REFORCE: A TEXT-TO-SQL AGENT WITH SELF-REFINEMENT, FORMAT RESTRICTION, AND COLUMN EXPLORATION

Minghang Deng[1] **Ashwin Ramachandran**[1] **Canwen Xu**[2]
Lanxiang Hu[1,2] **Zhewei Yao**[2] **Anupam Datta**[2] **Hao Zhang**[1,2*]
[1]University of California, San Diego [2]Snowflake AI Research

ABSTRACT

Text-to-SQL systems have unlocked easier access to critical data insights by enabling natural language queries over structured databases. However, deploying such systems in enterprise environments remains challenging due to factors such as large, complex schemas ($>$ 3000 columns), diverse SQL dialects (e.g., BigQuery, Snowflake) and sophisticated query requirements (e.g., transformation, analytics). Current state-of-the-art performance on the Spider 2.0 dataset — a benchmark built to mimic such complex environments — remains limited at 20%. Key limitations include inadequate instruction-following, poor long-context comprehension, weak self-refinement, and insufficient dialect-specific knowledge. To address these gaps, we propose **ReFoRCE** (Self-**Re**finement Agent with **Fo**rmat **R**estriction and **C**olumn **E**xploration) which introduces (1) table compression to mitigate long-context limitations (2) format restriction to ensure accurate answer format, and (3) iterative column exploration for enhanced schema understanding. Additionally, it employs self-refinement pipeline consisting of (1) parallelized workflows with voting mechanisms and (2) a Common Table Expression (CTE) based refinement approach to handle unresolved cases. ReFoRCE achieves state-of-the-art results scoring 31.26 on the Spider 2.0-Snow and scoring 30.35 on the Spider 2.0-Lite tasks.

1 INTRODUCTION

Text-to-SQL converts natural language queries into SQL queries, serving as a key technology for lowering the barrier to accessing relational databases (Zelle & Mooney, 1996; Zettlemoyer & Collins, 2012; Zhong et al., 2017; Yu et al., 2018; Wang et al., 2019; Gao et al., 2023a; Lei et al., 2024). This technique enables natural language interfaces for databases, supporting critical applications such as business intelligence and automated processes. Text-to-SQL reduces repetitive human labor and alleviates the burden on data analysts and programmers alike.

Previous Text-to-SQL research primarily focused on model training and fine-tuning (Zhong et al., 2017; Wang et al., 2019; Scholak et al., 2021) on simpler datasets like Spider 1.0 (Yu et al., 2018). The rise of large language models (LLMs) has shifted this paradigm from "pre-train and fine-tune" to prompting, thanks to the strong code generation capabilities of LLMs (Anthropic, 2023; Roziere et al., 2023; Achiam et al., 2023). Consequently, numerous Text-to-SQL methods now rely on prompting (Zhang et al., 2023; Gao et al., 2023a; Pourreza & Rafiei, 2024; Talaei et al., 2024) with powerful LLM API services.

These methods achieve impressive performance on classic benchmarks, e.g., exceeding 90% on Spider 1.0 (Yu et al., 2018) and 70% on BIRD (Li et al., 2024b). However, these datasets are often built on non-industrial databases with few tables and columns, simplistic SQL queries, and straightforward questions. These limitations fail to reflect the complexity of real-world tasks. As a result, existing methods struggle with the newly proposed Spider 2.0 dataset (Lei et al., 2024), which mirrors real-world challenges by requiring multiple SQL dialects, varying syntax and functions, nested columns, external knowledge, and the ability to handle ambiguous requests and column names.

*Correspondence to haozhang@ucsd.edu

This complexity of realistic Text-to-SQL problems calls for agentic methods that enable LLMs to dynamically interact with their environment, i.e., the databases. These methods utilize tools, execute commands, observe feedback, and plan actions, surpassing simple prompting to tackle more complex tasks such as planning (Wang et al., 2023; Shinn et al., 2024), reasoning (Wei et al., 2022; Besta et al., 2024; Shao et al., 2024), and advanced code generation (Chen et al., 2023; 2024; Yang et al., 2024). As a result, Lei et al. (2024) introduced Spider-Agent, a framework based on the ReAct paradigm (Yao et al., 2023), which combines reasoning and acting components to navigate and overcome the challenges posed by the Spider 2.0 dataset.

However, code agents often face challenges in maintaining control, particularly in long-context scenarios where they may fail to follow instructions or overlook critical task details. Providing an LLM with complete database information without specifying value types or SQL dialects often leads to repeated iterations for fixing syntax errors, correcting data types, or selecting the correct functions, leaving limited room for meaningful reasoning. Furthermore, existing Text-to-SQL methods struggle to handle challenges such as multiple dialects, nested columns, and complex data types in the Spider 2.0 dataset.

