From Simple to Complex: An Agent Framework with a Progressive Difficulty Planning Strategy for Text-to-SQL

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have shown significant potential in text-to-SQL tasks. However, most existing methods operate within simplified scenarios with pre-prepared inputs (*i.e.,* questions and database schemas) and outputs (*i.e.,* predicted SQL). In contrast, real-world SQL development often requires consulting external knowledge and interacting with the database environment through iterative steps. To address this, we propose SoC-Agent, a novel LLM-powered agent framework designed for SQL generation in complex environments. SoC-Agent emulates the human iterative development process, breaking down tasks into a series of subtasks of increasing difficulty. Specifically, the agent first tackles simpler subtasks, iteratively refining its approach based on previous results, and then addresses more complex tasks. This incremental strategy enhances the agent's reasoning ability for complex SQL generation. Additionally, agent can also leverage external knowledge sources and dynamically interacts with the database environment to gather necessary information for each subtask, ensuring that the results are both accurate and contextually relevant. We evaluate our method on Spider 2.0 dataset, specifically designed for agentic tasks, demonstrating the superiority in handling complex SQL generation. Our codes are available at: `https://anonymous.4open.science/r/SoC-Agent-694B`.

1 Introduction

Text-to-SQL (Qin et al., 2022; Deng et al., 2022; Kumar et al., 2022) aims to translate natural language queries into SQL statements, enabling users to interact with databases without needing database expertise. It enhances database accessibility and usability, allowing non-experts to perform data operations through intuitive language inputs, providing greater convenience for data analysis.

In recent years, the remarkable success of Large Language Models (LLMs) in various fields has led to the emergence of LLM-based methods (Shi et al., 2024; Liu et al., 2024b; Mohammadjafari et al., 2024; Zhu et al., 2024b) as the mainstream paradigm in the text-to-SQL domain. These methods can be broadly categorized into two main approaches: ❶ **Prompt-based methods** leverage the zero-shot in-context learning (ICL) capabilities of LLMs for SQL generation. Building on this foundation, subsequent efforts, including DIN-SQL (Pourreza and Rafiei, 2024a), DAIL-SQL (Gao et al., 2024), MAC-SQL (Wang et al., 2024a), and C3 (Dong et al., 2023), have enhanced LLM performance through schema-linking, question representation, task decomposition, and techniques such as chains of thought (CoT) (Wei et al., 2022) and self-consistency (Wang et al., 2023). ❷ **Fine-tuning-based methods** aim to elevate the capabilities of open-source LLMs through supervised fine-tuning (SFT), with the goal of aiming to match or surpass the close-source LLMs. For instance, DTS-SQL (Pourreza and Rafiei, 2024b) introduces a

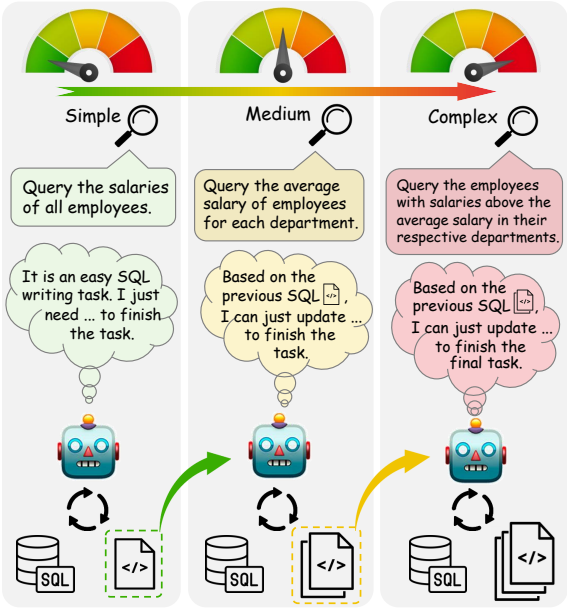


Figure 1: The workflow of our proposed agent planning framework (SoC) for text-to-SQL tasks.

two-stage SFT method, incorporating SFT in both the schema-linking and SQL generation stages; CHESS (Talaei et al., 2024) combines ICL and SFT strategies; and SENSE (Yang et al., 2024a) further improves the capabilities of open-source models by synthesizing data through strong models.

Despite their effectiveness, most studies (Yu et al., 2018; Li et al., 2024c) operate in relatively simple text-to-SQL scenarios, where LLMs are expected to directly produce the predicted SQL for a given query and database schema. However, in real-world complex SQL writing tasks, developers often need to frequently consult external knowledge documents and interact with the database environment to complete the task through multiple plans and steps (Lei et al., 2024). Moreover, language agents (Guo et al., 2024; Wang et al., 2024b; Xi et al., 2023), which utilize the advanced reasoning abilities of LLMs to interface with executable tools, have become crucial elements of AI systems intended to tackle complex tasks (Liu et al., 2024a). These language agents offer promising potential for automatic SQL generation in real database development scenarios. Although some LLM-powered agent frameworks have been designed to solve code generation problems (Chen et al., 2024; Xia et al., 2024; Yang et al., 2024b; Pan et al., 2024), the design of agent frameworks specifically for text-to-SQL scenarios still remains largely unexplored.

To address this gap, we focus on developing an effective LLM-powered agent framework tailored for a realistic SQL development environment. Drawing inspiration from the SQL writing strategies employed by professional database developers, we note that they frequently use a simple-to-complex strategy (Huang et al., 2023). This involves breaking down a complex task into manageable sub-problems, which are then solved individually and iteratively combined to address the original task. This observation leads us to a critical question: how can we integrate this problem-solving methodology into the design of an agent framework for handling complex text-to-SQL tasks?

In this paper, we propose a Simple-to-Complex Agent planning framework (SoC-Agent). Specifically, for a given task, it decomposes the task into a series of versions ranging from simple to complex, solving them sequentially. As illustrated in Figure 1, the agent first addresses simpler versions of the subtask, iteratively refining its approach based on previous results, and then tackles more complex tasks. This incremental strategy enhances the agent's ability for complex SQL generation. Additionally, during the completion of these tasks, SoC-Agent utilizes external SQL knowledge by invoking tools and dynamically interacting with the database to ensure accurate task completion. We conducted extensive experiments on Spider 2.0 (Lei et al., 2024), a real-world enterprise-level text-to-SQL dataset specifically customized for agentic settings. The experimental results demonstrate the effectiveness of the proposed framework.

