A Plug-and-Play Natural Language Rewriter for Natural Language to SQL

Peixian Ma[1], **Boyan Li**[1], **Runzhi Jiang**[1], **Ju Fan**[2], **Nan Tang**[1], **Yuyu Luo**[1*]

[1]The Hong Kong University of Science and Technology (Guangzhou)
[2] Renmin University of China
{pma929, rjiang073@connect.hkust-gz.edu.cn, boyanli, nantang, yuyuluo}@hkust-gz.edu.cn, fanj@ruc.edu.cn

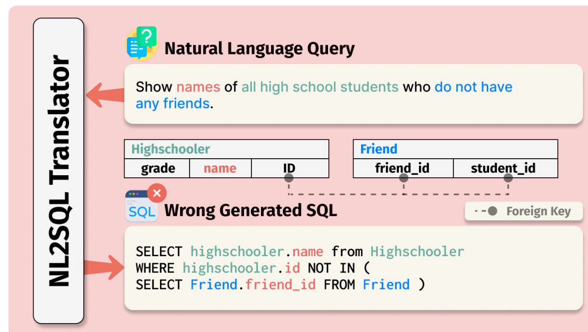
Abstract

Existing Natural Language to *SQL* (*NL2SQL*) solutions have made significant advancements, yet challenges persist in interpreting and translating *NL* queries, primarily due to users' limited understanding of database schemas or memory biases toward specific table or column values. These challenges often result in incorrect *NL2SQL* translations. To address these issues, we propose REWRITER, a plug-and-play module designed to enhance *NL2SQL* systems by automatically rewriting ambiguous or flawed *NL* queries. By incorporating database knowledge and content (e.g., column values and foreign keys), REWRITER reduces errors caused by flawed *NL* inputs and improves *SQL* generation accuracy. Our REWRITER treats *NL2SQL* models as *black boxes*, ensuring compatibility with various *NL2SQL* methods, including agent-based and rule-based *NL2SQL* solutions. REWRITER comprises three key components: `Checker`, `Reflector`, and `Rewriter`. The `Checker` identifies flawed *NL* queries by assessing the correctness of the generated *SQL*, minimizing unnecessary rewriting and potential hallucinations. The `Reflector` analyzes and accumulates experience to identify issues in *NL* queries, while the `Rewriter` revises the queries based on `Reflector`'s feedback. Extensive experiments on the Spider and BIRD benchmarks demonstrate that REWRITER consistently enhances downstream models, achieving average improvements of 1.6% and 2.0% in execution accuracy, respectively.

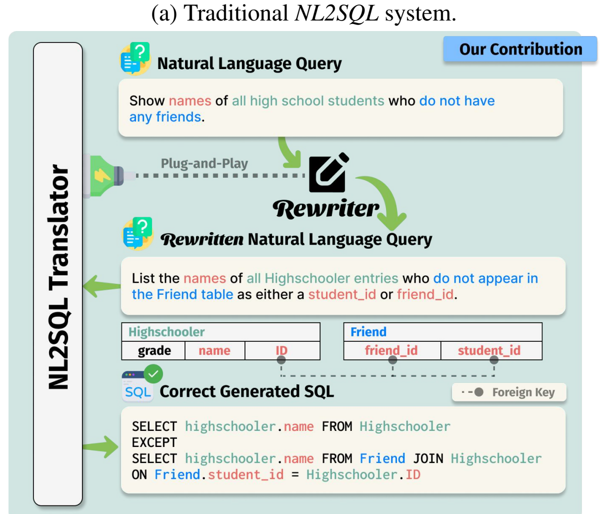
Introduction

Natural Language to *SQL* (*NL2SQL*) allows users to convert natural language (*NL*) queries into structured *SQL* statements, simplifying database interaction without requiring *SQL* expertise (Li et al. 2024a; Liu et al. 2024), and supports a wide range of data science tasks (Luo et al. 2018a, 2020, 2022a). Recent advancements in pre-trained and large language models (PLMs and LLMs) have significantly enhanced the semantic parsing capabilities of *NL2SQL* systems, improving the integration of *NL* and database content (*DB*) (Katsogiannis-Meimarakis and Koutrika 2023). Recently, research efforts have primarily focused on optimizing various components of the *NL2SQL* pipeline, including schema linking (Li et al. 2023a), database representation (Talaei et al. 2024), encoding and decoding strategies (Fu et al. 2023; Wang et al. 2019; Li et al. 2024b), and post-processing techniques (Pourreza and Rafiei 2023; Wang et al. 2024; Dong et al. 2023).

Despite these advancements, the quality of input *NL* queries remains an underexplored challenge. In practice,



(a) Traditional *NL2SQL* system.



(b) Deployment of a plug-and-play module to rewrite user *NL* in above *NL2SQL* system.

Figure 1: Demonstration of our work. The proposed plug-and-play REWRITER clearly indicates the relationships of foreign keys between tables in the rewritten *NL*, thereby avoiding the generation of incorrect *SQL* statements due to the gap between unclear user intent and the *DB*.

*Yuyu Luo is the corresponding author.