To address these issues, we propose ReFoRCE (Self-**Re**finement Agent with **Fo**rmat **R**estriction and **C**olumn **E**xploration), which simplifies the process by breaking it into manageable subtasks for better control. As shown in Figure 1, we use table information compression to mitigate long-context issues in large databases, a common limitation of current Text-to-SQL methods. We introduce answer format restriction to enhance instruction adherence and ensure accurate responses. Additionally, we conduct column exploration to iteratively execute SQL queries, progressing from simple to complex, to understand SQL dialects, data types, and nested columns. Finally, we implement a self-refinement workflow to correct answers and apply self-consistency to increase confidence in the outputs. To further boost reliability, we apply parallelization by running the entire workflow across multiple threads simultaneously and employ a voting mechanism to determine the most likely correct outcome. Due to the challenge of the dataset and also our strict consistency mechanism, sometimes executing the generated SQL query does not return any rows. For these examples, we apply a Common Table Expression (CTE) based self-refinement approach. CTE is a temporary result set that can be used in a SQL query. We parse agent-generated SQLs and extract CTE statements for execution. In this way, we allow the agent to examine the intermediate CTE results and approach a solution step-by-step.

We evaluate our methods on two tasks of Spider 2.0 (Lei et al., 2024): Spider 2.0-Snow and Spider 2.0-Lite. Spider 2.0-Snow is unanimously in Snowflake SQL, while Spider 2.0-Lite includes examples in BigQuery and SQLite, additionally. ReFoRCE supports multiple dialects with minimal changes in prompts, making it versatile across various database systems. Our ReFoRCE agent achieves state-of-the-art results, scoring 31.26 on Spider 2.0-Snow and 30.35 on Spider 2.0-Lite, outperforming Spider-Agent's score around 20. This demonstrates the effectiveness of our method in handling multiple dialects, nested columns and complex data types in Spider 2.0.

2 Related Work

2.1 Text-to-SQL Methods

Recent Text-to-SQL methods primarily involve fine-tuning and LLM-prompting techniques. Fine-tuning approaches (Wang et al., 2019; Scholak et al., 2021; Li et al., 2023; 2024a) focus on optimizing models for benchmarks by capturing schema representation, query formats, and logical relationships. In contrast, LLM-prompting (Zhang et al., 2023; Gao et al., 2023a; Pourreza & Rafiei, 2024; Talaei et al., 2024) leverages carefully crafted prompts, often in few-shot or zero-shot settings, to eliminate the need for task-specific fine-tuning. While these methods excel on simpler datasets like Spider 1.0 (Yu et al., 2018) and BIRD (Li et al., 2024b), they struggle with complex benchmarks such as Spider 2.0 (Lei et al., 2024) due to challenges in database comprehension, ambiguity resolution, and SQL dialect handling. To break down, schema linking, SQL generation, and iterative refinement are the core tasks in Text-to-SQL. Techniques like semantic matching (Kothyari et al., 2023), in-context examples (Gao et al., 2023b), sub-query decomposition (Pourreza & Rafiei, 2024), and self-refinement with memory (Shinn et al., 2024) have advanced SQL generation. Calibration techniques, such as using LLM log probabilities (Ramachandran & Sarawagi, 2024) or direct yes/no validation (Tian et al., 2023), further enhance confidence in SQL correctness, demonstrating the increasing sophistication of Text-to-SQL methods.

2.2 CODING AGENTS

Coding agents enable LLMs to interact dynamically with their environment by using tools, executing commands, observing feedback, and planning actions. Early frameworks like ReAct (Yao et al., 2023) introduced reasoning and acting components, while Self-Debugging (Chen et al., 2023) and InterCode (Yang et al., 2024) showcased iterative problem-solving through debugging and lightweight reinforcement learning. Plan-and-Solve Prompting (Wang et al., 2023) and multi-agent systems like CodeR (Chen et al., 2024) and Reflexion (Shinn et al., 2024) further enhanced task decomposition and iterative improvement. Specialized frameworks, such as Spider Agent (Lei et al., 2024), addressed domain-specific challenges like SQL query generation. However, coding agents often face limitations in specialized tasks, where domain-specific solutions may outperform generalized frameworks (Xia et al., 2024).

Another line of agent research target on structured and predefined workflows that guide LLMs and tools for more reliable performance designed for specific tasks. Originating from concepts like Chain-of-Thought (Wei et al., 2022) and Self-Consistency (Wang et al., 2022), workflows have evolved with advancements such as Flows (Josifoski et al., 2023), AutoGen (Wu et al., 2023), and FlowMind (Zeng et al., 2023), enabling modular, collaborative, and automated workflows. In code generation, frameworks like MetaGPT (Hong et al., 2023) and AlphaCodium (Ridnik et al., 2024) have demonstrated the utility of structured workflows in coding tasks. While existing coding agents and workflows address iterative refinement and modular problem-solving, ReFoRCE uniquely integrates table compression, format restriction, and iterative column exploration as new weapons to tackle enterprise-scale SQL challenges.