Our main contributions can be summarized as follows:

- We emphasize the necessity of addressing text-to-SQL tasks within agentic task settings, a domain that remains largely underexplored.
- We introduce a novel agent framework, SoC-Agent, which employs a simple-to-complex planning strategy to enhance the performance of agents in text-to-SQL tasks.
- We validate the effectiveness of our approach on the latest public dataset, Spider 2.0, achieving state-of-the-art results. Case studies further verifies the efficacy of our approach.

2 Preliminaries

2.1 Problem Definition of Text-to-SQL

Consider an input triplet $\mathcal{X} = (\mathcal{T}, \mathcal{D}, \mathcal{K})$, where $\mathcal{T}$ represents a natural language task, $\mathcal{D}$ denotes the database schema, and $\mathcal{K}$ stands for optional external knowledge. The objective of the text-to-SQL task is to generate the correct SQL query $\mathcal{S}^*$ that corresponds to the given task $\mathcal{T}$. Text-to-SQL can be framed as a generation problem, where a LLM $\mathcal{M}$ is guided to produce the correct SQL query by designing suitable prompts:

$$\max_{f} \mathbb{P}_{\mathcal{M}}(\mathcal{S}^* \mid f(\mathcal{T}, \mathcal{D}, \mathcal{K})), \tag{1}$$

where the function f determines the representation of the target task $\mathcal{T}$, the database schema $\mathcal{D}$, and any additional external knowledge $\mathcal{K}$ necessary to complete the task. Additionally, f can incorporate elements such as instructional statements, rule implications, and foreign key information.

2.2 Text-to-SQL in Agentic Setting

The primary distinction from traditional text-to-SQL tasks is that the agentic setting includes an SQL development environment, necessitating the agent to accomplish the final task through multiple interactions with the database and command line

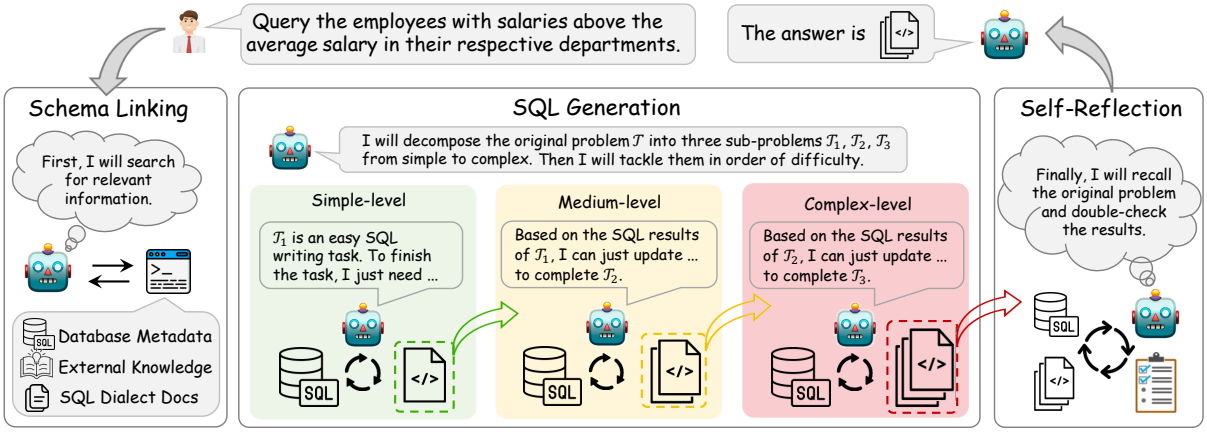


Figure 2: The overview of the proposed SoC-Agent framework.

interface. The agentic setting with an SQL development environment was first introduced by Spider 2.0 (Lei et al., 2024). Our work adheres strictly to this framework. Specifically, given a task $\mathcal{T}$, a database interface $\mathcal{I}$, and a codebase $\mathcal{C}$ (which includes project context, configuration, and documentation), the task involves iteratively modifying the code (*e.g.,* Bash/SQL/Python) based on observations $\mathcal{O}_i = \text{execute}(\mathcal{C}, \mathcal{I}, \mathcal{T})$ until the final result $\mathcal{R}$ (such as text, table, or database) is achieved. In essence, the final observation $\mathcal{O}$ serves as the agent's response to the question, *i.e.,* $\mathcal{R} = \mathcal{O}$.

3 Methodology

In this section, we present our proposed SoC-Agent for text-to-SQL tasks. The framework, as depicted in Figure 2, consists of three primary modules: schema linking, SQL generation, and self-reflection. Each of these components is discussed in detail below.

3.1 Schema Linking

When given a natural language task, the agent must retrieve pertinent information, such as tables, fields, and other external knowledge bases necessary to complete the task. Hence, we first outline the workspace required for the agent to perform its tasks and the process of information retrieval.

Action Space. We follow the environment settings of Spider 2.0 and action space of Spider-Agent (Lei et al., 2024), which require the agent to complete the final task through multiple rounds of interaction with the database and command line interface. The tools available to the agent are defined as follows:

- **Command line operations:** Inspect files and execute scripts using shell commands.
- **File operations:** Generate and modify files.

Algorithm 1: SoC Planning

Input: The user's task $\mathcal{T}$.
Output: The generated SQL $\mathcal{S}$.

Initialization:
  Start with an initial empty SQL $\mathcal{S}_0 \leftarrow \phi$;
  Decompose the task $\mathcal{T}$ into N versions: $[\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_N]$, where $\mathcal{T}_N = \mathcal{T}$ and the order of task complexity is $\mathcal{T}_1 \leq \mathcal{T}_2 \leq, \ldots, \leq \mathcal{T}_N$;
for $i = 1$ *to* N **do**
  Based on the previous SQL $\mathcal{S}_{i-1}$, write a new SQL $\mathcal{S}_i$ to address $\mathcal{T}_i$;
  Execute $\mathcal{S}_i$ in the database system to obtain the result $\mathcal{R}_i$;
  if $\mathcal{R}_i$ *does not solve* $\mathcal{T}_i$ **then**
    Continuously modify and execute $\mathcal{S}_i$ until it satisfactorily addresses $\mathcal{T}_i$.

- **SQL execution operations:** Run SQL queries by interfacing with a local or cloud-based database API.
- **Termination operations:** The agent can assess whether the task is complete or has failed, thus concluding the task process.

Information Collection. Much like how humans navigate files, the agent is instructed to first survey the files in the current directory and pinpoint the information most pertinent to the user's query, including tables, fields, and external knowledge. In this module, we refer to the prompt in Spider-Agent (Lei et al., 2024) and make appropriate adjustments and improvements. We provide specific prompts, please refer to Appendix A.1 and A.2 for details.

