NL2SQLBench: A Modular Benchmarking Framework for LLM-Enabled NL2SQL Solutions [Experiment, Analysis & Benchmark]

Shizheng Hou
National University of Singapore
shizheng_hou@u.nus.edu

Wenqi Pei
National University of Singapore
wenqi_pei@u.nus.edu

Nuo Chen
National University of Singapore
nuochen@comp.nus.edu.sg

Quang-Trung Ta
National University of Singapore
taqt@comp.nus.edu.sg

Peng Lu
Zhejiang University
peng.lu@zju.edu.cn

Beng Chin Ooi
National University of Singapore
ooibc@comp.nus.edu.sg

ABSTRACT

Natural Language to SQL (NL2SQL) technology empowers non-expert users to query relational databases without requiring SQL expertise. While large language models (LLMs) have greatly improved NL2SQL algorithms, their rapid development outpaces systematic evaluation, leaving a critical gap in understanding their effectiveness, efficiency, and limitations. To this end, we present **NL2SQLBench**, the first modular evaluation and benchmarking framework for LLM-enabled NL2SQL approaches. Specifically, we dissect NL2SQL systems into three core modules: Schema Selection, Candidate Generation, and Query Revision. For each module, we comprehensively review existing strategies and propose novel fine-grained metrics that systematically quantify module-level effectiveness and efficiency. We further implement these metrics in a flexible multi-agent framework, allowing configurable benchmarking across diverse NL2SQL approaches. Leveraging **NL2SQLBench**, we rigorously evaluate ten representative open-source methods on two datasets, the BIRD development and the ScienceBenchmark datasets, using two LLMs, DeepSeek-V3 and GPT-4o mini. We systematically assess each approach across the three core modules and evaluate multiple critical performance dimensions. Our evaluation reveals significant gaps in existing NL2SQL methods, highlighting not only substantial room for accuracy improvements but also the significant computational inefficiency, which severely hampers real-world adoption. Furthermore, our analysis identifies critical shortcomings in current benchmark datasets and evaluation rules, emphasizing issues such as inaccurate gold SQL annotations and limitations in existing evaluation rules. By synthesizing these detailed insights into a unified, transparent, and reproducible benchmarking, our study not only establishes a clear reference point for fair comparison across approaches but also serves as essential guidance for future targeted innovation in NL2SQL technology, thus advancing the practical deployment and real-world applicability of NL2SQL technologies.

PVLDB Reference Format:
Shizheng Hou, Wenqi Pei, Nuo Chen, Quang-Trung Ta, Peng Lu, and Beng Chin Ooi. **NL2SQLBench**: A Modular Benchmarking Framework for LLM-Enabled NL2SQL Solutions [Experiment, Analysis & Benchmark]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://anonymous.4open.science/r/NL2SQLBench-D6EC/.

1 INTRODUCTION

Natural Language to SQL (NL2SQL) is a pivotal technology that empowers non-expert users to query relational databases using natural language (NL) questions, without requiring any SQL expertise. This technology holds significant value as it democratizes access to data, empowering users to engage with complex database systems and supporting diverse applications across various business domains [32, 39, 55, 59, 66, 70, 75]. Concurrently, as noted by Liu et al. [34], the provision of NL2SQL solutions has shifted from a conceptual idea to an essential strategy among database vendors. Consequently, the question of how to effectively implement such solutions has become an actively discussed topic [1, 13].

Recent advancements in Large Language Models (LLMs) [20, 42, 78] have introduced a transformative paradigm for NL2SQL tasks, offering new opportunities for enhanced performance and adaptability. Numerous LLM-based approaches [3, 4, 11, 12, 15, 16, 21, 27, 29, 31, 36–38, 46, 48, 50, 53, 54, 61, 73, 77] have been proposed, establishing themselves as the most prominent solutions in the NL2SQL landscape.

As articulated by Liu et al. [34], these systems typically consist of three core modules: (1) *Schema Selection*: select the most relevant tables and columns from the databases; (2) *Candidate Generation*: generate candidate SQL queries based on the NL queries; (3) *Query Revision*: refine the candidate SQL queries to generate the final SQL queries. These modules form the fundamental structure of most modern NL2SQL systems, as depicted in the top half of Figure 1.