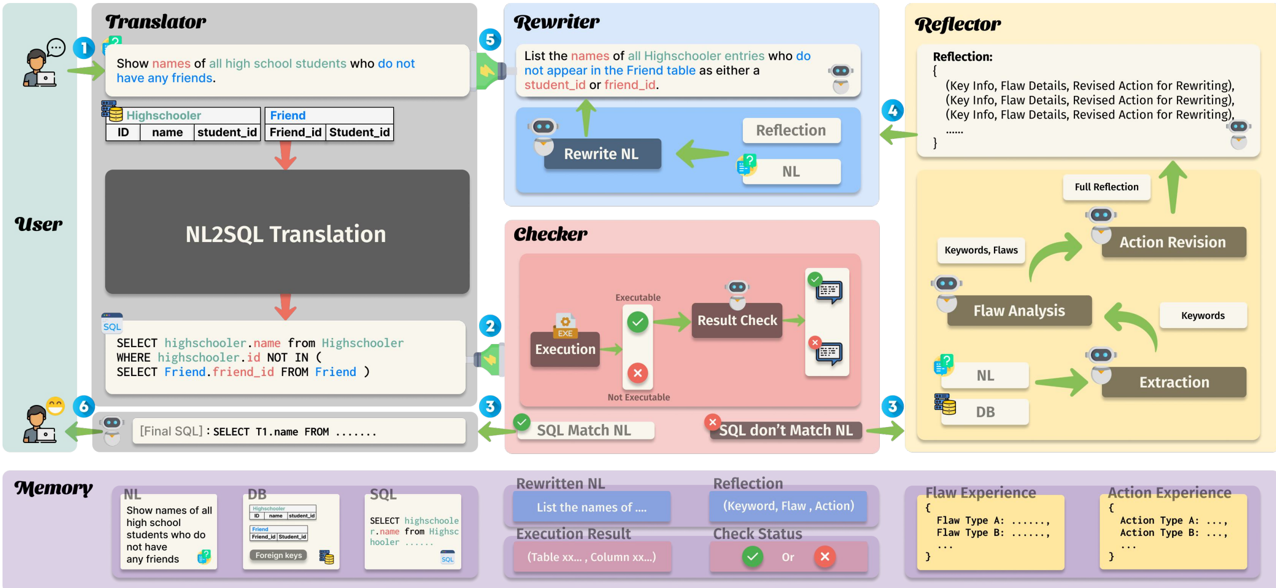


Figure 2: An overview of the proposed REWRITER framework, which comprises the following components: (i) `Checker`, which determines whether *NL* matches the generated *SQL*; (ii) `Reflector`, which analyzes the flawed *NL* and gives the rewriting reflection with the reference of *DB*; (iii) `Rewriter`, which rewrites the flawed *NL* under the guidance of reflection. In addition, a task-specific `Memory` module provides information exchange and storage services for these agents.

users often lack sufficient knowledge of database schemas or struggle to express complex queries, particularly in multi-table scenarios. As depicted in Figure 1a, this can result in ambiguous, incomplete, or flawed *NL* queries, creating a significant gap between user intent and accurate *SQL* generation.

To tackle this issue, we propose REWRITER, a plug-and-play framework designed to refine flawed *NL* queries and enhance the overall *NL2SQL* process. As shown in Figure 1b, REWRITER leverages database knowledge (e.g., column values and foreign keys) to rewrite ambiguous or incomplete *NL* queries, improving the performance of downstream *NL2SQL* models. Our REWRITER treats *NL2SQL* models as *black boxes*, ensuring compatibility with diverse *NL2SQL* solutions, including agent-based and LLM-based approaches.

However, our study presents several research challenges.

(C1) Lack of Training Data. Real-world datasets with flawed *NL* queries and their corrected versions are scarce, making it difficult to directly train rewriting models.

(C2) Adaptability to Various NL2SQL Solutions. The varying capabilities of different *NL2SQL* models pose a significant challenge. Rewriting strategies must be flexible enough to accommodate diverse systems, ranging from rule-based approaches to advanced LLM-based solutions, while ensuring consistency and robustness across varied user inputs and database schemas.

(C3) Minimizing Negative Impact. Rewriting flawed *NL* is a double-edged sword; while it can improve query clarity, it may also inadvertently degrade the performance of downstream *NL2SQL* models. Therefore, it is crucial to carefully balance the benefits of rewriting with the risks of introducing errors or misinterpretations.

To address these challenges, we propose a multi-agent framework named REWRITER, which integrates three key components, i.e., `Reflector`, `Checker`, and `Rewriter`, supported by a shared `Memory` module that facilitates experience accumulation and exchange (see Figure 2).

To address the first challenge *(C1)*, our `Reflector` employs a self-reflection mechanism that iteratively learns from its own trial-and-error process without relying on extensive labeled datasets. By analyzing *NL* flaws and accumulating rewriting experiences, it extracts key details, identifies errors, and generates rewriting actions. It organizes this information into actionable reflections (e.g., keyword mismatches, missing information, or ambiguity) to guide precise revisions.

To address the second challenge *(C2)*, the `Rewriter` focuses on targeted revisions informed by feedback from the `Reflector`, enhancing query clarity while maintaining the user's original intent. In parallel, the `Checker` serves as a versatile evaluation module, ensuring alignment between *NL* queries and generated *SQL* across a wide range of *NL2SQL* systems, from rule-based to LLM-based approaches.

To minimize negative impact *(C3)*, the `Checker` ensures that the rewriting process targets only flawed queries, avoiding unnecessary modifications to correct *NL*. By executing the generated *SQL* on the database and comparing results to the user's intent, it classifies queries as either valid (`SQL Match NL`) or flawed (`SQL Don't Match NL`). This selective approach reduces the risk of over-rewriting, hallucinations, and errors.

Contributions. We make the following contributions.

- **Plug-and-Play Framework.** We present REWRITER, a novel plug-and-play framework that identifies and rewrites ambiguous or flawed *NL* queries in *NL2SQL* systems. The framework integrates seamlessly with diverse *NL2SQL* methodologies, including rule-based and LLM-based solutions.

- **Self-Reflective Learning.** REWRITER incorporates a self-reflection mechanism to iteratively refine rewriting actions without relying on large-scale training data. Its conservative rewriting strategies ensure improved query clarity while minimizing potential negative impacts on downstream models.
- **Comprehensive Evaluation.** Through extensive experiments on multiple benchmarks, we demonstrate that REWRITER enhances the performance of various *NL2SQL* models. Our results highlight the framework's adaptability and ability to learn effectively from experience.

Methodology

Overview

In this section, we propose REWRITER, a plug-and-play multi-agent framework designed to optimize user *NL* in *NL2SQL* task. As illustrated in Figure 2, REWRITER consists of the following LLM-based agents: the `Checker` for discriminating samples that *NL* does not match *SQL*, the `Reflector` for analyzing the flawed *NL* and developing rewrite actions, and the `Rewriter` for rewriting flawed *NL* under the guidance of `Reflector`. Additionally, the `Memory` component stores key information throughout the rewriting process and facilitates communication among the agents.

Checker

`Checker` plugs the output port of the *NL2SQL* model. The primary objective of it is to identify the samples that the generated *SQL* query does not match the user's *NL*. As demonstrated in Figure 2, the workflow of `Checker` can be delineated into two distinct phases. In the first phase, `Checker` executes the generated *SQL* queries in the database and filters out those that have syntax errors or cannot be executed correctly. These samples will be marked as `NL DO NOT MATCH SQL`. If the generated SQL query executes successfully, it will be included in the instructions for the second stage of the Checker agent. This will also comprise the related *NL* statement, *DB*, and the corresponding query result. These rules focus not only on checking *SQL* syntax but also on identifying potential semantic errors, reference errors, and other issues among the *NL*, *DB*, *SQL*, and query results. In the second stage, the `Checker` agent assesses whether the query result aligns with the intent of the provided *NL* based on the above information and professional verification rules. Samples that pass both stages of verification will be marked as `NL MATCH SQL` while those that do not will be labeled as `NL DO NOT MATCH SQL` and stored accordingly.