3.2 SQL Generation

Once the relevant information has been gathered, the agent moves on to the crucial stage of SQL

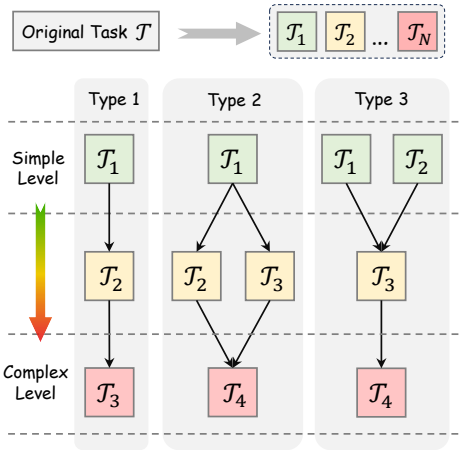


Figure 3: Illustration of different types of SQL canvas.

generation. Producing a completely accurate SQL query in a single attempt is particularly challenging, especially for complex tasks. To address this, we propose an innovative planning strategy to guide the agent in effectively and accurately completing intricate SQL generation tasks.

Task decomposition. Initially, the agent decomposes task into several subtasks, arranged from simple to complex. Specifically, for a given task $\mathcal{T}$, the agent decomposes it into $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_N$, where the complexity relationship is $\mathcal{T}_1 \leq \mathcal{T}_2 \leq \ldots \leq \mathcal{T}_N$.

SoC Planning. Drawing inspiration from professional SQL developers, who often divide intricate tasks into manageable subtasks and iteratively refine the SQL scripts, we adopt a similar idea. The SQL for each subtask is progressively refined and expanded to address subsequent subtasks, culminating in the completion of the overall task. Figure 3 presents a schematic diagram of various sub-SQL canvases generated by a professional SQL developer when tackling a complex problem $\mathcal{T}$. Our goal is for the agent to emulate the strategies employed by professional SQL developers to handle more challenging SQL generation tasks. The process of solving the SQL generation problem based on the SoC process is detailed in Algorithm 1. Our SoC planning strategy incorporates two types of prompts: a pure text description of the workflow, as shown in Figure 11 in Appendix A.3, and a pseudocode workflow description for LLM, as shown in Figure 12 in Appendix A.3. In our experiments, we consider both types of prompts to ensure clarity in the detailed planning process.

Demonstrations. Inspired by the few-shot strategy in in-context learning, we also provide an example to illustrate the process of solving a specific problem using the LLM Agent. We use the question shown in Figure 2: "Query the employees with salaries above the average salary in their respective departments." We simulate the agent's thought and action process to complete the entire SQL generation task. Specifically, the agent begins by decomposing the task into a series of sub-tasks, ranging from simple to complex. The agent then addresses each sub-task sequentially, following the progression from simpler to more complex tasks. Finally, the agent reviews the problems and verifies the results to ensure accuracy. For specific prompt details, please refer to Figure 14 in Appendix A.5.

3.3 Self-Reflection

Task Recall. While the LLM Agent can effectively enhance the success rate of solving complex problems through the SoC strategy, it may forget crucial details of the original problem after multiple rounds of interaction. Despite the final task $\mathcal{T}_N = \mathcal{T}$, the agent might overlook or misinterpret some aspects of the original problem due to language and literal rewrites. To mitigate this, we require the agent to recall the original task upon completing the task, ensuring no important issues are neglected.

Result Verification. Beyond reviewing the problem, the agent must also reflect on whether there is a discrepancy between the SQL-generated results and the original problem requirements. Although the agent generally adheres to the original requirements, it may still commit basic errors, leading to suboptimal results. Common issues include:

- **Data Validity**: Ensure the SQL query results are not empty and contain valid data. An empty file or one with only headers indicates an incorrect SQL query.
- **Sample Size Limitation**: Verify if the task specifies extracting the "most X", "top X", or "first X" entities. If so, include "LIMIT X" in the SQL to appropriately restrict the result set.
- **Field Completeness**: Always return both the entity ID and the entity name for any identified players or entities, along with any other relevant details.

Additionally, there are important result checks and precautions that the model must consider. For specific prompts, please refer to Figure 13 in the Appendix A.4. This reflection strategy helps prevent the agent from overlooking details, thereby reducing the likelihood of basic errors.

Table 1: Statistics of the datasets.

Dataset	# Test Examples	# Easy Examples	# Medium Examples	# Hard Examples	# Test DB	# Col. / DB	# Tok. / SQL	# Func. / SQL
Spider 2.0-lite	547	128	246	173	158	803.6	144.5	6.5
Spider 2.0-snow	547	128	246	173	152	812.1	161.8	6.8

Table 2: Performance comparison. Numbers in **bold** indicate the best performance.

Method	Spider 2.0-snow				Spider 2.0-lite			
	Easy	Medium	Hard	Overall	Easy	Medium	Hard	Overall
SFT CodeS-15B	0.00%	0.00%	0.00%	0.00%	1.65%	0.86%	0.00%	0.73%
DIN-SQL + GPT-4o	0.00%	0.00%	0.00%	0.00%	5.79%	0.43%	0.00%	1.46%
CHESS + GPT-4o	4.69%	0.41%	0.00%	1.28%	9.92%	3.00%	1.24%	3.84%
DAIL-SQL + GPT-4o	6.25%	1.63%	0.00%	2.20%	13.20%	5.58%	1.24%	5.68%
Spider-Agent + QwQ-32B	-	-	-	8.96%	-	-	-	11.33%
Spider-Agent + DeepSeek-R1	-	-	-	10.55%	-	-	-	**13.71%**
Spider-Agent + GPT-4o	19.53%	10.16%	5.20%	10.79%	21.09%	10.57%	4.05%	10.97%
SoC-Agent + QwQ-32B	**22.66%**	**11.38%**	**6.94%**	**12.61%**	**25.00%**	**12.60%**	**4.62%**	12.98%

4 Experiments

In this section, we conduct experiments to answer the following research questions:

- **RQ1**: How does SoC-Agent perform in a real-world, complex SQL development environment?
- **RQ2**: Can we conduct a more in-depth analysis of the SoC-Agent framework during SQL development tasks?
- **RQ3**: Does SoC-Agent follow the planned steps to complete the SQL generation task?