3 METHODOLOGY

3.1 PRELIMINARIES

Spider 2.0 (Lei et al., 2024) is a comprehensive code agent task where, given a question $\mathcal{Q}$, a database interface $\mathcal{I}$, and a codebase $\mathcal{C}$ (including context, configuration, and documentation as shown in Fig. 1), the goal is to iteratively modify the code (SQL/Python) $\mathcal{C}$ based on observations $\mathcal{O}_k = \text{execute}(\mathcal{C}, \mathcal{I}, \mathcal{Q})$ until the final result $\mathcal{A}$ (text/table/database) is obtained. The final observation $\mathcal{O}_k$ serves as the agent's answer to the question, i.e., $\mathcal{A} = \mathcal{O}_k$. In contrast, Spider 2.0-snow and Spider 2.0-lite are self-contained Text-to-SQL tasks. Given a database schema $\mathcal{D}$, a natural language question $\mathcal{Q}$, and auxiliary documentation $\mathcal{E}$, the Text-to-SQL parser $f(\cdot)$ generates the SQL query $\mathcal{S} = f(\mathcal{Q}, \mathcal{D}, \mathcal{E} \mid \theta)$, where θ denotes the parser's parameters.

Lei et al. (2024) introduced Spider-Agent, a framework built on the ReAct Yao et al. (2023) paradigm with function-calling capabilities such as `EXEC_SQL` for executing SQL queries and `TERMINAL` for performing command-line operations to navigate DBT (Data Build Tool) projects and read schema-related files. The agent operates by receiving observations, which represent the current state of the environment or the outcome of a function call initiated by the agent. Based on these observations, the agent generates a "Thought" and selects an appropriate "Action" from a predefined list of function calls. The task is considered complete when the agent invokes the `TERMINATE` function.

3.2 REFORCE: SELF-REFINEMENT AGENT WITH FORMAT RESTRICTION AND COLUMN EXPLORATION

Since ReAct agents have a high degree of freedom, their workflow lacks necessary reliability and predictability. To address this problem, we propose **ReFoRCE** (Self-**Re**finement Agent with **Fo**rmat **R**estriction and **C**olumn **E**xploration), which simplifies the process by dividing it into manageable subtasks for greater control. ReFoRCE employs a self-refinement workflow that incorporates format restriction and column exploration to identify challenging examples. We implement a self-refinement workflow to correct answers and apply self-consistency to increase confidence in the outputs, as shown in Figure 1. To further boost reliability, we apply parallelization by running the entire workflow across multiple threads simultaneously and employ a voting mechanism to determine the most likely correct outcome. Due to the challenge of the dataset and also our strict consistency mechanism, sometimes executing the generated SQL query does not return any rows. For these examples, we apply a Common Table Expression (CTE) based self-refinement approach. CTE is a temporary result

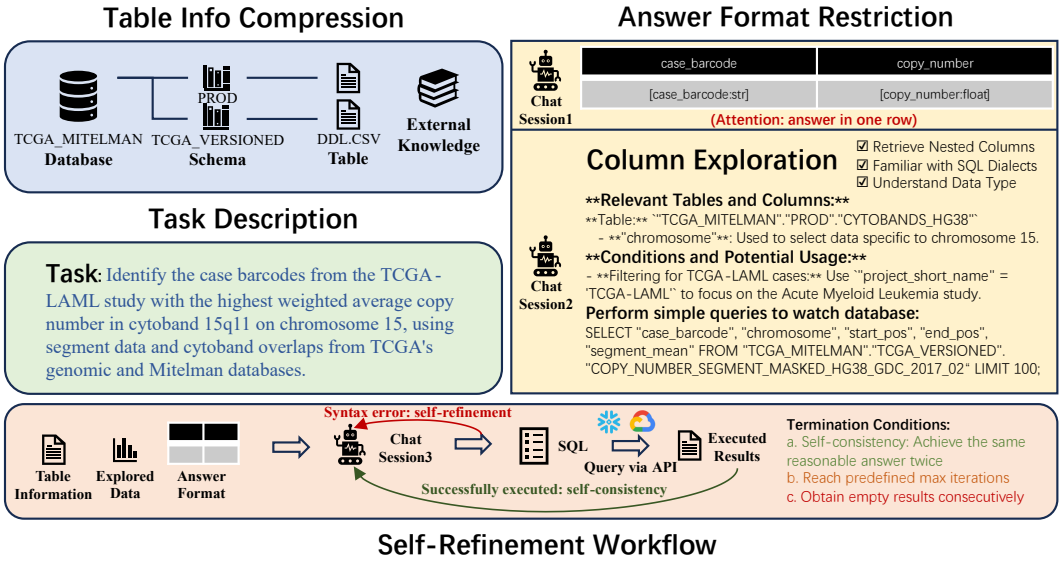

Figure 1: An overview of our Self-**Re**finement Agent with **Fo**rmat **R**estriction and **C**olumn **E**xploration (**ReFoRCE**) workflow.

set that can be used in a SQL query. We parse agent-generated SQLs and extract CTE statements for execution. In this way, we allow the agent to examine the intermediate CTE results and approach a solution step-by-step. Notably, these techniques work alike on various database systems, thus adding support for a new type of database takes as little effort as adding several prompts.