Reflector

`Reflector` works as the core of the REWRITER framework. In the process of reasoning, it firstly extracts the key information (e.g. keywords or description of related tables and columns) in given *NL* and compares the above information with *DB*. Then, it verifies the key information with the provided *DB* (e.g., Table names and column names, foreign keys)

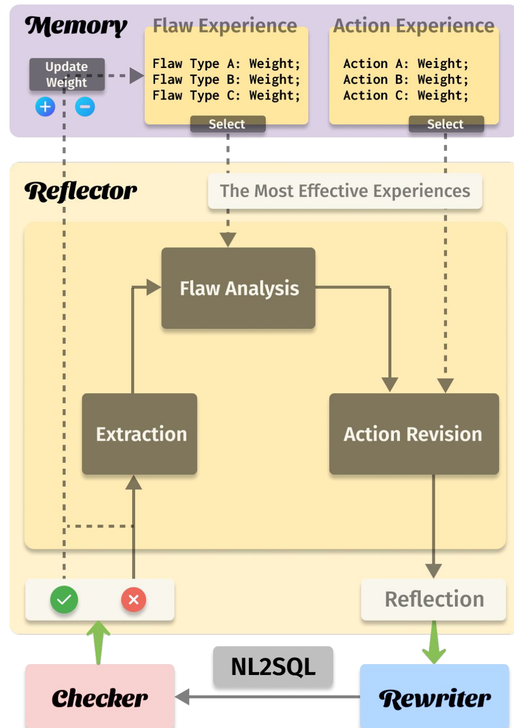


Figure 3: Demonstration of self-reflection mechanism. In the rewriting process, the experiences with the highest weights will be loaded into the rules. Subsequently, the `Reflector` updates the weights of the experience, which is applied in the detailed reflection based on the checker's feedback on the new results.

and then analyzes the information for any potential flaws or mismatch of user's intent (e.g., Ambiguity or Missing information). For example, `Reflector` examines the foreign keys in the database to determine if there is an intention to query multiple tables and columns. Finally, specific rewriting suggestions for the above flaws are generated with reference to the above *DB* and action experience. The final reflection output is collated as a triples text (`Keyword`, `Flaw`, `Action`), where the `Flaw` represents the flawed analysis of the key information and the `Action` represents the related rewriting suggestions.

Due to the lack of sufficient labeled training data, the `Reflector` is unable to summarize practical flaw experience or action experience through training or fine-tuning. Consequently, as shown in Figure 3, we employ a self-reflection mechanism for updating the above experience. In the initialization stage, the `Reflector` is instructed to generate various potential flaws of the user's *NL* and related rewriting actions as the initialized experiences, which are condensed and not related to concrete database content or values. These experiences are stored in the `Memory` with associated weights corresponding to their validity and importance. During the process of flaw analysis and action revision, a batch of experiences with the highest weight is selected as the most likely defect and the most effective rewriting action in the current *NL* for the reference of `Reflector`. Subsequently, the `Reflector` utilizes these experiences and *DB* to provide concrete flaw descriptions and revision suggestions pertaining to database values and concatenate them as the final reflection text. Additionally, it marks the specific experiences used in the reflection. Before the next round of reasoning and rewriting, the `Reflector` will adjust the weights of the above experience based on the reward signal, which is characterized by a sparse binary state (`NL MATCH SQL` or `NL DO NOT MATCH SQL`) provided by the `Checker`. By continuously looping through the above process at reasoning, `Reflector` can accumulate some effective flaw experiences and action experiences, enabling it to adjust itself in time based on the distribution of flaw occurrences in different *NL*., enhancing the robustness of rewriting *NL*.

Rewriter

The task of `Rewriter` is to rewrite the specified flawed *NL* and plug the rewritten *NL* into the *NL2SQL* translator. It retrieves the reflection text provided by the `Reflector` and the related *DB* from the `Memory` and performs the rewrite action guided by that information and specific rules. In addition, `Rewriter` also seeks to maintain consistency of narrative style between the rewritten *NL* and the original *NL*, which can avoid creating new hallucinations and misunderstandings for the *NL2SQL* translation. The `Rewriter` will only attempt to modify the narrative style when the reflection explicitly states that a comprehensive modification of the expressive semantics is necessary.

Memory

The function of `Memory` is to store the data stream record of the above agents, such as *NL*, *DB*, *SQL*, or details of flaws and actions summarized by `Reflector`. `Memory` can continuously accumulate historical data from different *NL2SQL* methods during the workflow of REWRITER, and support the agents to adapt to different *NL* representation styles, thus avoiding dependence on a single rewrite strategy and improving the robustness of the rewritten *NL*.

Experiments

Settings

Datasets We evaluated the proposed REWRITER and related *NL2SQL* translation models on two benchmarks, Spider (Yu et al. 2018) and BIRD (Li et al. 2023b). Spider comprises 10,181 questions paired with 5,693 complex *SQL* queries from 200 databases and 138 domains. BIRD (Li et al. 2023b) comprises 12,751 *NL2SQL* pairs encompassing 95 databases from 37 specialized domains.

Metrics For fair comparisons, we follow the standard evaluation metric of each benchmark. For Spider, we use Execution Accuracy (EX) and Exact Match Accuracy (EM) as the metric. For BIRD, we utilize EX and Valid Efficiency Score (VES) as the metric. EX is used to estimate the percentage of questions that predict the same result for the query and the basic gold query across all query requests. EM focuses on measuring the exact match between model predictions and actual results. It treats each clause as a set and compares the prediction of each clause with the corresponding clause in the reference query. VES is a metric that evaluates the efficacy of *SQL* queries generated by the *NL2SQL* model. It specifically assesses the validity of the *SQL* queries by comparing their result sets with those of basic gold *SQL* queries. For the evaluation of the `Checker`, we utilized Check Precision (CP) as the metric. CP is defined as the precision of the bad samples that *NL* does not match the *SQL*.

Base Models To compare the efficiency of different base models on agent performance, we select several open-source

Table 1: *NL2SQL* translation in Spider-dev set.

NL2SQL **Translation Methods**	**CP**	**EX**	**EM**
C3 + GPT-3.5-turbo	-	81.9	46.9
+ REWRITER	55.8	**82.4**	**47.0**
DAIL-SQL + GPT-3.5-turbo	-	76.3	60.5
+ REWRITER	78.3	**78.1**	**61.2**
DAIL-SQL + GPT-4	-	83.1	70.0
+ REWRITER	60.5	**83.6**	**70.0**
DTS-SQL + Deepseek-6.7B	-	75.2	77.2
+ REWRITER	66.0	**77.5**	47.9
NatSQL + T5-Base	-	69.7	65.3
+ REWRITER	81.3	**72.2**	**68.5**
NatSQL + T5-3B	-	71.4	68.0
+ REWRITER	82.2	**75.7**	**71.9**
RESDSQL + T5-3B	-	81.8	78.1
+ REWRITER	57.0	**82.2**	**78.6**

Table 2: *NL2SQL* translation results in BIRD-dev set.