4.1 Experimental Settings

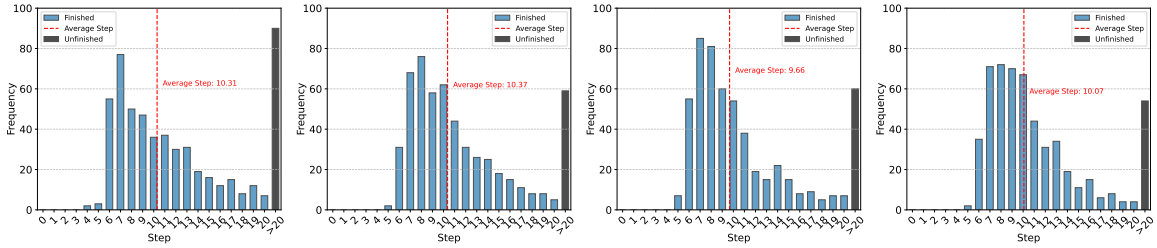
Datasets. We select two versions of the Spider 2.0 dataset, namely Spider 2.0-lite and Spider 2.0-snow, for experiments. Consistent with (Lei et al., 2024), we categorize task difficulty based on the length of the golden SQL: < 80 tokens as easy task, $80 \sim 159$ as medium task, and ≥ 160 as hard task. The detailed statistical information of the dataset is shown in Table 1.

Metrics. Following the settings recommended in (Lei et al., 2024), we use the Execution Accuracy (EX) metric to assess the accuracy of SQL execution results. The evaluation scripts accept output in the form of strings, tables, or databases. For each example, an evaluation script is run, producing a score of either 0 or 1.

Baselines. We evaluate our approach against several state-of-the-art and widely recognized text-to-SQL methods. These include LLM-prompting techniques such as DIN-SQL, DAIL-SQL, and CHESS. Additionally, we consider SFT CodeS, which involves fine-tuning open-source models on large text-to-SQL datasets, and Spider-Agent, an agent-based text-to-SQL framework. Consistent with (Lei et al., 2024), we optimize prompt structures across all methods to ensure they are well-suited to the tasks at hand.

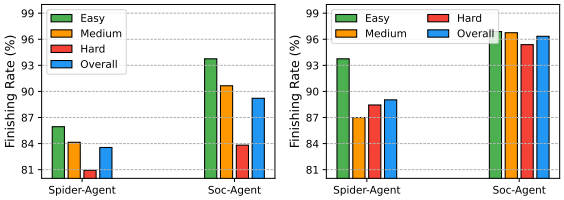
- **SFT CodeS** (Li et al., 2024b) is a series of pre-trained language models, with parameters ranging from 1 billion to 15 billion, specifically tailored for the text-to-SQL task. It employs an incremental pre-training approach using a meticulously curated SQL-centric corpus.
- **DIN-SQL** (Pourreza and Rafiei, 2024a) incorporates classification and decomposition modules. It classifies each query into one of three categories, subsequently applying distinct strategies to process each group effectively.
- **CHESS** (Talaei et al., 2024) decomposes the text-to-SQL task into a three-stage pipeline, comprising entity and context retrieval, schema selection, and query generation. It achieves a performance of 67.86% on the BIRD dataset.
- **DAIL-SQL** (Gao et al., 2024) introduces an innovative prompt strategy, designed from the perspectives of question representation, example selection, and example organization, achieving a performance of 86.6% on the Spider 1.0 dataset (Yu et al., 2018).
- **Spider-Agent** (Lei et al., 2024) is the first agent framework implemented on the enterprise-level text-to-SQL dataset, Spider 2.0. It is developed based on the ReAct (Yao et al., 2023) framework, with a primary focus on database-related coding tasks and projects.

Our Setups. To ensure fairness, we utilize GPT-



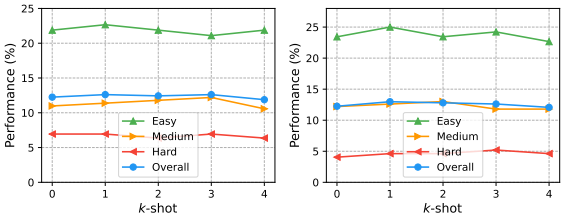
(a) Spider-Agent on Spider 2.0-snow (b) SoC-Agent on Spider 2.0-snow (c) Spider-Agent on Spider 2.0-lite (d) SoC-Agent on Spider 2.0-lite

Figure 4: Step distribution.



(a) Spider 2.0-snow (b) Spider 2.0-lite

Figure 5: Finish rate.



(a) Spider 2.0-snow (b) Spider 2.0-lite

Figure 6: k-shot examples.

4o (2024-08-06) for all methods. The value of N is determined dynamically by the model based on the problem's complexity, ranging from 2 to 4. For our demonstration prompt, we employ a 1-shot approach. The maximum number of execution steps is set to 20, meaning the process terminates if the agent exceeds this limit.

4.2 Overall Performance (RQ1)

LLM-based Methods. The experimental results in Table 2 indicate that existing LLM-based methods struggle with these enterprise-level text-to-SQL tasks. SFT CodeS-15B and DIN-SQL fail to complete any tasks on the Spider 2.0-snow dataset. On the hard version of Spider 2.0-snow, none of the four LLM-based methods succeed. The state-of-the-art text-to-SQL method, DAIL, achieves only 2.20% and 5.68% performance on the snow and lite datasets, respectively, highlighting the need for an agent-based framework.

Agent-based Methods. Spider-Agent, a relatively simple agent-based framework, demonstrates significant performance improvements. Compared to the strongest LLM-based method, DAIL-SQL, Spider-Agent shows improvements of 8.59% and 5.29% on the Spider 2.0-snow and Spider 2.0-lite datasets, respectively. Our proposed method, SoC-Agent, further enhances performance, with improvements of 1.82% and 2.01% over Spider-Agent. These results validate the effectiveness of our SoC planning approach.

Table 3: Results of the ablation studies.

Method	Spider 2.0-snow			
	Easy	Medium	Hard	Overall
SoC-Agent	22.66%	11.38%	6.94%	12.61%
w/o SoC	20.31%	10.16%	5.20%	10.97%
w/o De	21.88%	10.98%	6.94%	12.25%
w/o Self	21.09%	10.98%	6.36%	11.88%

4.3 In-depth Analysis (RQ2)

Running Step. The running step refers to the number of steps the agent needs to execute to generate SQL results. We plot the distribution of steps executed by Spider-Agent and our method on the Spider 2.0-snow and Spider 2.0-lite dataset. Figure 4 illustrates these results, where blue bars represent the frequency of each step in completed tasks, and black bars indicate the frequency of tasks not completed after exceeding the maximum of 20 steps. The red dotted line shows the average number of steps in completed tasks. Our method requires a slightly higher average number of steps compared to Spider-Agent, likely because our SoC planning involves more steps to solve subtasks.