The primary evaluation metrics currently employed in existing NL2SQL benchmarks [22, 28, 72], typically employ three metrics, namely *Exact Match (EM)*, *Execution Accuracy (EX)*, and *Valid Efficiency Score (VES)* that only concern the overall correctness and execution efficiency of the finally generated SQL. However, such metrics do not assess the effectiveness and efficiency of each individual core module, hindering deeper insights into where improvements are needed most urgently.

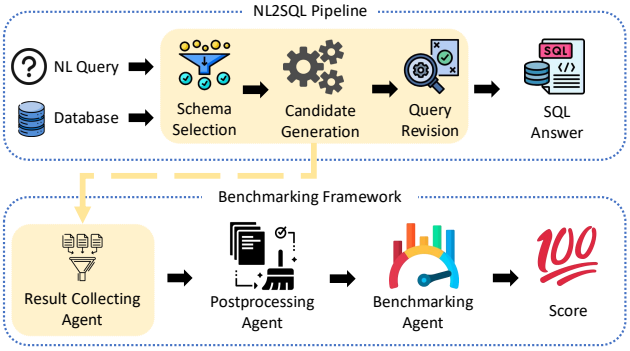


Figure 1: Overview of LLM-enabled NL2SQL approaches and our benchmarking framework.

Furthermore, although an increasing number of solutions continue to shine on NL2SQL leaderboards [22, 28, 72], the computational costs and latency incurred by LLMs remain overlooked. Additionally, variability in evaluation settings across studies complicates fair comparisons between different strategies for each module. There is no unified framework to perform a holistic comparison of the individual modules under the same experimental setting.

Therefore, the rapid development of LLM-based NL2SQL methods has outpaced systematic evaluation, leaving a critical gap in understanding their effectiveness and efficiency. These limitations highlight two urgent and interconnected research questions: (1) *How can we systematically measure the effectiveness and efficiency of individual NL2SQL modules?*, and (2) *How can we establish a consistent, reproducible framework to evaluate these modules across different methods fairly?* Answering these questions is crucial for understanding the performance of techniques used in each module, which ultimately guides the development of more accurate and efficient NL2SQL algorithms.

Our work. To tackle these challenges, we present **NL2SQLBench**, the first modularized benchmarking framework specifically designed for systematic evaluation of the core modules *Schema Selection*, *Candidate Generation*, and *Query Revision* of LLM-enabled NL2SQL solutions. We first review comprehensively different techniques implemented in each module by existing NL2SQL systems. To facilitate the modularized evaluation, we propose new evaluation metrics for each module, enabling a holistic assessment of the effectiveness and efficiency of these techniques. Then, we develop a flexible, multi-agent benchmarking framework consisting of three agents: *Result Collecting*, *Result Postprocessing*, and *Benchmarking* agent. These agents are designed to be modular and configurable, allowing compatibility with diverse NL2SQL approaches.

We apply our framework to evaluate ten representative and open-sourced LLM-based NL2SQL approaches from the BIRD leaderboard[1] [28], using two datasets, the BIRD [28] and the ScienceBenchmark [76] datasets, and two LLMs, DeepSeek-V3 [9] and GPT-4o mini [45]. Then, we conduct a comprehensive analysis of their performance. Our comprehensive benchmarking analysis reveals that apart from the large room for improving accuracy at the modular level, computational efficiency is another obstacle to industrial adoption. Furthermore, our findings highlight previously underappreciated challenges regarding benchmark dataset quality and evaluation rules, areas that warrant immediate attention. Beyond just identifying these challenges, we further synthesize our insights into a series of practical guides and usability analyses (e.g., Sec. 4.2.4, 4.7), offering actionable guidance for developers to navigate the accuracy-efficiency trade-offs in real-world system development.

Contributions. We summarize our contributions as follows.

- We comprehensively review the three core modules of LLM-enabled NL2SQL solutions, namely *Schema Selection*, *Candidate Generation*, and *Query Revision*, on their details, representative implementations.
- We propose a set of new, fine-grained evaluation metrics specifically tailored for systematically assessing the effectiveness and efficiency of each individual module. Unlike existing overall metrics, our proposed metrics enable precise, detailed analyses of performance across multiple dimensions.
- We develop **NL2SQLBench**, the first modular, multi-agent benchmarking framework for evaluating core modules. Our framework comprises three seamlessly integrated agents—*Result Collecting*, *Result Postprocessing*, *Benchmarking*—each designed with flexibility in mind, ensuring broad applicability across diverse NL2SQL methods.
- Leveraging **NL2SQLBench**, we comprehensively benchmark ten representative open-source NL2SQL approaches from the BIRD leaderboard. Our evaluation reveals substantial room for improvement in both accuracy and computational efficiency, warranting further study and providing valuable insights for researchers aiming to advance NL2SQL methodologies.
- Our in-depth analysis uncovers previously unnoticed yet critical limitations in the quality of existing datasets and evaluation rules. Our findings strongly advocate for the creation of more robust, reliable, and flexible benchmark datasets and evaluation methods in future research.
- We distill the benchmarking results into practical guides and usability analyses, providing concrete recommendations and a systematic workflow to help practitioners effectively diagnose bottlenecks, navigate design trade-offs, and iteratively improve NL2SQL systems.