NL2SQL **Translation Methods**	**CP**	**EX**	**VES**
DAIL-SQL + GPT-4	-	54.3	56.1
+ REWRITER	53.0	**54.6**	56.0
GPT-4	-	46.4	49.8
+ REWRITER	58.4	**49.9**	**56.9**
GPT-3.5-Turbo	-	36.6	43.8
+ REWRITER	60.6	**38.9**	**44.0**

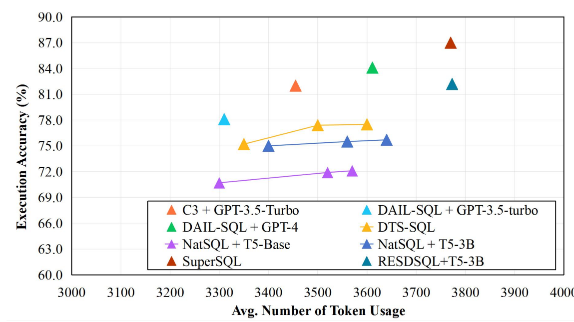


Figure 4: Execution accuracy and token efficiency of REWRITER in single-round and multi-round rewriting on Spider-dev set.

and closed-source LLM for comparison experiments, including GPT-4o, GPT-4 (OpenAI 2023), GLM-4 (GLM et al. 2024) and LLaMA-3-8B (Dubey et al. 2024).

NL2SQL Translator Baselines To comprehensively evaluate the performance of the REWRITER, we conduct experiments on both BIRD and Spider benchmark by plugging REWRITER on the following *NL2SQL* translator baselines: C3 (Dong et al. 2023), DTS-SQL (Pourreza and Rafiei 2024), NatSQL + T5 (Rai et al. 2023), RESDSQL (Li et al. 2023a) are specific to Spider dataset; GPT-3.5 & GPT-4 (OpenAI 2023) are specific to BIRD dataset; DAIL-SQL (Gao et al. 2023) is specific to both datasets.

Environments We conduct all experiments of REWRITER on a server with one NVIDIA A40 48GB GPU, one Intel(R) Xeon(R) Platinum 8358P CPU, 512 GB memory and Ubuntu 20.04.2 LTS operating system.

Overall Performance

SQL Generation Performance Table 1 and Table 2 illustrate the performance of the proposed REWRITER with various *NL2SQL* baseline methods in the dev set of Spider and BIRD benchmark. For most *SQL* generation results of the baselines, the `Checker` can achieve the CP of over 50% in terms of the bad samples. The *SQL* queries generated by the rewritten *NL* outperform those of the relevant baseline results in terms of execution accuracy, exact match accuracy, and valid efficiency score on both *NL2SQL* benchmarks. It is noticeable that most *SQL* generation results with REWRITER rewriting part of flawed *NL* are able to outperform the EX score of the baseline results by 1%-3%. In addition, we also explore the performance of multi-round rewriting. As demonstrated in Figure 4, we observe that multiple rounds of rewriting partial flawed *NL* can effectively improve the accuracy of *SQL* generation on some *NL2SQL* methods.

Token Efficiency For the agents in the REWRITER, we utilize different LLM as their base model. In light of the fact that the LLM APIs incur charges based on the token count and that the inference time of the LLM is directly correlated to the token length, we aim to minimize the computational cost associated with rewriting while upholding the quality of the rewritten *NL*. The total number of tokens is mainly influenced by the input prompt, including the representation of database content and the agent-specific rules. Our experiments evaluate the token efficiency of the `Checker` with different LLMs, which comprise close-source LLM and open-source LLM. To compute the cost of the agent, we repeat the experiment 5 times and calculate the average token usage of its response. Due to the lack of representative rewritten samples, all experiments are performed with zero-shot.

With the above settings, Figure 5 illustrates the performance and token efficiency of the `Checker` by employing different base models. In the comparison of closed-source LLMS, GPT-4o achieves more than 50% CP on the result of all *NL2SQL* baselines, and the token cost is close to that of GPT-4 and GLM-4. Although open-source LLMs demonstrate the potential to minimize token cost, their performance lags behind that of closed-source LLMs. The deficiency in available training data and the absence of domain knowledge may contribute to this performance disparity. In addition, Figure 4 also shows the impact of multi-round rewriting on token usage. Since the increasing length of rewritten *NL* may cause the token increment in the whole workflow. However, due to the subsequent rounds only processing the bad samples, the overall token usage is relatively reduced.

Table 3: Ablation study on Spider-dev set. CK represents the `Checker`. RE represents the `Reflector`.

NL2SQL **Methods**	**Base**	**w/o CK**	**w/o RE**	**All**
C3 + GPT-3.5-Turbo	81.9	78.8(↓)	80.8(↓)	**82.4**
DAIL + GPT-3.5-Turbo	76.3	76.9(↑)	77.7(↑)	**78.1**
DAIL + GPT-4	83.1	78.1(↓)	81.8(↓)	**83.6**
DTS-SQL	75.2	71.0(↓)	74.4(↓)	**77.5**
NatSQL + T5-Base	69.4	70.2(↑)	71.5(↑)	**72.2**
NatSQL + T5-3B	71.8	71.6(↓)	72.9(↑)	**75.7**
RESDSQL	81.8	77.3(↓)	80.4(↓)	**82.2**

Table 4: Ablation study on Spider-dev set. HC. represents the hand-craft experiences, IN. represents the self-initialized experiences, LE. represents the learning-updated experiences.

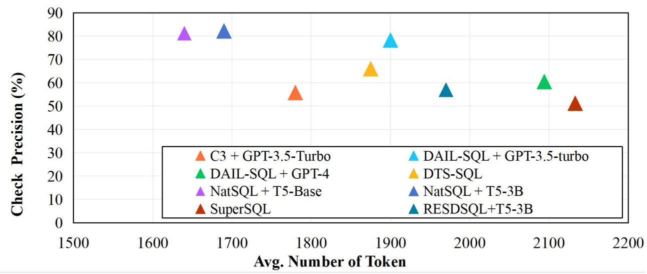
NL2SQL **Methods**	**Base**	**HC.**	**IN.**	**LE.**
DTS-SQL	75.2	77.2 (↑)	76.4 (↑)	**77.5**
NatSQL + T5-Base	69.4	70.7 (↑)	72.1 (↑)	**72.2**
NatSQL + T5-3B	71.8	75.0 (↑)	73.7 (↑)	**75.7**

Ablation Study

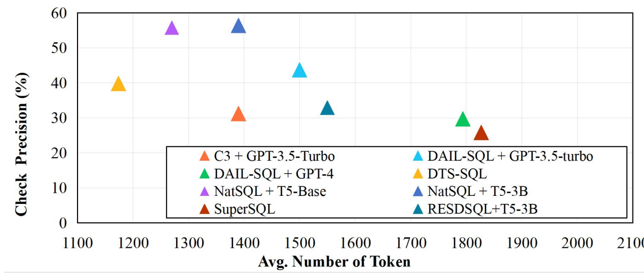
We perform several ablation studies for REWRITER.