Finish Rate. The finish rate measures the proportion of tasks where the agent outputs the final SQL execution result, divided by the total number of tasks. Although the final result may not always be correct, the finish rate indicates that the agent completes the entire task process without issues such as interruptions, unexpected jumps, or exceeding the maximum step limit. As shown in Figure 5, our

Task: What is the highest number of electric bike rides lasting more than 10 minutes taken by subscribers with 'Student Membership' in a single day, excluding rides starting or ending at 'Mobile Station' or 'Repair Shop'?

Observation: …… (previous schema linking process) ……

Thought: I now have the necessary information about the BIKESHARE_TRIPS table. To solve the task, I will break it down into simpler sub-tasks: **T1:** Query all electric bike rides lasting more than 10 minutes taken by subscribers with 'Student Membership'. **T2:** Exclude rides that start or end at 'Mobile Station' or 'Repair Shop'. **T3:** Calculate the number of such rides for each day. **T4:** Find the highest number of rides in a single day. I will start by writing the SQL query for **T1**.
Action: `SNOWFLAKE_EXEC_SQL(sql_query= ,is_save=False)`

```
SELECT start_time
FROM AUSTIN.AUSTIN_BIKESHARE.BIKESHARE_TRIPS
WHERE subscriber_type = 'Student Membership'
AND bike_type = 'electric'
AND duration_minutes > 10
```

Observation: …… (returned results from database) ……

Thought: The result for **T1** is correct, as it returns the start times of electric bike rides lasting more than 10 minutes by subscribers with 'Student Membership'. Next, I will address **T2** by excluding rides that start or end at 'Mobile Station' or 'Repair Shop'.
Action: `SNOWFLAKE_EXEC_SQL(sql_query= ,is_save=False)`

```
SELECT start_time
FROM AUSTIN.AUSTIN_BIKESHARE.BIKESHARE_TRIPS
WHERE subscriber_type = 'Student Membership'
AND bike_type = 'electric'
AND duration_minutes > 10
AND start_station_name
    NOT IN ('Mobile Station', 'Repair Shop')
AND end_station_name
    NOT IN ('Mobile Station', 'Repair Shop')
```

Observation: …… (returned results from database) ……

Thought: The result for **T2** is correct, as it excludes rides starting or ending at 'Mobile Station' or 'Repair Shop'. Now, I will address **T3** by calculating the number of such rides for each day. I will convert the "start_time" to a date and count the rides for each day.
Action: `SNOWFLAKE_EXEC_SQL(sql_query= ,is_save=False)`

```
SELECT
    TO_DATE(TO_TIMESTAMP(start_time/1000000)) AS ride_date,
    COUNT(*) AS ride_count
FROM AUSTIN.AUSTIN_BIKESHARE.BIKESHARE_TRIPS
WHERE subscriber_type = 'Student Membership'
AND bike_type = 'electric'
AND duration_minutes > 10
AND start_station_name
    NOT IN ('Mobile Station', 'Repair Shop')
AND end_station_name
    NOT IN ('Mobile Station', 'Repair Shop')
GROUP BY ride_date
```

Observation: …… (returned results from database) ……

Thought: The result for **T3** is correct, as it provides the count of rides for each day. Now, I will address **T4** by finding the highest number of rides in a single day from the results obtained in **T3**.
Action: `SNOWFLAKE_EXEC_SQL(sql_query= ,is_save=True)`

```
SELECT MAX(ride_count) AS max_ride_count
FROM (
    SELECT TO_DATE(TO_TIMESTAMP("start_time" / 1000000))
    AS ride_date,
    COUNT(*) AS ride_count
    FROM AUSTIN.AUSTIN_BIKESHARE.BIKESHARE_TRIPS
    WHERE subscriber_type = 'Student Membership'
    AND bike_type = 'electric'
    AND duration_minutes > 10
    AND start_station_name
        NOT IN ('Mobile Station', 'Repair Shop')
    AND end_station_name
        NOT IN ('Mobile Station', 'Repair Shop')
    GROUP BY ride_date
)
```

Observation: Results saved to /workspace/result.csv

Thought: The SQL query executed successfully, and the results are saved in "result.csv". I will now display the contents of this CSV file to verify the result.
Action:

```
cat /workspace/result.csv
```

Observation:
```
MAX_RIDE_COUNT
201
```

Thought: The result appears correct, showing the highest number of electric bike rides lasting more than 10 minutes taken by subscribers with 'Student Membership' in a single day, excluding rides starting or ending at 'Mobile Station' or 'Repair Shop'. The maximum number of such rides is 201.
Action:

```
Terminate(output="result.csv")
```

Figure 7: Case study of SoC-Agent in Spider 2.0-snow dataset.

method achieves a significantly higher completion rate than Spider-Agent across all task difficulties.

Ablation Study. We conduct a series of ablation studies to assess the impact of different modules in our method on overall performance. As shown in Table 3, "SoC-Agent" represents the original method, "w/o SoC" represents the removal of SoC planning, "w/o De" denotes the removal of demonstration, and "w/o Self" indicates the removal of self-reflection. The results clearly demonstrate a slight performance drop upon the removal of these modules, underscoring their necessity. In particular, the performance drops more significantly when SoC planning is removed, and the performance on hard tasks also drops significantly, highlighting the critical role of SoC planning in solving complex SQL generation tasks. Additionally, we experiment with multiple-shot examples in our demonstration module, as depicted in Figure 6. While the presence of examples proves effective, an excessive number does not yield significant performance improvements.