3.2.1 Table Information Compression

Following the approach of Spider 2.0-Snow (Lei et al., 2024), we create a dictionary for each example, incorporating external knowledge and table structures using Database Definition Language (DDL) files. In some specific examples, DDL files exceed 300 KB, surpassing the context limitations of models like ChatGPT. To address this, we apply a pattern-based matching that merge tables with similar prefixes or suffixes. For these tables, we retain only one representative DDL file as input, while for others, we provide only the table names to the model.

Given the database information $\mathcal{D}$ and auxiliary documentation $\mathcal{E}$, we apply the `compress` function to compress it and concatenate the result with the question $\mathcal{Q}$ as the initial input prompt $\mathcal{P}_{\text{init}}$:

$$\mathcal{P}_{\text{init}} = \texttt{compress}(\mathcal{D}) + \mathcal{E} + \mathcal{Q}. \tag{1}$$

For example, in the GA360 database, there is one year of data with table names ranging from `GA_SESSIONS_20160801` to `GA_SESSIONS_20170801`. Each table's DDL file occupies more than 150 KB, resulting in a total length of over 50 MB—an impractically large size for LLMs to process. Our pattern-based compression significantly reduces DDL file sizes of such databases.

3.2.2 Expected Answer Format Restriction

Realistic Text-to-SQL problems often face challenges related to long context issues. When the context exceeds 100k tokens, the model may lose critical information, such as detailed task descriptions. In cases where the task description is clear and unambiguous, determining the expected answer format (e.g., column names, data types, and the number of rows) is typically straightforward. However, when overwhelmed by excessive information, LLMs often struggle to follow instructions accurately. Even with prompts designed to guide or correct the output, LLMs may keep generating incorrect answers and fail to correct their responses.

To address this, we propose Expected Answer Format Restriction, which involves generating the expected format at the outset and consistently reinforcing this format during self-refinement. The response must strictly adhere to the specified format in CSV style, ensuring alignment with executed CSV files. Each column should be explicitly defined, including all necessary attributes, and

each record should occupy a separate row. The format should account for specific cases, such as superlatives, percentages, or coordinates, ensuring the output is concise, clear, and unambiguous. For ambiguous terms, potential values or additional columns can be added to maintain clarity and precision. Figure 1 illustrates a case of format restriction. For example, when a question asks for the highest number, the answer should be presented in a single row. Annotations such as "answer in one row" can be appended to the format table to ensure clarity. Additionally, in the Spider 2.0 evaluation setting (Lei et al., 2024), it is acceptable to include extra columns. For instance, even if the task only requires barcodes, our format also includes a "copy number" column, which is highly relevant to the task. This approach ensures consistency and accuracy, even when managing long contexts and complex task descriptions.

For an LLM chat session $\mathcal{L}_{\text{session}}$, we input initial prompts alongside format prompts $\mathcal{P}_{\text{format}}$ to generate the expected answer format $\mathcal{F}$:

$$\mathcal{F} = \mathcal{L}_{\text{session}}(\mathcal{P}_{\text{init}}, \mathcal{P}_{\text{format}}). \tag{2}$$

3.2.3 Exploration of Potentially Useful Columns

When directly providing the entire database information to an LLM, the lack of details on value types and SQL dialects often leads to repeated iterations for refining syntax errors, correcting data types, or invoking the correct functions. This process is not only time-consuming but also leaves little room for real reasoning. For baselines such as DAIL-SQL (Gao et al., 2023a) and DIN-SQL (Pourreza & Rafiei, 2024) in Spider 2.0-Lite, sample rows provided by the Lite dataset are often used to help the model understand the structure of the tables. However, for nested columns, even a few rows can be too lengthy to be fed into the LLM. Additionally, specific values in sample rows often lack diversity and are biased, which can mislead the model into generating incorrect answers based on these limited samples.

To address these challenges and ensure a comprehensive understanding of the database structure, we design a systematic approach to explore potentially useful columns. The process begins with identifying relevant tables and columns, guided by prompts designed to extract meaningful information. Dynamically generated SQL queries progress from simple, non-nested formats to more complex ones, enabling a gradual understanding of the database and arriving at the correct answer. These queries follow the structure `SELECT DISTINCT "COLUMN_NAME" FROM DATABASE.SCHEMA.TABLE WHERE ...` (specific to the Snowflake dialect) and adhere to constraints such as avoiding Common Table Expressions (CTEs) and schema-level checks, while limiting the output to 100 rows or 5 KB per query. Columns in JSON or nested formats are explicitly handled using techniques like `LATERAL FLATTEN` to extract nested values. Additionally, string-matching queries utilize fuzzy patterns (e.g., `%target_str%`) to enhance flexibility.