Ablation Study for Agents Table 3 illustrates the results of the ablation study for the REWRITER, where w/o `Checker` represents the rewriting of all *NL* in the dataset, and w/o `Reflector` represents instructing `Rewriter` to rewrite in the absence of analysis. The experimental results demonstrate that the `Checker` and `Reflector` within the REWRITER significantly contribute to the whole rewrite action. Removing any one of these agents can result in a decrease in the quality of rewritten *NL*. In addition, Table 1 and Table 3 illustrate that rewriting all *NL* may lead to new hallucinations, which negatively affect the *NL2SQL* model. The `Checker` can effectively detect the wrong *SQL*, so as to ensure that the REWRITER can process the real flaw *NL* as much as possible, minimizing the new hallucinations caused by rewriting the correct *NL*.

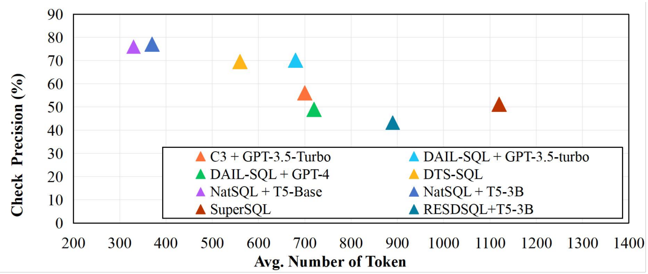
Ablation Study for Self-reflection Table 4 presents the ablation experiment on `Reflector`. In this study, we examine the performance of `Reflector` in various settings, encompassing the utilization of hand-crafted experiences, self-initialized experiences, and learning-updated experiences. The hand-crafted experience refers to the flaws and rewriting action schemes summarized by NL2SQL experts; The self-initialized experience refers to related experience generated at the initialization stage of `Reflector`; The learning-updated



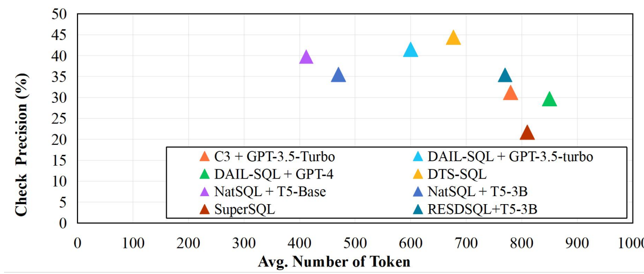
(a) GPT-4o



(b) GPT-4



(c) GLM-4



(d) LLaMA-3-8B

Figure 5: Effectiveness of `Checker`. Token efficiency vs. Precision on Spider-dev set.

experience refers to the experience that the `Reflector` summarizes and updates in the multi-round rewriting process. Experimental results illustrate that the rules initialized by `Reflector` can better enable rewritten *NL* to improve *SQL* generation accuracy on most *NL2SQL* methods. Over the process of multi-round rewriting, the `Reflector` is able to improve further the performance of subsequent rewriting based on the experience and feedback accumulated from previous rewritten *NL* and generation result.

Discussion

Based on the aforementioned experiments, several experiences and heuristic insights are as follows:

- Table 5 and Table 6 enumerate common flaw and corresponding rewriting actions for flawed *NL* summarized by `Reflector`. During the process of *NL2SQL*, these flaws can lead to misunderstandings in the database, query misinterpretations, or other issues, ultimately resulting in reduced accuracy of *SQL* generation. The rewriting actions implemented by REWRITER improve the performance of associated *NL2SQL* methods by addressing these flaws.
- `Reflector` does not initialize too many experiences, as this could result in duplicate or similar experience descriptions. For example, `ambiguity` and `fuzzy statements` actually refer to the same flaw that the given *NL* have multiple meanings or are ambiguous.
- The running and testing of the REWRITER framework and above *NL2SQL* method resulted in a significant consumption of tokens, estimated to be around 45 million tokens. Due to experimental cost constraints, more *NL2SQL* methods cannot be plugged into REWRITER.

Related Work

NL2SQL Translation The recent advancement of LLM contributed to the progress of *NL2SQL* translation. Researchers are able to construct *NL2SQL* models through prompt engineering, supervised fine-tuning, and other innovative techniques. The current research focuses on designing and optimizing *NL2SQL* workflow modules, including preprocessing modules (e.g., schema linking (Li et al. 2023a), database content representation (Talaei et al. 2024; Li et al. 2024b)), translation strategies (e.g design of encoder and decoder (Fu et al. 2023; Wang et al. 2019), intermediate representations (Gu et al. 2023a,b; Gan et al. 2021; Wolfson et al. 2020; Eyal et al. 2023; Gao et al. 2022; Li et al. 2024c)), and post-processing module (e.g., *SQL* correction (Pourreza and Rafiei 2023; Wang et al. 2024), self-consistency (Gao et al. 2023; Dong et al. 2023)), with limited attention given to the original input *NL*. These approaches might ignore potential ambiguities and missing details within the original *NL* (Tang et al. 2022; Luo et al. 2021, 2022b, 2018b; Shen et al. 2021; Qin et al. 2020). In real-world situations, users' understanding of the database can significantly impact the quality of *NL*, potentially leading to model errors such as hallucinations or incorrect *SQL* generation.

In this paper, our goal is to rewrite the flawed *NL*, minimize the misrepresentations and ambiguous statements in the *NL*, and fix the gap between *NL* and *NL2SQL* model, which can enhance the robustness and performance of *SQL* generation.

Text Rewriting Text rewriting methodologies aim to assist users in enhancing the quality of their input *NL* or adapting the narrative style to suit various contexts. In the existing research, text rewriting is extensively employed for privacy protection (Igamberdiev and Habernal 2023), text summary (Bao

Flaw Type	Description	Flawed *NL*	Details
`Missing Info.`	Users may omit information that is essential to generate correct *SQL* queries, which may cause the model to fail to generate accurate *SQL* statements.	What model has the most different versions?	*NL* does not refer to which table or column to query.
`Wrong Entity`	Entities (e.g. table names or column names) mentioned by the user in the *NL* may be wrong or not present in the database, which will cause the model to generate invalid *SQL* queries.	What are the locations and names of all <u>stations</u> with capacity between 5000 and 10000?	**stations** do not exist in the *DB*, the real entity should be **stadiums**.
`Ambiguity`	*NL* may have multiple meanings or ambiguous words that make it difficult for the model to generate the correct *SQL* query.	What are the <u>names</u> and release years for all the songs of the youngest singer?	There are multiple similar column names (**Song Name** and **Name**) in *DB*.
`Non-standard Statement`	Users may use non-standard or colloquial expressions, which may cause the model to fail to understand the user's intention correctly.	Show name, country, and age for all singers ordered by age <u>from the oldest to the youngest</u>.	The expression of the second half of the sentence should be modified.