4.4 Case Study (RQ3)

To intuitively verify the workflow of the proposed SoC-Agent, we sample a case from Spider 2.0-snow dataset and demonstrate the complete workflow of the agent, as shown in Figure 7. For a given task, the agent first collects and retrieves relevant information along the current working path, which we omit here for brevity. After information collection, the agent decomposes the original problem from simple to complex, splitting it into four sub-problems in this example. Specifically, the original task $\mathcal{T}$ is "What is the highest number

of electric bike rides lasting more than 10 minutes taken by subscribers with 'Student Membership' in a single day, excluding rides starting or ending at 'Mobile Station' or 'Repair Shop'?". According to the decomposition results of the agent, we can clearly see that $\mathcal{T}_1$ is the simplest query for tram riding records, and the complexity of $\mathcal{T}_2$, $\mathcal{T}_3$, and $\mathcal{T}_4$ gradually increases. Starting with the simplest, the agent writes a 5-line SQL query to successfully solve $\mathcal{T}_1$. Building on the SQL for $\mathcal{T}_1$, additional constraints are added to solve $\mathcal{T}_2$. Subsequently, using the SQL from $\mathcal{T}_2$, the number of rides per day is calculated, successfully solving $\mathcal{T}_3$. Finally, based on the SQL from $\mathcal{T}_3$, the maximum number of rides is determined, completing the final task. This process of generating SQL from simple to complex aligns with our SoC planning workflow. Additionally, the agent reviews the problem and checks the generated results at the end, further ensuring the effectiveness and accuracy of SQL generation.

5 Related Work

LLMs in Text-to-SQL. LLMs have revolutionized text-to-SQL tasks (Shi et al., 2024; Liu et al., 2024b; Li et al., 2024a) through their exceptional reasoning capabilities and world knowledge integration. Early studies leverage the zero-shot capabilities (Chang and Fosler-Lussier, 2023; Liu et al., 2023; Rajkumar et al., 2022; Dong et al., 2023) of LLMs, enabling them to generate valid SQL queries without prior examples. Building on the principles of in-context learning (Wei et al., 2022), DAIL-SQL (Gao et al., 2024) utilizes problem-relevant examples to guide SQL generation. DIN-SQL (Pourreza and Rafiei, 2024a) introduces a decomposition approach that breaks down complex queries into manageable sub-problems. MAC-SQL (Wang et al., 2024a) presents a collaborative framework involving multiple agents to tackle the challenges of SQL generation. Studies (Nan et al., 2023; Luo et al., 2024; Pourreza et al., 2024; Caferoğlu and Ulusoy, 2024; Qu et al., 2024; Mao et al., 2024) also explore various advanced prompting techniques for text-to-SQL. Beyond prompt design, some studies emphasize the importance of schema linking. For instance, CHESS (Talaei et al., 2024) and PURPLE (Ren et al., 2024) focus on improving SQL generation capabilities through retrieval mechanisms and schema pruning strategies. Conversely, study (Maamari et al., 2024) offers a critical perspective, arguing that the loss of essential information during schema linking may adversely affect the accuracy of SQL generated by LLMs. In addition to prompt-based approaches, other studies investigate supervised fine-tuning (SFT) methods applied to open-source LLMs. For example, DB-GPT-Hub (Zhou et al., 2024) examines the influence of different SFT strategies on the performance of open-source LLMs, while SENSE (Yang et al., 2024a) introduces an innovative data synthesis technique that enables open-source LLMs to outperform their closed-source counterparts for the first time.

LLM-Powered Agents. LLMs have become a key technology in the quest for Artificial General Intelligence (AGI), providing strong support for developing intelligent agent systems (Wang et al., 2024b; Xi et al., 2023; Park et al., 2023; Liu et al., 2024a; Huang et al., 2024a). Existing efforts primarily focus on agent planning (Yao et al., 2023; Chen et al., 2023; Song et al., 2023) and using external tools (Qin et al., 2024b; Qiao et al., 2024; Qin et al., 2024a; Qu et al., 2025). Recently, LLM-powered agents are being used to automate the generation of code (Zhang et al., 2023; Zhu et al., 2024a; Yin et al., 2023; Huang et al., 2024b; Chen et al., 2021, 2024; Xia et al., 2024), which can make the development process faster and reduce the need for human programmers. Despite this, agent-based methods are rarely explored in text-to-SQL (Deng et al., 2025; Wang et al., 2024c). MAC-SQL (Wang et al., 2024a) first proposes the concept of using agents to solve text-to-SQL tasks but lacks the process of interacting with complex environments in real scenarios. Spider-Agent (Lei et al., 2024), based on the ReAct (Yao et al., 2023) and Intercode (Yang et al., 2024b) frameworks, for the first time runs the agent in a real-world SQL development environment, establishing a strong baseline approach.

6 Conclusion

In this paper, we propose a novel agent framework, SoC-Agent, to address the task of text-to-SQL in real-world development scenarios. Specifically, for a given task, SoC-Agent decomposes it into a series of subtasks ranging from simple to complex, solving them sequentially based on task complexity. The final self-reflection module ensures the effectiveness of the generated results. We conduct extensive experiments on real-world enterprise-level SQL benchmarks, and the results demonstrate the effectiveness of our proposed method.

Limitations

This paper has several key limitations that warrant attention. Firstly, due to the cost constraints, we do not use more advanced models for experiments, such as OpenAI o1 or o3. Secondly, while our method outperforms existing state-of-the-art methods in real-world text-to-SQL tasks, the improvement is limited. Therefore, there is significant room for performance enhancement in the design of more advanced agent workflows, such as incorporating more advanced planning strategies, better memory management techniques, and improved tools. We will consider these limitations as research directions for our future work.

Ethics Statement

We affirm that this study adheres to the ethical guidelines set forth by the relevant academic and research institutions. The datasets utilized in our research are publicly accessible and have been widely adopted in the field of text-to-SQL research. This ensures that our work is transparent and that our results can be reproduced by other researchers. Additionally, the outputs of our study are in the form of SQL queries, which are less likely to contain harmful or biased content compared to natural language text. Our team conducts thorough reviews of all outputs to ensure they do not contain any politically sensitive or biased information.

References

Hasan Alp Caferoğlu and Özgür Ulusoy. 2024. E-sql: Direct schema linking via question enrichment in text-to-sql. *arXiv preprint arXiv:2409.16751*.

Shuaichen Chang and Eric Fosler-Lussier. 2023. How to prompt llms for text-to-sql: A study in zero-shot, single-domain, and cross-domain settings. In *NeurIPS 2023 Second Table Representation Learning Workshop*.

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

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F Karlsson, Jie Fu, and Yemin Shi. 2023. Autoagents: A framework for automatic agent generation. *arXiv preprint arXiv:2309.17288*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. 2025. Reforce: A text-to-sql agent with self-refinement, format restriction, and column exploration. *arXiv preprint arXiv:2502.00675*.

Naihao Deng, Yulong Chen, and Yue Zhang. 2022. Recent advances in text-to-sql: A survey of what we have and what we expect. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187.

Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Jinshu Lin, Dongfang Lou, et al. 2023. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306*.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-sql empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment*, 17(5):1132–1145.