Here, we employ an additional LLM chat session, $\mathcal{L}'_{\text{session}}$, where the input consists of $\mathcal{P}_{\text{init}}$ alongside column exploration prompts $\mathcal{P}_{\text{exploration}}$. This generates exploration content, including relevant tables and columns $\mathcal{P}_{\text{column}}$ and SQL queries $\mathcal{S}_{\text{exploration}}$ for interacting with the database. Database APIs are then invoked to execute the SQL queries and retrieve results $\mathcal{R}_{\text{exploration}}$:

$$\mathcal{P}_{\text{column}}, \mathcal{S}_{\text{exploration}} = \mathcal{L}'_{\text{session}}(\mathcal{P}_{\text{init}}, \mathcal{P}_{\text{exploration}}), \tag{3}$$

$$\mathcal{R}_{\text{exploration}} = \texttt{API}(\mathcal{S}_{\text{exploration}}). \tag{4}$$

During this stage, the LLM generates more than 10 SQL queries simultaneously, making it essential to execute each query accurately for effective exploration. However, if one SQL query contains an error, subsequent queries are likely to exhibit similar issues. To address this, we propose Algorithm 1, which offers a structured approach to executing SQL queries while dynamically addressing errors through self-correction.

3.2.4 Self-Refinement Workflow for Problem-Solving

Self-Refinement with Execution Feedback After obtaining the table information $\mathcal{P}_{\text{init}}$, exploring value data $\mathcal{P}_{\text{column}} + \mathcal{R}_{\text{exploration}}$, and defining the expected answer format $\mathcal{F}$, we input these elements into the model and employ a self-refinement process. This process enables the model to correct errors, improve its answers, and achieve results $\mathcal{R}_{\text{final}}$ with high confidence through self-consistency.

$$\mathcal{R}_{\text{final}} = \texttt{self-refinement}(\mathcal{P}_{\text{init}}, \mathcal{P}_{\text{column}} + \mathcal{R}_{\text{exploration}}, \mathcal{F}). \tag{5}$$

The details in self-refinement workflow are presented in Algorithm 2. Beginning with purified information, it generates SQL queries and executes them via an API, evaluating their correctness and reasonableness based on the returned results. Identified errors are iteratively refined to improve accuracy. The refinement process is governed by termination conditions, including achieving self-consistency, where the same reasonable answer is obtained twice. Empty results or columns containing only empty strings or zeros are treated as incorrect and excluded from consistency checks. The workflow also halts if a predefined maximum number of iterations is reached, as this may indicate that the model is producing low-confidence answers, which are treated as empty. Furthermore, consecutive empty results signal the termination of further refinements. This conservative strategy prioritizes high confidence in non-empty answers, even at the cost of retaining a significant number of empty results, ensuring robust and reliable outputs.

CTE-based Refinement Additionally, if the self-refinement workflow fails to generate a SQL, we further attempt to construct a step-by-step data flow using Common Table Expressions (CTE). CTEs are intermediate tables that helps breaking down a complex query into multiple easier queries. We explicitly prompt the agent to generate the SQL as a set of meaningful CTEs conjoined.

After the agent generates a SQL, we parse it to obtain the individual CTEs and obtain execution results at the end of each CTE. The agent is asked to verify each execution result individually. If a result is not as expected, the agent has the choice to rewrite the current CTE. This pipeline allows the agent to localize errors. This also provides a way for the agent to check the logic of each CTE individually and help explore alternative columns.

3.2.5 PARALLELIZATION

Despite the self-consistency mechanism, variations in results may arise across runs of the same example due to differing perspectives in column exploration. To enhance confidence in the outcomes, we employ parallelization by launching multiple threads to execute the entire workflow simultaneously. The results are first compared programmatically, using a voting mechanism to identify the most likely correct outcome. If voting alone cannot resolve discrepancies, the model further evaluates the results to determine the most accurate answer. This approach considers diverse perspectives and multiple iterations, facilitating convergence on higher-quality outcomes.