Table 5: The most common flaws summarized by `Reflector` and related *NL* examples.

Action Type	Description	Rewritten *NL*
`Complete Information`	Add omitted key information entities to *NL* to ensure that the generated *SQL* query is accurate.	Which <u>**car model**</u> in the <u>**model list**</u> table has the highest number of distinct versions in the <u>**cars data**</u> table?
`Correct Entities`	Fix incorrect entity in the *NL* with reference of *DB*, ensuring that all mentioned entities are correct and present in the *DB*.	What are the locations and names of all <u>**stadiums**</u> with capacity between 5000 and 10000?
`Disambiguation`	Reformulate the ambiguous keywords in *NL* in conjunction with the *DB*, and ensure that each word has a clear meaning.	Show the <u>**Song Name**</u> and <u>**Song release year**</u> of the song by the singer with the lowest Age
`Normalize Statement`	Convert non-standard or colloquial expressions into standard language that the model can understand.	Show name, country, and age for all singers ordered by age <u>**in descending order**</u>.

Table 6: The most effective action experience summarized by `Reflector` and the rewritten *NL* for the examples in Table 5.

and Zhang 2023), heterogeneous data representation (Wu et al. 2023), and various other tasks. In addition, several studies have explored the possibility of text rewriting to improve the quality of *NL* and align user's intent (Hwang et al. 2023; Shu et al. 2024).

Based on the above references, this study aims to apply text rewriting techniques to *NL2SQL* scenarios. The objective is to guide *NL2SQL* models towards generating SQL queries that better align with the user's intent, by enhancing the quality and coherence of *NL* of users.

LLM Agents LLM-based agents framework is designed for the simulation of interactive behavior of specific targets or assisting users in solving specified tasks (Park et al. 2023; Hua et al. 2023; Wang et al. 2023; Zhu et al. 2024). It usually comprises various modules, such as observation, memory, planning, and response (Zhou et al. 2023; Zhang et al. 2024). In NLP-related applications, existing research usually designs different agents in the form of task decomposition and utilizes data stream to make them collaborative (Talaei et al. 2024; Wang et al. 2024; Li et al. 2023c), which can reduce the processing difficulty of sub-tasks and improve the accuracy of generation.

In this research, we propose REWRITER, a plug-and-play multi-agent framework to rewrite the user's *NL*. It comprises three agents with different divisions of labor, jointly committed to the check-reflection-rewrite workflow.

Conclusion

In this paper, we propose REWRITER, a novel plug-and-play framework designed to enhance *NL2SQL* systems by automatically rewriting flawed *NL* inputs. By leveraging database knowledge such as column values and foreign keys, REWRITER reduces translation errors and improves *SQL* generation accuracy. Extensive experiments on the Spider and BIRD benchmarks demonstrate the effectiveness of REWRITER, achieving consistent improvements in execution accuracy by 1.6% and 2.0%, respectively.

References

Bao, G.; and Zhang, Y. 2023. A general contextualized rewriting framework for text summarization. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 31: 1624–1635.

Dong, X.; Zhang, C.; Ge, Y.; Mao, Y.; Gao, Y.; Lin, J.; Lou, D.; et al. 2023. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306*.

Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Yang, A.; Fan, A.; et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783*.

Eyal, B.; Bachar, A.; Haroche, O.; Mahabi, M.; and Elhadad, M. 2023. Semantic Decomposition of Question and SQL for Text-to-SQL Parsing. *arXiv preprint arXiv:2310.13575*.

Fu, H.; Liu, C.; Wu, B.; Li, F.; Tan, J.; and Sun, J. 2023. Catsql: Towards real world natural language to sql applications. *Proceedings of the VLDB Endowment*, 16(6): 1534–1547.

Gan, Y.; Chen, X.; Xie, J.; Purver, M.; Woodward, J. R.; Drake, J.; and Zhang, Q. 2021. Natural SQL: Making SQL Easier to Infer from Natural Language Specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, 2030–2042. Punta Cana, Dominican Republic: Association for Computational Linguistics.

Gao, C.; Li, B.; Zhang, W.; Lam, W.; Li, B.; Huang, F.; Si, L.; and Li, Y. 2022. Towards generalizable and robust text-to-sql parsing. *arXiv preprint arXiv:2210.12674*.

Gao, D.; Wang, H.; Li, Y.; Sun, X.; Qian, Y.; Ding, B.; and Zhou, J. 2023. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *CoRR*, abs/2308.15363.

GLM, T.; Zeng, A.; Xu, B.; Wang, B.; Zhang, C.; Yin, D.; Rojas, D.; Feng, G.; Zhao, H.; Lai, H.; et al. 2024. ChatGLM: A Family of Large Language Models from GLM-130B to GLM-4 All Tools. *arXiv preprint arXiv:2406.12793*.

Gu, Z.; Fan, J.; Tang, N.; Cao, L.; Jia, B.; Madden, S.; and Du, X. 2023a. Few-shot text-to-sql translation using structure and content prompt learning. *Proceedings of the ACM on Management of Data*, 1(2): 1–28.

Gu, Z.; Fan, J.; Tang, N.; Zhang, S.; Zhang, Y.; Chen, Z.; Cao, L.; Li, G.; Madden, S.; and Du, X. 2023b. Interleaving Pre-Trained Language Models and Large Language Models for Zero-Shot NL2SQL Generation. CoRR abs/2306.08891 (2023). *arXiv preprint arXiv:2306.08891*.

Hua, W.; Fan, L.; Li, L.; Mei, K.; Ji, J.; Ge, Y.; Hemphill, L.; and Zhang, Y. 2023. War and peace (waragent): Large language model-based multi-agent simulation of world wars. *arXiv preprint arXiv:2311.17227*.

Hwang, A.; Oza, N.; Callison-Burch, C.; and Head, A. 2023. Rewriting the script: Adapting text instructions for voice interaction. In *Proceedings of the 2023 ACM designing interactive systems conference*, 2233–2248.

Igamberdiev, T.; and Habernal, I. 2023. DP-BART for privatized text rewriting under local differential privacy. *arXiv preprint arXiv:2302.07636*.

Katsogiannis-Meimarakis, G.; and Koutrika, G. 2023. A survey on deep learning approaches for text-to-SQL. *The VLDB Journal*, 32(4): 905–936.

Li, B.; Luo, Y.; Chai, C.; Li, G.; and Tang, N. 2024a. The Dawn of Natural Language to SQL: Are We Fully Ready? *Proc. VLDB Endow.*, 17(11): 3318–3331.

Li, H.; Zhang, J.; Li, C.; and Chen, H. 2023a. RESDSQL: Decoupling Schema Linking and Skeleton Parsing for Text-to-SQL. In *AAAI*.