T Guo, X Chen, Y Wang, R Chang, S Pei, NV Chawla, O Wiest, and X Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. In *33rd International Joint Conference on Artificial Intelligence (IJCAI 2024)*. IJCAI; Cornell arxiv.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2024a. Mlagentbench: Evaluating language agents on machine learning experimentation. In *ICML*.

Quzhe Huang, Yanxi Zhang, and Dongyan Zhao. 2023. From simple to complex: A progressive framework for document-level informative argument extraction. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 6129–6140.

Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, et al. 2024b. Da-code: Agent data science code generation benchmark for large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 13487–13521.

Ayush Kumar, Parth Nagarkar, Prabhav Nalhe, and Sanjeev Vijayakumar. 2022. Deep learning driven natural languages text to sql query conversion: a survey. *arXiv preprint arXiv:2208.04415*.

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

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024a. The dawn of natural language to sql: Are we fully ready? *arXiv preprint arXiv:2406.01265*.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024b. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data*, 2(3):1–28.

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

Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S Yu. 2023. A comprehensive evaluation of chatgpt's zero-shot text-to-sql capability. *arXiv preprint arXiv:2303.13547*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. 2024a. Agentbench: Evaluating LLMs as agents. In *ICLR*.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2024b. A survey of nl2sql with large language models: Where are we, and where are we going? *arXiv preprint arXiv:2408.05109*.

Ruilin Luo, Liyuan Wang, Binghuai Lin, Zicheng Lin, and Yujiu Yang. 2024. Ptd-sql: Partitioning and targeted drilling with llms in text-to-sql. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3767–3799.

Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. 2024. The death of schema linking? text-to-sql in the age of well-reasoned language models. In *NeurIPS 2024 Third Table Representation Learning Workshop*.

Wenxin Mao, Ruiqi Wang, Jiyu Guo, Jichuan Zeng, Cuiyun Gao, Peiyi Han, and Chuanyi Liu. 2024. Enhancing text-to-sql parsing through question rewriting and execution-guided refinement. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 2009–2024.

Ali Mohammadjafari, Anthony S Maida, and Raju Gottumukkala. 2024. From natural language to sql: Review of llm-based text-to-sql systems. *arXiv preprint arXiv:2410.01066*.

Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. 2023. Enhancing text-to-sql capabilities of large language models: A study on prompt design strategies. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 14935–14956.

Jiayi Pan, Yichi Zhang, Nicholas Tomlin, Yifei Zhou, Sergey Levine, and Alane Suhr. 2024. Autonomous evaluation and refinement of digital agents. *arXiv preprint arXiv:2404.06474*.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2024. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943*.

Mohammadreza Pourreza and Davood Rafiei. 2024a. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *NeurIPS*, 36.

Mohammadreza Pourreza and Davood Rafiei. 2024b. Dts-sql: Decomposed text-to-sql with small large language models. *arXiv preprint arXiv:2402.01117*.

Shuofei Qiao, Honghao Gui, Chengfei Lv, Qianghuai Jia, Huajun Chen, and Ningyu Zhang. 2024. Making language models better tool learners with execution feedback. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3550–3568.

Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, et al. 2022. A survey on text-to-sql parsing: Concepts, methods, and future directions. *arXiv preprint arXiv:2208.13629*.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, et al. 2024a. Tool learning with foundation models. *ACM Computing Surveys*, 57(4):1–40.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024b. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *ICLR*.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343.

Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. Before generation, align it! A novel and effective strategy for mitigating hallucinations in text-to-sql generation. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 5456–5471.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-sql capabilities of large language models. *arXiv preprint arXiv:2204.00498*.

Tonghui Ren, Yuankai Fan, Zhenying He, Ren Huang, Jiaqi Dai, Can Huang, Yinan Jing, Kai Zhang, Yifan Yang, and X. Sean Wang. 2024. PURPLE: making a large language model a better SQL writer. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*, pages 15–28. IEEE.

Liang Shi, Zhengju Tang, Nan Zhang, Xiaotong Zhang, and Zhi Yang. 2024. A survey on employing large language models for text-to-sql tasks. *arXiv preprint arXiv:2407.15186*.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2998–3009.

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

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, et al. 2024a. Mac-sql: A multi-agent collaborative framework for text-to-sql. *arXiv preprint arXiv:2312.11242*.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024b. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *ICLR*.

Zhongyuan Wang, Richong Zhang, Zhijie Nie, and Jaein Kim. 2024c. Tool-assisted agent on sql inspection and refinement in real-world scenarios. *arXiv preprint arXiv:2408.16991*.

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

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*.

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

Jiaxi Yang, Binyuan Hui, Min Yang, Jian Yang, Junyang Lin, and Chang Zhou. 2024a. Synthesizing text-to-sql data from weak and strong llms. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7864–7875.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2024b. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *NeurIPS*, 36.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *ICLR*.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek K Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, et al. 2023. Natural language to code generation in interactive data science notebooks. In *The 61st Annual Meeting Of The Association For Computational Linguistics*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. Planning with large language models for code generation. In *ICLR*.

Fan Zhou, Siqiao Xue, Danrui Qi, Wenhui Shi, Wang Zhao, Ganglin Wei, Hongyang Zhang, Caigai Jiang, Gangwei Jiang, Zhixuan Chu, et al. 2024. Db-gpt-hub: Towards open benchmarking text-to-sql empowered by large language models. *arXiv preprint arXiv:2406.11434*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024a. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Xiaohu Zhu, Qian Li, Lizhen Cui, and Yongkang Liu. 2024b. Large language model enhanced text-to-sql generation: A survey. *arXiv preprint arXiv:2410.06011*.

A Prompts

A.1 Task Description

Task Description

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! Carefully review the markdown content below, as it contains the information you need to successfully complete the task.

ACTION SPACE
`{action_space}`

Figure 8: The prompt of task description

A.2 Schema Inspection and SQL Task Guide

File Checking and Information Gathering

- You are in the `/workspace` directory. Begin by checking if there are any markdown files in this directory (e.g. `ls -R`). If found, read them as they may contain useful information for answering your questions.
- 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.
- Do not write SQL queries to retrieve the schema; use the existing schema documents in the folders.

Figure 9: The prompt of schema inspection and SQL task guide

Snowflake-Query Execution Rules

- 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.
- Focus on SQL queries rather than frequently using Bash commands like grep and cat, though they can be used when necessary.
- 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`.
- Column names must be enclosed in quotes.
- 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.

