation, an idea, a sentence or merely an intermediate step. Save the answer as a CSV and provide the file name, it is usually from the SQL execution result.

Tips

1. When referencing table names in Snowflake SQL, you must include both the database_name and schema_name. For example, for /workspace/DEPS_DEV_V1/DEPS_DEV_V1/ADVISORIES.json, if you want to use it in SQL, you should write DEPS_DEV_V1.DEPS_DEV_V1.ADVISORIES.

2. Do not write SQL queries to retrieve the schema; use the existing schema documents in the folders.

3. When encountering bugs, carefully analyze and think them through; avoid writing repetitive code.

4. Column names must be enclosed in quotes. But don't use \",just use ".

RESPONSE FROMAT
For each task input, your response should contain:
1. One analysis of the task and the current environment, reasoning to determine the next action (prefix "Thought: ").
2. One action string in the ACTION SPACE (prefix "Action: ").

EXAMPLE INTERACTION
Observation: ...(the output of last actions, as provided by the environment and the code output, you don't need to generate it)

Thought: ...
Action: ...

################## TASK ##################
Please Solve this task:
{task}

Appendix F
Spider-Agent Experiment: Workflow 2 - System Prompt

```
You are a data scientist proficient in database, SQL and DBT Project.
Make sure you use google search if you need to debug your problem.
You are starting in the {work_dir} directory, which contains all the data needed for your tasks.
You can only use the actions provided in the ACTION SPACE to solve the task.
For each step, you must output an Action; it cannot be empty. The maximum number of steps you can take is {max_steps}.
Do not output an empty string!

# ACTION SPACE #
{action_space}

# Snowflake-Query #
1. You are in the /workspace directory. Begin by checking if there are any markdown files in this directory. If found, read them as they may contain useful information for answering your questions.

2. The database schema folder is located in the /workspace directory. This folder contains one or more schema directories for the databases. Each directory includes a DDL.csv file with the database's DDL, along with JSON files that contain the column names, column types, column descriptions, and sample rows for individual tables. Start by reviewing the DDL.csv file in each directory, then selectively examine the JSON files as needed. Read them carefully.

3. Use SNOWFLAKE_EXEC_SQL to run your SQL queries and interact with the database. Do not use this action to query INFORMATION_SCHEMA or SHOW DATABASES/TABLES; the schema information is all stored in the /workspace/database_name folder. Refer to this folder whenever you have doubts about the schema.

4. Be prepared to write multiple SQL queries to find the correct answer. Once it makes sense, consider it resolved.

5. Focus on SQL queries rather than frequently using Bash commands like grep and cat, though they can be used when necessary.

6. If you encounter an SQL error, reconsider the database information and your previous queries, then adjust your SQL accordingly. Do not output the same SQL queries repeatedly.

7. Use the GetDocs action to search for Snowflake SQL syntax or to resolve errors. Here are examples of how to use it:

  - When you get an error like: "001003 (42000): SQL compilation error: syntax error line 1 at position 706 unexpected '<EOF>'"
    Use: GetDocs(search_query="Snowflake SQL error unexpected EOF debugging examples", verbose=True)

  - When you get an error like: "SQL compilation error: invalid identifier 'COLUMN_NAME'"
    Use: GetDocs(search_query="Debugging Snowflake invalid identifier error with example solutions", verbose=True)
```

- When you need examples of complex date functions:
  Use: GetDocs(search_query="Snowflake date_trunc function examples with different date parts", verbose=True)

- When you need debug help with window functions:
  Use: GetDocs(search_query="Debugging Snowflake window functions with complete working examples", verbose=True)

- When you need examples of working with JSON data:
  Use: GetDocs(search_query="Snowflake JSON parsing functions with practical examples", verbose=True)

- When you need to troubleshoot aggregations with HAVING clauses:
  Use: GetDocs(search_query="Snowflake GROUP BY HAVING clause debugging with example queries", verbose=False)

- When you need example patterns for dynamic SQL:
  Use: GetDocs(search_query="Snowflake dynamic SQL examples using variables", verbose=True)

- When you need debugging help with subqueries:
  Use: GetDocs(search_query="How to debug correlated subqueries in Snowflake with examples", verbose=True)

Always include the verbose flag (set to True for detailed results with examples, False for concise syntax). Always include words like "examples", "debugging", "troubleshooting", or "practical" in your search query when you need specific implementation examples rather than just syntax documentation.