Li, H.; Zhang, J.; Liu, H.; Fan, J.; Zhang, X.; Zhu, J.; Wei, R.; Pan, H.; Li, C.; and Chen, H. 2024b. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data*, 2(3): 1–28.

Li, J.; Hui, B.; Qu, G.; Li, B.; Yang, J.; Li, B.; Wang, B.; Qin, B.; Geng, R.; Huo, N.; Zhou, X.; Ma, C.; Li, G.; Chang, K. C. C.; Huang, F.; Cheng, R.; and Li, Y. 2023b. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. arXiv:2305.03111.

Li, Y.; Yu, Y.; Li, H.; Chen, Z.; and Khashanah, K. 2023c. TradingGPT: Multi-agent system with layered memory and distinct characters for enhanced financial trading performance. *arXiv preprint arXiv:2309.03736*.

Li, Z.; Wang, X.; Zhao, J.; Yang, S.; Du, G.; Hu, X.; Zhang, B.; Ye, Y.; Li, Z.; Zhao, R.; et al. 2024c. PET-SQL: A Prompt-enhanced Two-stage Text-to-SQL Framework with Cross-consistency. *arXiv preprint arXiv:2403.09732*.

Liu, X.; Shen, S.; Li, B.; Ma, P.; Jiang, R.; Luo, Y.; Zhang, Y.; Fan, J.; Li, G.; and Tang, N. 2024. A Survey of NL2SQL with Large Language Models: Where are we, and where are we going? *CoRR*, abs/2408.05109.

Luo, Y.; Qin, X.; Chai, C.; Tang, N.; Li, G.; and Li, W. 2022a. Steerable Self-Driving Data Visualization. *IEEE Trans. Knowl. Data Eng.*, 34(1): 475–490.

Luo, Y.; Qin, X.; Tang, N.; and Li, G. 2018a. DeepEye: Towards Automatic Data Visualization. In *ICDE*, 101–112. IEEE Computer Society.

Luo, Y.; Qin, X.; Tang, N.; Li, G.; and Wang, X. 2018b. DeepEye: Creating Good Data Visualizations by Keyword Search. In *SIGMOD Conference*, 1733–1736. ACM.

Luo, Y.; Tang, N.; Li, G.; Chai, C.; Li, W.; and Qin, X. 2021. Synthesizing Natural Language to Visualization (NL2VIS) Benchmarks from NL2SQL Benchmarks. In *SIGMOD Conference*, 1235–1247. ACM.

Luo, Y.; Tang, N.; Li, G.; Li, W.; Zhao, T.; and Yu, X. 2020. DeepEye: A Data Science System for Monitoring and Exploring COVID-19 Data. *IEEE Data Eng. Bull.*, 43(2): 121–132.

Luo, Y.; Tang, N.; Li, G.; Tang, J.; Chai, C.; and Qin, X. 2022b. Natural Language to Visualization by Neural Machine

Translation. *IEEE Trans. Vis. Comput. Graph.*, 28(1): 217–226.

OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*.

Park, J. S.; O'Brien, J. C.; Cai, C. J.; Morris, M. R.; Liang, P.; and Bernstein, M. S. 2023. Generative Agents: Interactive Simulacra of Human Behavior. In *In the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*, UIST '23. New York, NY, USA: Association for Computing Machinery.

Pourreza, M.; and Rafiei, D. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. *arXiv preprint arXiv:2304.11015*.

Pourreza, M.; and Rafiei, D. 2024. DTS-SQL: Decomposed Text-to-SQL with Small Large Language Models. *arXiv preprint arXiv:2402.01117*.

Qin, X.; Luo, Y.; Tang, N.; and Li, G. 2020. Making data visualization more efficient and effective: a survey. *VLDB J.*, 29(1): 93–117.

Rai, D.; Wang, B.; Zhou, Y.; and Yao, Z. 2023. Improving Generalization in Language Model-based Text-to-SQL Semantic Parsing: Two Simple Semantic Boundary-based Techniques. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 150–160. Toronto, Canada: Association for Computational Linguistics.

Shen, L.; Shen, E.; Luo, Y.; Yang, X.; Hu, X.; Zhang, X.; Tai, Z.; and Wang, J. 2021. Towards Natural Language Interfaces for Data Visualization: A Survey. *CoRR*, abs/2109.03506.

Shu, L.; Luo, L.; Hoskere, J.; Zhu, Y.; Liu, Y.; Tong, S.; Chen, J.; and Meng, L. 2024. Rewritelm: An instruction-tuned large language model for text rewriting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 18970–18980.

Talaei, S.; Pourreza, M.; Chang, Y.-C.; Mirhoseini, A.; and Saberi, A. 2024. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*.

Tang, J.; Luo, Y.; Ouzzani, M.; Li, G.; and Chen, H. 2022. Sevi: Speech-to-Visualization through Neural Machine Translation. In *SIGMOD Conference*, 2353–2356. ACM.

Wang, B.; Ren, C.; Yang, J.; Liang, X.; Bai, J.; Chai, L.; Yan, Z.; Zhang, Q.-W.; Yin, D.; Sun, X.; and Li, Z. 2024. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. arXiv:2312.11242.

Wang, B.; Shin, R.; Liu, X.; Polozov, O.; and Richardson, M. 2019. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942*.

Wang, L.; Ma, C.; Feng, X.; Zhang, Z.; Yang, H.; Zhang, J.; Chen, Z.; Tang, J.; Chen, X.; Lin, Y.; et al. 2023. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*.

Wolfson, T.; Geva, M.; Gupta, A.; Gardner, M.; Goldberg, Y.; Deutch, D.; and Berant, J. 2020. Break it down: A question understanding benchmark. *Transactions of the Association for Computational Linguistics*, 8: 183–198.

Wu, Y.; Hu, N.; Qi, G.; Bi, S.; Ren, J.; Xie, A.; and Song, W. 2023. Retrieve-rewrite-answer: A kg-to-text enhanced llms framework for knowledge graph question answering. *arXiv preprint arXiv:2309.11206*.

Yu, T.; Zhang, R.; Yang, K.; Yasunaga, M.; Wang, D.; Li, Z.; Ma, J.; Li, I.; Yao, Q.; Roman, S.; Zhang, Z.; and Radev, D. 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*. Brussels, Belgium: Association for Computational Linguistics.

Zhang, J.; Xiang, J.; Yu, Z.; Teng, F.; Chen, X.; Chen, J.; Zhuge, M.; Cheng, X.; Hong, S.; Wang, J.; Zheng, B.; Liu, B.; Luo, Y.; and Wu, C. 2024. AFlow: Automating Agentic Workflow Generation. *CoRR*, abs/2410.10762.