Figure 10: The prompt of Snowflake-Query execution rules

A.3 Gradual SQL Query Development: From Simple to Complex

881

Workflow Description of SoC Planning

Creating a SQL query that works perfectly on the first try can be difficult. To enhance accuracy, please simplify the problem step by step before writing the final SQL query. Begin by addressing a basic version of the task, then progressively enhance your SQL query to tackle more complex versions, ultimately solving the original task. The process is outlined below:

- **Decompose the task**: Break down the original task $\mathcal{T}$ into N versions: $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_N$, where the complexity increases with each index. Here: $\mathcal{T}_1$ is the simplest version; $\mathcal{T}_2$ builds on $\mathcal{T}_1$ with added functionality; ...; $\mathcal{T}_N$ is the most complex version, equivalent to the original task $\mathcal{T}$. The value of N should generally be between 1 and 5, depending on the task's complexity.
- **Complete tasks from simple to complex**: First, write SQL query $\mathcal{S}_1$ to accomplish task $\mathcal{T}_1$. Next, create SQL $\mathcal{S}_2$ to complete task $\mathcal{T}_2$, building upon $\mathcal{S}_1$. Continue this process until you write SQL statement $\mathcal{S}_{(N-1)}$ for the penultimate task. Finally, write the SQL $\mathcal{S}_N$ to complete the final task T based on $\mathcal{S}_{(N-1)}$.

Figure 11: The prompt of workflow description of SoC planning

Pseudocode Description of SoC Planning

1. Start with an initial empty SQL $\mathcal{S}_0 \leftarrow \phi$.
2. Decompose the problem T into n versions: $[\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_N]$, where $\mathcal{T}_N = \mathcal{T}$.
3. **For** $i = 1$ to N **do**
   3.1. Based on the previous SQL $\mathcal{S}_{i-1}$, write a new SQL $\mathcal{S}_i$ to address $\mathcal{T}_i$.
   3.2. Execute $\mathcal{S}_i$ in the database system to obtain the result $\mathcal{R}_i$ (set `is_save=False`).
   3.3. **If** $\mathcal{R}_i$ does not solve $\mathcal{T}_i$ **then**
      3.3.1. Continuously modify and execute $\mathcal{S}_i$ until it satisfactorily addresses $\mathcal{T}_i$.
4. **End For**

Figure 12: The prompt of pseudocode description of SoC planning

A.4 Self-Reflection

Task Recall and Result Verification

Before terminating the task, you MUST recall the original task thoroughly. This review is essential to ensure that the generated results meet the specified criteria and that nothing is overlooked.

Task Recall
After generating the final result (e.g., producing "result.csv"), you MUST first recall the original task ($\mathcal{T}$). Make sure you fully understand what was requested.

Final Result Inspection
Evaluate the output to confirm it meets the task requirements. If not, adjust the SQL query. Consider the following:

- **Output Format**: 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 final output as a CSV and provide the file name, typically derived from the SQL execution result. Do not create the file to save results; use `SNOWFLAKE_EXEC_SQL` directly and just set "`is_save=True`" to generate the result file.
- **Data Validity**: Ensure the CSV is not empty and contains valid data. An empty file or one with only headers indicates an incorrect SQL query.
- **Sample Size Limitation**: Pay attention to whether the task specifies extracting the "most X," "top X," or "first X" entities. If so, include "LIMIT X" in your SQL query to restrict the result set accordingly.
- **Field Completeness**: Always return both the entity ID and the entity name for any identified players or entities, along with any other relevant details.
- **Unit Conversion**: Verify that results adhere to specified units (e.g., hours, meters) from the original task. Perform necessary conversions for accuracy.
- **Confidence in Results**: If the result is obtained from your query or calculation and you believe your SQL (or Python) is correct, do not doubt the result based on objective data; avoid adding unnecessary interpretations. For example, if the average value through correct SQL is 0, do not assume that the result should be a non-zero value.

Figure 13: The prompt of task recall and result verification

A.5 Response Format and Example

Response Format and Example

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: ").

Example
Task: List employees with salaries above the average level in their respective departments.
Observation: ... (Output from previous actions, as provided by the environment; no need to generate it).
Thought: For the original task $\mathcal{T}$, we can break it down into the following three sub-tasks from easy to difficult: $\mathcal{T}_1$: "Query the basic information and salaries of all employees in 2023". $\mathcal{T}_2$: "Query the average salary of employees for each department in 2023". $\mathcal{T}_3$: "Query the employees with salaries above the average salary in their respective departments in 2023, including id, name, department, salary". Firstly, I will write and execute SQL $\mathcal{S}_1$ to check if it can solve $\mathcal{T}_1$.
Action: `SNOWFLAKE_EXEC_SQL(sql_query="SELECT employee_id, employee_name, department, salary FROM employees WHERE year = 2023", is_save=False)`
Observation: ... (Output from previous actions).
Thought: The result seems correct. The next task $\mathcal{T}_2$ is: "Query the average salary of employees for each department in 2023". I will write SQL $\mathcal{S}_2$ based on $\mathcal{S}_1$.
Action: `SNOWFLAKE_EXEC_SQL(sql_query="SELECT department, AVG(salary) AS avg_salary FROM employees WHERE year = 2023 GROUP BY department", is_save=False)`
Observation: ... (Output from previous actions).
Thought: The result seems correct. The result seems correct. The next task $\mathcal{T}_3$ is: "Query the employees with salaries above the average salary in their respective departments in 2023, including id, name, department, salary". I will write SQL $\mathcal{S}_3$ based on $\mathcal{S}_2$.
Action: `SNOWFLAKE_EXEC_SQL(sql_query="WITH department_avg AS (SELECT department, AVG(salary) AS avg_salary FROM employees WHERE year = 2023 GROUP BY department) SELECT e.employee_id, e.employee_name, e.department, e.salary FROM employees e JOIN department_avg da ON e.department = da.department WHERE e.salary > da.avg_salary AND e.year = 2023", is_save=True)`
Observation: Results saved to `/workspace/result.csv`
Thought: The SQL query executed successfully, and results are saved. I MUST recall the original task T, it is "List employees with salaries above the average level in their respective departments.". Then, I will display the contents of this CSV file to verify the result.
Action: `Bash(code="cat /workspace/result.csv")`
Observation: ... (Output from previous actions).
Thought: The result appears correct, containing all required information, including employee ID, name, department and salary.
Action: `Terminate(output="result.csv")`

Figure 14: The prompt of response format and example