2 COMPREHENSIVE REVIEW OF LLM-ENABLED NL2SQL SOLUTIONS

2.1 NL2SQL Task Formulation

An NL2SQL task involves translating a natural language query into a corresponding SQL statement that can be executed on a relational database. Specifically, given a natural language query Q on a database D, the task is aimed to produce a SQL statement S based on Q and D. Formally, the task can be defined as: $S = f(Q, D)$.

Most LLM-enabled NL2SQL approaches follow a three-stage architecture comprising three core modules: *Schema Selection*, *Candidate Generation*, and *Query Revision*, which correspond to the three phrases *pre-processing*, query generation, and post-processing [34, 57], as depicted in Figure 1. Since these three modules all leverage LLMs to perform their functionalities, they need to consider various aspects such as how to design proper prompts, select few-shot examples, and decide the context length to feed into the LLMs.

[1] https://bird-bench.github.io/

In the following, we comprehensively review how these modules are implemented in existing systems.

2.2 Schema Selection Module

Leveraging LLMs to generate SQL queries from an NL question requires constructing a prompt that incorporates the question, database schema, and task-specific instructions. However, the limited context window of LLMs poses challenges in accommodating large schemas [13, 32]. Furthermore, lengthy prompts increase cost and degrade performance [33]. Hence, *Schema Selection* module, which is also known as schema linking [4, 5, 17, 23, 26, 34, 48, 62], is implemented by many solutions to extract the most relevant tables and columns. We describe representative strategies employed in existing works below and summarize them in Table 1.

Table 1: Classification of Schema Selection strategies.

Approach	Few-shot CoT	Multi-Stage Pruning	Preliminary SQL
C3-SQL [11]	✗	✓	✗
DIN-SQL [48]	✓	✗	✗
MAC-SQL [61]	✓	✗	✗
CHESS$_{(IR,SS,CG)}$ [60]	✓	✓	✗
TA-SQL [53]	✗	✗	✓
OpenSearch-SQL [68]	✓	✓	✗
RSL-SQL [5]	✓	✗	✓
PET-SQL [31]	✓	✗	✓
GSR [2]	✗	✗	✓

Few-Shot Chain-of-Thought (CoT). This strategy utilizes the few-shot in-context learning [10, 30] capabilities of LLMs supplemented by CoT prompting [7, 65] to identify relevant tables and columns for NL queries, enabling LLMs to generalize without extensive task-specific training. For instance, MAC-SQL [61] and DIN-SQL [48] selects demonstrations that map NL queries to target schemas. By guiding the LLM through a step-by-step process, the CoT method helps the model better interpret the NL query and effectively select relevant tables and columns.

Multi-Stage Schema Pruning. By progressively narrowing the database schema in a multi-step manner, this strategy aims to ensure that only the necessary schema are considered. For instance, CHESS$_{(IR,SS,CG)}$ [60] leverages locality-sensitive hashing and vector retrieval to efficiently identify relevant database values and catalogs, followed by a three-stage pruning process: broadly columns filtering, followed by selecting the most relevant table, and finally picking necessary columns. OpenSearch-SQL [68] employs a similar way: information retrieval and column filtering - but chooses to recall relevant schema by leveraging both LLMs and vector retrieval.

Preliminary-SQL Enhanced. Instead of explicitly instructing LLMs on schema selection, this method begins by prompting the LLMs to generate a preliminary SQL query based on the natural language query. Once such a query is generated, relevant schema elements, such as tables, columns, and relationships, are extracted from it. TA-SQL [53] is a notable example of employing this strategy. It first generates a dummy SQL query and then extracts schema entities. PET-SQL [31] and GSR [2] also adopt similar methods.