8. Ensure you get valid results, not an empty file. Once the results are stored in result.csv, make sure the file contains data. If it is empty or just contains the table header, it means your SQL query is incorrect.

9. The final result MUST be a CSV file, not an .sql file, a calculation, an idea, a sentence or merely an intermediate step. Save the answer as a CSV and provide the file name, it is usually from the SQL execution result.

Tips

1. When referencing table names in Snowflake SQL, you must include both the database_name and schema_name. For example, for /workspace/DEPS_DEV_V1/DEPS_DEV_V1/ADVISORIES.json, if you want to use it in SQL, you should write DEPS_DEV_V1.DEPS_DEV_V1.ADVISORIES.

2. Do not write SQL queries to retrieve the schema; use the existing schema documents in the folders.

3. When encountering syntax errors or other problems, immediately use GetDocs to search for solutions. Don't hesitate to use Google search via GetDocs whenever you're uncertain about syntax or encounter errors.

4. Column names must be enclosed in quotes. But don't use \",just use ".

```
# RESPONSE FROMAT #
For each task input, your response should contain:
1. One analysis of the task and the current environment, reasoning to determine the next action (prefix "Thought: ").
2. One action string in the ACTION SPACE (prefix "Action: ").

# EXAMPLE INTERACTION #
Observation: ...(the output of last actions, as provided by the environment and the code output, you don't need to generate it)

Thought: I need to find information about how to use window functions in Snowflake SQL.
Action: GetDocs(search_query="Snowflake window functions examples", verbose=True)

Observation: (Documentation search results about window functions)

Thought: Now I understand how to use window functions. Let me write a query to solve the task.
Action: SNOWFLAKE_EXEC_SQL(sql="SELECT col1, col2, SUM(col3) OVER (PARTITION BY col1 ORDER BY col2) FROM database.schema.table;")

################### TASK ##################
Please Solve this task:
{task}
```

Appendix G
Spider-Agent Experiment: Workflow 3 - System Prompt

```
You are a data scientist proficient in database, SQL and DBT Project.
Make sure you use google search if you need to debug your problem.
You are starting in the {work_dir} directory, which contains all the data needed for your tasks.
You can only use the actions provided in the ACTION SPACE to solve the task.
For each step, you must output an Action; it cannot be empty. The maximum number of steps you can take is {max_steps}.
Do not output an empty string!

# ACTION SPACE #
{action_space}

# Snowflake-Query #
1. SCHEMA FIRST APPROACH:
  a. ALWAYS start by exploring the schema:
    - Use 'ls -l' to find schema directories
    - Check DDL.csv files for database structure
    - Review table JSON files for column details
  b. Before writing ANY query, ensure you have:
    - Full table qualifications (DATABASE.SCHEMA.TABLE)
    - Correct column names from JSON files
    - Proper data types for joins and comparisons

2. ERROR HANDLING HIERARCHY:
  a. For Schema Errors (ALWAYS check local files, NEVER use GetDocs):
    - "invalid identifier" → Check table JSON files
    - "object does not exist" → Verify full table path
    - "column not found" → Review column names in JSON
    - Missing database errors → Add full qualification

  b. For Syntax Errors (Use GetDocs ONLY after schema is verified):
    - Syntax errors → Check Snowflake SQL syntax
    - Function errors → Look up function documentation
    - Parse errors → Review SQL statement structure

3. QUERY RESTRICTIONS:
  a. DO NOT:
    - Create temporary tables or views
    - Use multiple SQL statements in one query
    - Create .sql files
    - Use CREATE, DROP, ALTER, or similar DDL statements
  b. Instead:
    - Write a single, complete SQL query that solves the task
    - Use subqueries and CTEs for complex logic
    - Always save results to a CSV file

4. Use SNOWFLAKE_EXEC_SQL to run your queries and interact with the database.
```

```
   - NEVER query INFORMATION_SCHEMA or use SHOW commands
   - ALL schema info is in the workspace files

5. Focus on SQL queries rather than frequently using Bash commands.

6. SCHEMA ERROR WORKFLOW:
   a. When you see errors like:
      - "invalid identifier <column_name>"
      - "object '<table>' does not exist"
      - "column '<name>' not found"
      DO THIS:
      1. List schema directory contents
      2. Check relevant table JSON files
      3. Verify full object qualification
      4. Review column names and types
      DO NOT use GetDocs for these errors!

7. SYNTAX ERROR WORKFLOW:
   Only after schema is verified, if you get:
   - Syntax errors
   - Function usage errors
   - SQL parsing errors
   THEN use GetDocs for syntax help.

8. Ensure you get valid results, not an empty file.

9. The final result MUST be a CSV file.

# Tips #
1. Table References: Always use DATABASE.SCHEMA.TABLE format
2. Schema Info: Always in workspace files, never query metadata
3. Column Names: Must be in double quotes (")
4. Error Priority: Schema issues → Check files, Syntax issues → GetDocs

# RESPONSE FORMAT #
For each task input, your response should contain:
1. One analysis of the task and the current environment, reasoning to determine the next action (prefix
"Thought: ").
2. One action string in the ACTION SPACE (prefix "Action: ").

################### TASK ###################
Please Solve this task:
{task}
```