Using the same format $\mathcal{F}$ and database information $\mathcal{P}_{\text{init}}$, we launch multiple threads (we set it to 3 in our experiments) to execute the entire process independently in parallel:

$$\mathcal{R}_{\text{parallel}} = \bigcup \text{Parallel} \left\{ \begin{array}{l} \mathcal{P}_{\text{column}}, \mathcal{S}_{\text{exploration}} = \mathcal{L}'_{\text{session}}(\mathcal{P}_{\text{init}}, \mathcal{P}_{\text{exploration}}), \\ \mathcal{R}_{\text{exploration}} = \texttt{API}(\mathcal{S}_{\text{exploration}}), \\ \mathcal{R}_{\text{final}} = \texttt{self-refinement}(\mathcal{P}_{\text{init}}, \mathcal{P}_{\text{column}} + \mathcal{R}_{\text{exploration}}, \mathcal{F}), \end{array} \right\}. \quad (6)$$

Finally, the results from the parallel execution are aggregated through a voting mechanism:

$$\mathcal{R}_{\text{vote}} = \texttt{model_vote}(\mathcal{R}_{\text{parallel}}). \quad (7)$$

Furthermore, since each example is independent and both the model and database APIs support parallel execution, we enable parallelization across different examples as well. This strategy significantly accelerates the overall process while ensuring reliable and consistent performance.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Dataset We evaluate our approach using the Spider 2.0 dataset (Lei et al., 2024), which includes two subsets: Spider 2.0-Snow and Spider 2.0-Lite. Both subsets consist of 547 examples, encompassing over 150 databases with an average of 800 columns per database. Each SQL query contains approximately 150 tokens, making the task particularly challenging. The key difference between the two subsets lies in their SQL dialects: Spider 2.0-Snow focuses exclusively on the Snowflake dialect, whereas Spider 2.0-Lite supports BigQuery, Snowflake, and SQLite dialects.

Table 1: Comparison of Methods with Execution Accuracy (EX) on the Spider 2.0-Snow and Spider 2.0-Lite Datasets. Higher values indicate better performance.

Method	Model	EX (↑) on Spider 2.0-Snow	EX (↑) on Spider 2.0-Lite
ReFoRCE (ours)	o1-preview	**31.26**	**30.35**
Spider-Agent	o1-preview	23.58	23.03
Spider-Agent	GPT-4o	12.98	13.16
DAIL-SQL	GPT-4o	2.20	5.68
CHESS	GPT-4o	1.28	3.84
DIN-SQL	GPT-4o	0.00	1.46
SFT CodeS-15B	–	0.00	0.73

Evaluation Metrics We evaluate performance using the widely adopted metric, Execution Accuracy (EX) (Yu et al., 2018; Li et al., 2024b). For certain examples, ambiguous questions may not explicitly specify which columns to return. The evaluation scripts are designed to focus on the essential components of the answers, disregarding irrelevant columns and concentrating on the core elements outlined in the instructions. As a result, the inclusion of extra columns is considered acceptable.

Large Language Models We conduct our experiments using the GPT family models, specifically GPT-4o (Achiam et al., 2023) and o1-preview (OpenAI, 2024). GPT-4o is used for testing some baselines, while o1-preview is employed for our method and Spider-Agent. We opted for o1-preview instead of the formal o1 or o3-mini because o1-preview achieves better results for this particular task.

Baselines We employ the state-of-the-art code agent framework Spider-Agent (Lei et al., 2024) for the Spider 2.0-Snow and Spider 2.0-Lite dataset. Additionally, we utilize widely recognized prompting methods such as DAIL-SQL (Gao et al., 2023a), DIN-SQL (Pourreza & Rafiei, 2024) and SFT CodeS (Li et al., 2024a).

4.2 Evaluation Results

The results in Table 1 highlight the superior performance of our method on the Spider 2.0-Snow and Spider 2.0-Lite datasets. Using the `o1-preview` model, our method achieves execution accuracy (EX) scores of 31.26 on Spider 2.0-Snow and 30.35 on Spider 2.0-Lite, significantly outperforming all other methods.[1]

These results demonstrate the robustness of our approach in addressing the challenges posed by the datasets. On Spider 2.0-Snow, our method excels in effectively handling nested columns and complex data types. On Spider 2.0-Lite, which spans multiple dialects, including Snowflake, SQLite, and BigQuery, our method showcases remarkable adaptability and consistency. The slightly lower results on Spider 2.0-Lite are attributed to the fact that our prompts are primarily designed for the Snowflake dialect, leading to occasional errors when handling certain cases in the BigQuery dialect.

Compared to Spider-Agent, which scores around 23 on both datasets using the same `o1-preview` model, our method achieves higher scores, highlighting its superior capability in solving complex cases in Text-to-SQL tasks. Additionally, methods using `GPT-4o`, including Spider-Agent, DAIL-SQL, and CHESS, perform poorly, with scores ranging from 0.00 to 5.68. This underscores the importance of our specialized approach, which is specifically designed to address the intricacies of Text-to-SQL tasks across diverse SQL dialects.

In summary, the results firmly establish our method as the state-of-the-art in handling the challenges of Text-to-SQL tasks, particularly in complex scenarios involving multiple dialects, nested columns, and advanced data types. This achievement demonstrates the efficacy of our approach in pushing the boundaries of execution accuracy in Text-to-SQL parsing.