Hybrid Approaches. Multiple strategies can be combined in a hybrid approach to improve their effectiveness. For example, RSL-SQL [5] combines a zero-shot schema filtering and a preliminary SQL-based method, thus achieving a bidirectional schema linking design. CHESS$_{(IR,SS,CG)}$ [60] also employs a hybrid approach that combines few-shot CoT and schema pruning.

Horizontal Comparison. Few-shot CoT has low overhead and adapts quickly but is sensitive to exemplar coverage and phrasing. Multi-stage pruning performs well and is robust under ambiguity, but it is very token-intensive and increases latency. Preliminary-SQL is token-efficient and precise when the draft is reliable but degrades with draft errors or drift.

2.3 Candidate Generation Module

The Candidate Generation module tasked with converting NL queries into candidate SQL statements. This module directly interprets user intent and produces the corresponding SQL queries that align with the structure and constraints of the database schema. We describe the representative strategies for this module as follows and present the classification of existing works in Table 2.

Few-Shot CoT. By explicitly incorporating intermediate reasoning steps in few-shot NL2SQL generation examples, few-shot CoT methods help LLMs process and solve complex logical processes in NL2SQL. For example, DIN-SQL [48] employs human-designed CoT frameworks for different types of NL questions.

Query Classification. This strategy classifies NL queries into different classes according to their complexity and uses different prompts for each class. DIN-SQL [48] is a representative example that employs a classification strategy to generate SQL candidates.

Query Decomposition. The decomposition-based strategy extends the CoT approach by decomposing a problem into smaller, manageable components [69]. The decomposer agent introduced by MAC-SQL [61] breaks the query into a series of intermediate steps, such as sub-questions, and generates corresponding sub-queries for each step before generating the final SQL.

Intermediate Representation. To bridge the gap between NL and SQL queries, intermediate representations (IRs) [14, 17, 26], such as Pandas-like or SQL-like codes, have been introduced to facilitate the generation of SQL queries. TA-SQL [53] employs Pandas-like code as an IR, DIN-SQL adopts the IR from NatSQL [14] while OpenSearch-SQL invents an SQL-like language to encourage LLMs to focus more on logic before generating final SQL queries.

Multiple Candidate Generation. Some recent works choose to increase LLM calls or sampling numbers and generate multiple candidates before selecting the final answers among them. This strategy has been implemented in studies such as C3-SQL [11], CHESS [60], MCS-SQL [21] and OpenSearch-SQL [68].

Hybrid Approaches. Many works mentioned above also adopt multiple strategies to enhance performance.

Horizontal Comparison. Few-shot prompting is efficient but brittle to template and phrasing shift. Query classification improves alignment at modest cost but risks misrouting. Decomposition increases controllability and traceability but adds steps and propagates errors. Intermediate representations stabilize structure and reduce surface variance but may under-express rare constructs and add translation burden. Multi-candidate generation broadens coverage and hedges uncertainty but raises computation and depends on reliable selection.

Table 2: Classification of Candidate Generation strategies.

Approach	Few-shot CoT	Classification	Decomposition	Inter Represent	Multi Candidate
C3-SQL [11]	✗	✗	✗	✗	✓
DIN-SQL [48]	✓	✓	✗	NatSQL	✗
MAC-SQL [61]	✓	✗	✓	✗	✗
CHESS$_{(IR,SS,CG)}$ [60]	Zero-Shot	✗	✗	✗	✗
CHESS$_{(IR,CG,UT)}$ [60]	Zero-Shot	✗	✗	✗	✓
TA-SQL [53]	✗	✗	✗	Pandas-Like	✗
OpenSearch-SQL [68]	✓	✗	✗	SQL-Like	✓
RSL-SQL [5]	Zero-Shot	✗	✗	✗	✓
E-SQL [4]	✓	✗	✗	✗	✗
PET-SQL [31]	✓	✗	✗	✗	✓

2.4 Query Revision Module

This module refines candidate SQL queries to produce the final query. Its tasks involve correcting SQL syntax errors, refining the query based on execution results, and selecting the best query from the multiple candidates generated. We survey the representative strategies as follows and summarize the classification of existing works based on these strategies in Table 3.

LLM-Based. Similar to Self-Refine [40], generally, the process begins by presenting the candidate SQL query to LLMs alongside the original natural language query. In a zero-shot setting, the model is explicitly instructed to review the SQL query for potential issues, such as syntax errors, semantic misalignments, or missing components, and produce a corrected SQL query. DIN-SQL [48] employs this strategy effectively by integrating a self-correction module within its pipeline.