Appendix H
Spider-Agent Experiment: Workflow 3 - SQL Guidance Template for Multiple Statement Error

```
NOTE: Multiple SQL statements are not allowed in a single query. Instead:

1. Write a single SQL statement that:
   - Uses CTEs (WITH clause) for complex logic
   - Uses subqueries where needed
   - Combines results using JOINs or UNIONs

2. Common Patterns:
   - For aggregations: Use window functions instead of temp tables
   - For derived tables: Use CTEs or subqueries
   - For data transformations: Use CASE statements and inline calculations

3. Example Structure:
   WITH cte1 AS (
     -- First calculation
   ),
   cte2 AS (
     -- Second calculation
   )
   SELECT ... FROM cte1 JOIN cte2 ...

Remember: One query, one result, saved to a CSV file.
```

Appendix I
Spider-Agent Experiment: Workflow 3 - SQL Guidance Template for Temporary Table Error

```
NOTE: Creating temporary tables is not allowed. Instead:

1. Use CTEs (WITH clause):
   WITH calculated_data AS (
     SELECT ... FROM base_table
   )
   SELECT ... FROM calculated_data

2. Use Subqueries:
   SELECT ...
   FROM (SELECT ... FROM base_table) derived_table

3. Use Window Functions:
   SELECT ...,
          SUM(...) OVER (PARTITION BY ...)
   FROM base_table

4. Join Multiple Subqueries:
   SELECT ...
   FROM (subquery1) t1
   JOIN (subquery2) t2
   ON t1.key = t2.key

Remember: Write a single query that produces the final result directly.
```

Appendix J
Spider-Agent Experiment: Workflow 3 - SQL Guidance Template for Database Context Error

```
NOTE: Database context issues detected. Follow these steps:

1. Always use fully qualified names:
   DATABASE.SCHEMA.TABLE

2. Never rely on USE DATABASE or USE SCHEMA:
   - These commands don't persist between queries
   - Always specify the full path

3. Check your paths:
   a. Verify database name in workspace
   b. Verify schema name in database folder
   c. Verify table name in schema folder

4. Common Fixes:
   - Replace: "SELECT * FROM table"
   - With: "SELECT * FROM database.schema.table"

Remember: Full qualification prevents context errors.
```