[1]Leaderboard data as of February 13, 2025.

5 CONCLUSION

In this paper, we proposed **ReFoRCE**, a self-refinement framework for addressing the challenges of real-world Text-to-SQL tasks as exemplified by the Spider 2.0 dataset. By introducing techniques such as table information compression, format restriction, iterative column exploration, and CTE-based refinement, ReFoRCE effectively handles multiple SQL dialects, nested columns, and complex data types. Our approach achieved state-of-the-art performance, with scores of 31.26 on Spider 2.0-Snow and 30.35 on Spider 2.0-Lite, significantly outperforming the previous best results by Spider-Agent. These findings highlight the robustness and reliability of ReFoRCE in tackling real-world complexities.

Despite its strong performance, ReFoRCE has several limitations. The lossy compression of table information restricts its ability to handle very large table contexts, and the current simple exploration strategy struggles with ambiguous columns and local optima. Moreover, as ReFoRCE primarily focuses on preprocessing and self-refinement, it does not introduce significant improvements in reasoning capabilities, which are critical for handling more complex queries.

To address these limitations, future work will explore schema-linking techniques to enhance contextual understanding and advanced column exploration and reasoning strategies such as Monte Carlo Tree Search (MCTS) and Process Reward Model (PRM) or pure Reinforcement Learning (RL) (Guo et al., 2025) to better handle ambiguous columns and improve optimization. By addressing these challenges, we aim to advance the applicability of ReFoRCE to broader Text-to-SQL tasks and set new benchmarks for real-world database interaction.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Anthropic. Model card: Claude 3, 2023. URL `https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf`. Accessed: 2025-01-24.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 17682–17690, 2024.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363*, 2023a.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation, 2023b. URL `https://arxiv.org/abs/2308.15363`.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

Martin Josifoski, Lars Klein, Maxime Peyrard, Nicolas Baldwin, Yifei Li, Saibo Geng, Julian Paul Schnitzler, Yuxing Yao, Jiheng Wei, Debjit Paul, et al. Flows: Building blocks of reasoning and collaborating ai. *arXiv preprint arXiv:2308.01285*, 2023.

Mayank Kothyari, Dhruva Dhingra, Sunita Sarawagi, and Soumen Chakrabarti. CRUSH4SQL: Collective retrieval using schema hallucination for Text2SQL. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 14054–14066, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.868. URL https://aclanthology.org/2023.emnlp-main.868/.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*, 2024.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 13067–13075, 2023.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. Codes: Towards building open-source language models for text-to-sql. *Proc. ACM Manag. Data*, 2(3), May 2024a. doi: 10.1145/3654930. URL https://doi.org/10.1145/3654930.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024b.

OpenAI. Openai api, 2024. URL https://platform.openai.com/docs. Accessed: 2025-01-15.

Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems*, 36, 2024.

Ashwin Ramachandran and Sunita Sarawagi. Text-to-sql calibration: No need to ask – just rescale model probabilities, 2024. URL https://arxiv.org/abs/2411.16742.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093*, 2021.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*, 2024.

Katherine Tian, Eric Mitchell, Allan Zhou, Archit Sharma, Rafael Rafailov, Huaxiu Yao, Chelsea Finn, and Christopher D. Manning. Just ask for calibration: Strategies for eliciting calibrated confidence scores from language models fine-tuned with human feedback, 2023. URL https://arxiv.org/abs/2305.14975.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942*, 2019.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*, 2023.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: synergizing reasoning and acting in language models (2022). *arXiv preprint arXiv:2210.03629*, 2023.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, 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.

John M Zelle and Raymond J Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pp. 1050–1055, 1996.

Zhen Zeng, William Watson, Nicole Cho, Saba Rahimi, Shayleen Reynolds, Tucker Balch, and Manuela Veloso. Flowmind: automatic workflow generation with llms. In *Proceedings of the Fourth ACM International Conference on AI in Finance*, pp. 73–81, 2023.

Luke S Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *arXiv preprint arXiv:1207.1420*, 2012.

Hanchong Zhang, Ruisheng Cao, Lu Chen, Hongshen Xu, and Kai Yu. Act-sql: In-context learning for text-to-sql with automatically-generated chain-of-thought. *arXiv preprint arXiv:2310.17342*, 2023.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.