Zhou, W.; Jiang, Y. E.; Li, L.; Wu, J.; Wang, T.; Qiu, S.; Zhang, J.; Chen, J.; Wu, R.; Wang, S.; Zhu, S.; Chen, J.; Zhang, W.; Zhang, N.; Chen, H.; Cui, P.; and Sachan, M. 2023. Agents: An Open-source Framework for Autonomous Language Agents. arXiv:2309.07870.

Zhu, Y.; Du, S.; Li, B.; Luo, Y.; and Tang, N. 2024. Are Large Language Models Good Statisticians? *CoRR*, abs/2406.07815.

Appendix

Prompt of Agents

Prompt of the `Checker`

```
#### Task Description:
Based on the given database schema, Natural Language (NL), SQL query and execution result (only
showing top three records), determine whether the SQL query is expected to return the correct
results. You need to follow the steps below for step-by-step reasoning:
1. Syntax Check: Verify if the SQL query adheres to the basic SQL syntax rules.
2. Logical Verification: Extract the database schema information involved in the SQL query.
Based on the given guidelines below, step-by-step determine whether the corresponding SQL
logic matches the expectations of the NL.
3. Execution Analysis: Based on the result set returned from executing the SQL query in the
real database, verify whether it meets the requirements of the NL.
4. Ambiguity Detection: Check if there is any semantic ambiguity in the NL and whether the
tables or columns matching the NL in the SQL query are ambiguous. If any ambiguity exists,
explain the possible ambiguities and directly determine that the SQL query is incorrect.
5. Final Determination: If no issues are found in the above steps, predict the SQL query as
correct; otherwise, predict the SQL query as incorrect.

### Output Format:
{
  "details": <YOUR THINKING DETAILS>,
  "result": <"TRUE" OR "FALSE">
}

### INPUT:
SCHEMA:  # Fill the database content
{SCHEMA_SLOT}
NL:   # Fill the NL
{NL_SLOT}
SQL:  # Fill the generated SQL
{SQL_SLOT}
EXECUTION RESULT:  # Fill the execution result
{EXECUTION_RESULT_SLOT}

### OUPUT:
```

Prompt of the `Reflector`**- Initial Action Experience Generation**

```
#### Task Description:
Please think deeply about the corresponding modification operation and the corresponding operation
description according to the following problems of the user text, and output in JSON format.

### Output Format:
{
    'Operation Type': 'Description',
    'Operation Type': 'Description',
    ...
}

### The types of problems are:
{ERROR_SPACE}
```

Prompt of the `Reflector`- Initial Error Experience Generation

```
#### Task Description:
Please list 10 possible problems with user-provided text in the Text-to-SQL task and the
corresponding explanation of the problem), and output them in JSON format:

### Output Format:
{
    'Problem Type': 'Description',
    'Problem Type': 'Description',
    ...
}
```

Prompt of the `Reflector`

```
#### Task Description:
As the AI assistant, your task is to analyze the flaws of the questions (NL)
entered by the user, given the database information and examples. This NL cannot be properly
translated into a SQL query correctly.
The goal of this task is to provide an analysis and recommendations for a given NL that
can be modified and optimized. To do this well, you need to look for the following details
in the question:
{FLAW_SLOT}  # Fill the flaw experience

Please analyze whether there are above flaws in the above details in the NL. If so, please select the
available rewriting actions for the NL. The available rewriting actions are as follows:
{ACTION_SLOT} # Fill the action experience

Please clearly mark the keywords in each statement and the corresponding modification suggestions.
### Output Format:
{
  "reflection": <YOUR THINKING DETAILS>
}
### Here are a reflection example:
{
    "reflection":
        "The flaw is...
        (describe the specific FLAW with DB),
        the recommended operation is... ,
        (describes the ACTION with DB);
        The flaw is...
        (describe the specific FLAW with DB),
        the recommended operation is... ,
        (describes the ACTION with DB);
        ..."
}

### INPUT:
SCHEMA: # Fill the Database content
{SCHEMA_SLOT}
NL:     # Fill the flawed NL
{NL_SLOT}

### OUPUT:
```

Prompt of the `Rewriter`

```
#### Task Description:
As the AI assistant, your task is to rewrite the NL entered by the user based on the given
database information and reflection.
This NL has some flaws and got bad generation in the downstream models, so you need to make this
NL as reliable as possible.
The rewritten NL should express more complete and accurate database information requirements
as far as possible. In order to do this task well, you need to follow these steps to think and
process step by step:
1. Please review the given reflection and DB information, and first check whether the NL contains
the corresponding key information and the corresponding flaws. If they exists, please modify,
supplement or rewrite it in the statement of NL by combining the reflection and DB.
2. Please rewrite the original NL based on the above process. On the premise of providing more
complete and more accurate database information, the structure of the rewritten NL should be similar
to the original statement as far as possible. All rewritten statements do not allow delimiters,
clauses, additional hints or explanations. DONT CONVERT IT INTO QUERY.

{
  "details": <YOUR STEP-BY-STEP THINKING DETAILS>,
  "result": <YOUR FINAL REWRITED NL>
}

### INPUT:
SCHEMA: # Fill the database content
{SCHEMA_SLOT}
NL:     # Fill the flaw NL
{NL_SLOT}
REFLECTION: # Fill the reflection
{REFLECTION_SLOT}

### OUTPUT:
```

Description of *NL2SQL* Baselines

- **DAIL-SQL** (Gao et al. 2023) DAIL-SQL encodes structural knowledge and selects the corresponding few-shot prompt by calculating the similarity order of the skeleton. It also enhances reasoning efficiency by blocking cross-domain specific words in the representation.
- **C3** (Dong et al. 2023) implements schema filtering in the schema linking process, and uses prompt calibration and self-consistency to ensure the stability of *SQL* generation and reduce inference costs
- **DTS-SQL** (Pourreza and Rafiei 2024) is designed to address user privacy data optimization. It comprises two main sub-tasks: schema linking and *SQL* generation. A two-stage fine-tuning approach is implemented to effectively align the performance of the open-source LLM with that of the proprietary LLM.
- **NatSQL + T5** (Rai et al. 2023) is a semantic-boundary-based technique, which is based on semantic boundaries and involves the use of special tags to identify aligned semantic boundaries between the source problem and the target SQL.
- **RESDSQL** (Li et al. 2023a) comprises a ranking-enhanced encoder and a skeleton-aware decoder. The encoder selectively incorporates the most pertinent schema items, as opposed to the complete set of unordered schema items. Meanwhile, the decoder initially produces the framework and subsequently the specific *SQL* query, thereby implicitly constraining *SQL* parsing.
- **GPT-3.5 & GPT-4** (OpenAI 2023) uses chain-of-thought and zero-shot techniques to construct schema linking and generation prompts for the processing and the generation of the *SQL*.