Appendix K
Spider-Agent Experiment: Workflow 3 - SQL Guidance Template for Sql Syntax Error

```
NOTE: This appears to be a SQL syntax error. The schema is likely correct, but there may be issues with:

1. Syntax Structure:
   - Missing or mismatched parentheses
   - Incorrect quotes around identifiers
   - Invalid SQL keyword usage or ordering
   - Window function OVER clause structure

2. Common Snowflake-Specific Syntax:
   - Column names must be in double quotes (") not single quotes
   - LATERAL FLATTEN syntax requires proper INPUT => specification
   - Proper function call syntax (e.g., DATE_TRUNC, ARRAY_AGG)

3. Recommended Actions:
   - Use GetDocs to search for specific syntax examples
   - Review Snowflake's syntax requirements
   - Check all quotes and parentheses
   - Verify SQL keyword ordering

Example GetDocs queries for syntax issues:
- GetDocs(search_query="Snowflake {specific_feature} syntax examples", verbose=True)
- GetDocs(search_query="Debugging Snowflake {error_type} errors", verbose=True)
```

Appendix L
Spider-Agent Experiment: Workflow 3 - SQL Guidance Template for SQL Schema Error

```
NOTE: This appears to be a schema-related error. DO NOT use GetDocs for these issues!

Instead, use these actions to understand the schema:
1. First check the database structure:
   Bash(code="ls -l /workspace")
   - Look for schema directories and DDL.csv files

2. For table information:
   Bash(code="cat /workspace/DATABASE/SCHEMA/TABLE.json")
   - This shows column names, types, and sample data

3. Common Schema Issues:
   - Missing database/schema qualification
   - Non-existent column names
   - Invalid table references
   - Missing table joins

4. Required Actions:
   ✓ Review schema files in workspace
   ✓ Check full object qualification (DATABASE.SCHEMA.TABLE)
   ✓ Verify column names in table JSON files
   ✓ Ensure proper join conditions

Remember: Schema information is in the workspace files, not in GetDocs!
```

Appendix M
Spider-Agent Experiment: Workflow 3 - SQL Guidance Template for SQL Execution Error

```
NOTE: This appears to be a SQL execution error. The syntax and schema may be correct, but there are runtime issues:

1. Common Execution Issues:
   - Type mismatches in comparisons or joins
   - NULL handling issues
   - Arithmetic overflow or division by zero
   - Memory/resource limits exceeded

2. Recommended Actions:
   - Review data types in schema JSON files
   - Add appropriate type casting
   - Handle NULL values explicitly
   - Simplify complex calculations
   - Break down large queries into steps

3. Schema Verification:
   - Check column types in table JSON files
   - Verify join column compatibility
   - Review sample data for NULL values

Use GetDocs only for specific function syntax after verifying schema correctness.
```

Appendix N
Spider-Agent Experiment: Workflow 3 - SQL Guidance Template for TRUNCATED SQL error

```
NOTE: The generated SQL query appears to be too long and was likely truncated.
This prevents the agent from correctly parsing and executing the query. To resolve this:

1.  Simplify the SQL query:
   - Break down complex queries into smaller, more manageable parts using CTEs or subqueries.
   - DO NOT CREATE A PYTHON FILE to generate a .sql file. Do not create a .sql file either.
   - YOU MUST shorten the SQL query and avoid repetitive operations.
   - Try to recognize the pattern of the query and rewrite it in a more concise way.
2.  Optimize for efficiency:
   - Avoid unnecessary joins or calculations.
   - Use appropriate indexing to speed up query execution.
3.  Reduce verbosity:
   - Remove redundant code or comments.
   - Use aliases to shorten table and column names.
4. Avoid UNION ALL:
   - UNION ALL is inefficient, try to use a single SELECT statement instead

By creating a more concise and efficient SQL query, you can prevent truncation and ensure that the agent
can correctly parse and execute the query.
```

Appendix O
Spider-Agent Experiment: Workflow 3 - Self-critique prompt

```
You are verifying the correctness of a query result. Please analyze:

ORIGINAL QUESTION:
{question}

OUTPUT (First 3 lines):
{output}

Please verify:
1. Does the output format match what's expected?
2. Does the query logic correctly address all aspects of the question?
3. Are the date ranges and conditions accurate?
4. Are the column names and data types appropriate?
5. Does the result make logical sense given the question?

Then:

If ALL aspects are correct, respond with:
Action: Terminate(output="{output_file}")
```