Execution-Guided. This strategy leverages the execution results of candidate SQL queries as a critical feedback mechanism to improve the accuracy.. This process can be iterative, allowing multiple execution rounds, feedback analysis, and refinement until predefined termination conditions. MAC-SQL [61] employs a dedicated Refiner Agent that automates the error detection and correction process. Similarly, CHESS [60] starts with an initial draft query and executes it to gather feedback. The execution results, including error messages or outputs, are provided to the LLM, which adjusts and refines the SQL query accordingly.

Consistency-Based. Following the principle of self-consistency [63], this strategy generates multiple SQL queries through diverse reasoning paths, evaluates their execution results, and selects the most consistent query as the final output. C3-SQL [11] incorporates a Consistency Output module, where multiple SQL queries are executed and filtered, and a voting mechanism is applied to the execution results to identify the most consistent query. CHESS [60] selects the most consistent SQL query from three samples.

Unit-Test-Based. This strategy first generates multiple unit tests to highlight the differences between the candidate queries. Then, it selects the best query based on the evaluation results of the unit tests. To the best of our knowledge, CHESS$_{(IR,CG,UT)}$ is the first approach that introduces this novel strategy.

Question Rewriting. This strategy aims to better guide LLMs in revising and constructing SQL queries by enriching or rewriting the natural language questions. For example, E-SQL [4] enriches the questions by incorporating relevant database items, candidate predicates, and SQL generation steps. DART-SQL [41] utilizes database content to clarify and disambiguate questions through rewriting.

Ranking and Selection. Given multiple candidate queries, this strategy ranks the candidate queries based on some criteria and then selects the top-1 query from the query pool. A representative approach employing this method is CHASE-SQL [47], which fine-tunes a model specialized for ranking and selecting queries.

Hybrid Approaches. Similar to the Candidate Generation module, many works combine multiple of the above strategies to achieve better results. We present the detailed classification in Table 3.

Horizontal Comparison. Self-refine is lightweight but prone to superficial edits and regressions. Execution-guided repair addresses syntax and missing objects but adds iterations and cannot resolve silent semantic mismatches. Consistency-based selection reduces randomness and outliers but consumes sampling budget and may amplify shared biases. Unit-test-based checks deliver strong disambiguation but require specification effort and are sensitive to test quality. Question rewriting clarifies constraints and schema cues but can introduce bias or leakage. Ranking and selection consolidate decisions but may misrank under shift and add scoring overhead.

Table 3: Classification of Query Revision strategies.

Approach	LLM Based	Execution Guided	Consistency Based	Unit Test	Rank Select	Question Rewrite
C3-SQL [11]	✗	Single-Turn	✓	✗	✗	✗
DIN-SQL [48]	✓	✗	✗	✗	✗	✗
MAC-SQL [61]	✗	Up to 3	✗	✗	✗	✗
CHESS$_{(IR,SS,CG)}$ [60]	✗	Single-Turn	✓	✗	✗	✗
CHESS$_{(IR,CG,UT)}$ [60]	✗	Single-Turn	✗	✓	✗	✗
OpenSearch-SQL [68]	✗	Single-Turn	✓	✗	✗	✗
RSL-SQL [5]	✗	Up to 5	✗	✗	✓	✗
E-SQL [4]	✗	✓	✗	✗	✗	✓
DART-SQL [41]	✗	✓	✗	✗	✗	✓
GSR [2]	✗	✓	✗	✗	✗	✗
PET-SQL [31]	✗	Single-Turn	✓	✗	✗	✗
CHASE-SQL [47]	✗	✓	✗	✗	✓	✗

3 NL2SQLBENCH: BENCHMARKING AND EVALUATION FRAMEWORK

We now introduce **NL2SQLBench**, our modular benchmarking and evaluation framework for LLM-enabled NL2SQL approaches. We first propose fine-grained evaluation metrics tailored to each module, then present our multi-agent benchmarking framework for NL2SQL approaches, thus facilitating a comprehensive experimental analysis and providing deeper insights into their effectiveness, efficiency, and potential areas for improvement.

3.1 Evaluation Metrics

We propose a set of metrics for each module, enabling a modularized and fine-grained evaluation of NL2SQL systems. By isolating the performance of individual modules, we facilitate a more nuanced understanding of their effectiveness and efficiency.