A ALGORITHM FOR SQLS LIST EXECUTION WITH SELF-CORRECTION DURING COLUMN EXPLORATION

Algorithm 1: SQL List Execution with Self-Correction during Column Exploration

Input: List of SQL queries `sqls`, Chat session `chat_session`
Output: Result dictionary `result_dic`, Updated chat session `chat_session`

```
Initialize an empty dictionary result_dic and an error record list error_rec;
while sqls is not empty do
    Extract the first query sql from sqls and remove it from the list;
    Execute the query: results = execute_sql_api(sql);
    Initialize check_again_flag = False;
    if results are valid and not empty then
        Attempt to parse results into a DataFrame df_csv and check for empty or invalid
          columns;
        if no empty columns or invalid data found then
            Add sql and its results to result_dic;
            Append a message containing the query and results to
              chat_session.messages;
            continue;
    if results are invalid or require rechecking then
        Initialize max_iter = 3, simplify = False, and corrected_sql = None;
        while max_iter > 0 and correction is needed do
            Generate a corrected query using self_correct(sql, results,
              chat_session);
            Re-execute the corrected query: results =
              execute_sql_api(corrected_sql);
            Decrease max_iter by 1;
            if correction succeeds and results are valid then
                Add corrected_sql and its results to result_dic;
                Append a message containing the corrected query and results to
                  chat_session.messages;
                if results is a valid string and not empty then
                    Append 1 to error_rec;
                    Request additional corrections from the chat session for remaining queries:
                    Prompt the model: response =
                      chat_session.get_model_response();
                    Parse response into a list of SQL queries ;
                    response_sql. foreach query s in response do
                        Parse and clean each query, appending to response_sql;
                    if len(response_sql) ≥ len(sqls) then
                        Replace sqls with response_sql;
                break;
        if no valid results after 3 iterations then
            Append failure to error_rec;
            if the last 5 error records are all failures then
                return result_dic, chat_session;
            continue;
return result_dic, chat_session;
```

The algorithm describes a process for executing a list of SQL queries with a self-correction mechanism during column exploration. It iteratively processes each SQL query from the input list, executes it, and evaluates the results. If the results are valid and complete, the query and its output are stored. If the results indicate issues (e.g., missing or invalid data), the algorithm attempts to correct the query using a self-correction method that generates and refines SQL queries based on error feedback, up

to a maximum of three iterations. During this process, additional corrections for similar queries are dynamically generated by interacting with a large language model (LLM) chat session. The model suggests updated SQL queries, which can replace the remaining queries in the list if they are deemed sufficient. This iterative approach ensures robustness in handling invalid results and improves query accuracy by dynamically adapting to errors, while maintaining a focus on generating meaningful results for the task.

B ALGORITHM FOR SELF-REFINEMENT WORKFLOW FOR SQL QUERY EXECUTION

Algorithm 2: Self-Refinement for SQL Query Execution

Input: Input prompts $\mathcal{P}$, Chat session `chat_session`, API type `api`
Output: Refined SQL query and results

Initialize iteration counter `itercount` to 0, and empty lists `results_tables`;
Construct the initial query environment $\mathcal{P}$ using table information, task description, column exploration, format restriction, and API-specific query guidelines;
while *`itercount` < `max_iter`* **do**
  Generate the next SQL query from $\mathcal{P}$ using the chat session model;
  Execute the query: `response = execute_sql_api(response, api);`
  if *`response` contains valid results* **then**
    Parse the results into a DataFrame `df_csv` and process as follows:
      - Round numeric columns to two decimal places.
      - Check if nested values.

    If the results are unique and conform to the expected format, append them to `results_tables`;
    Save the refined query and results to the designated paths;
    if *the results appears in `results_tables` twice* **then**
      Satisfy self-consistency;
      break
  else if *`response` indicates an error or invalid data* **then**
    Refine the SQL query by addressing issues such as:
      - Missing or empty columns.
      - Incorrect handling of nested data or sorting constraints.
      - Syntax errors or invalid API-specific operations.

    Update $\mathcal{P}$ with error feedback and retry the process;
  Increment `itercount`;
  if *three or more consecutive errors occur* **then**
    Terminate the process and discard any incomplete save files;
if *`itercount` reaches `args.max_iter`* **then**
  Discard any incomplete save files;

The algorithm describes a self-refinement workflow for iteratively generating, executing, and refining SQL queries. It begins by constructing an initial query environment ($\mathcal{P}$) that incorporates input prompts, including table information, task descriptions, column exploration strategies, format restrictions, and API-specific query guidelines. During each iteration, a new SQL query is generated using the chat session model and executed via the specified API. If the query returns valid results, the results are processed by rounding numeric columns to two decimal places and handling nested values. The results are checked for uniqueness and format compliance, and if they satisfy these criteria, they are stored in a results table (`results_tables`). Self-consistency is achieved by verifying if the same result appears in `results_tables` twice, at which point the process terminates. If errors or invalid data are encountered, the algorithm refines the query by addressing issues such as missing or empty columns, improper handling of nested data, or syntax errors, and retries the process with the updated query. The workflow terminates either when valid results meet self-consistency, when

the maximum number of iterations (`max_iter`) is reached, or when three consecutive errors occur, ensuring incomplete or invalid outputs are discarded. This structured approach improves robustness and accuracy in SQL query execution.