3.1.1 Metrics for Schema Selection. To evaluate the effectiveness of the Schema Selection module, we propose three metrics: *Precision*, *Recall*, and *F1-score*, for both table and column retrieval. These metrics provide a standardized way of assessing the module's ability to select relevant schema elements while minimizing irrelevant selections. Let R represent the set of relevant schema elements, and

S represent the set of elements selected by the system. We define:

$$\text{Precision} : \mathrm{P} = |R \cap S| \,/\, |S| \tag{1}$$

$$\text{Recall} : \mathrm{R} = |R \cap S| \,/\, |R| \tag{2}$$

$$\text{F1-score} : \mathrm{F}_1 = 2 \cdot P \times R \,/\, (P + R) \tag{3}$$

Precision measures the proportion of relevant elements retrieved, while Recall assesses the system's ability to identify all relevant schema elements. A high F1-score reflects a balance between both. These metrics are calculated separately at the table and column levels to evaluate schema selection performance.

3.1.2 Metrics for Candidate Generation. Evaluating the effectiveness of this module is critical, as its output directly influences downstream processes such as query revision. Existing benchmarking, such as Spider [72] and BIRD [28], proposed the *Execution Accuracy* (EX) metric as the proportion of examples for which the executed results of both the predicted and ground-truth SQLs are identical, relative to the overall number of test cases.

To provide a comprehensive evaluation, we propose more fine-grained metrics: *Correct Rate* (CR)[2], *Incorrect Rate* (IR), and *Error Rate* (ER), to evaluate a generated SQL query by assessing whether it produces (1) a Correct query (a correct result), (2) an Incorrect query (an incorrect result without runtime execution errors), or (3) an Error query (a runtime execution error), respectively.

Let C be the set of correct SQL queries, I be the set of Incorrect queries that execute without runtime errors, E be the set of queries that produce execution errors, and Q be the total number of queries generated. We define:

$$\text{Correct Rate} : \mathrm{CR} = |C| \,/\, |Q| \tag{4}$$

$$\text{Incorrect Rate} : \mathrm{IR} = |I| \,/\, |Q| \tag{5}$$

$$\text{Error Rate} : \mathrm{ER} = |E| \,/\, |Q| \tag{6}$$

These metrics satisfy $C \cup I \cup E = Q$ and C, I, E are pairwise disjoint, ensuring that each generated query falls into exactly one category and that CR + IR + ER = 1.

For SQL queries producing runtime execution errors, we classify them into the most common categories based on error messages and failure types such as *no such table/column*, *no such function*, *syntax error*, *timeout*, and *other error types*. This fine-grained categorization provides additional diagnostic insights into error analysis.

Furthermore, we adopt `Pass@k` for approaches producing multiple candidate queries, similar to CHESS [60]. Specifically, `Pass@k` measures the rate of producing at least one correct SQL query among `k` SQL queries produced per NL question. Formally,

$$\texttt{Pass@k} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}\Big[\exists\, j \le k : \mathrm{Correct}\big(q_{i,j}\big) = 1\Big] \tag{7}$$

where the parameters are defined as follows.

- N: Total number of NL queries.
- k: Number of candidate SQL queries produced per NL query.
- $q_{i,j}$: The j^{th} candidate query generated for the i^{th} NL query.
- $\mathrm{Correct}(\cdot)$: Boolean function that returns 1 if a candidate query yields the correct execution result, and 0 otherwise.
- $\mathbb{I}(\cdot)$: Indicator function: 1 if the argument is true, 0 otherwise.

This metric is mainly to measure the upper limit of the achievable accuracy when producing multiple candidates, thus providing important references for evaluating the Candidate Generation module and the Query Revision module.

3.1.3 Metrics for Query Revision. To comprehensively evaluate the Query Revision module, we use the same primary metrics as for Candidate Generation—Correct Rate, Incorrect Rate, and Error Rate with detailed error categorization. Additionally, we introduce new metrics—Correctness Improvement Rate (CI), Incorrect-to-Correct Rate (I2C), Error-to-Correct Rate (E2C), Correct-to-Incorrect Rate (C2I), and Correct-to-Error Rate (C2E)—to specifically measure this module's ability to enhance query quality and resolve errors without introducing regressions.

Let CR_{pre} and CR_{post} denote the Correct Rates before and after Query Revision. Similarly, let C_{pre} and C_{post} represent the Correct query sets, I_{pre} and I_{post} represent the Incorrect query sets, and E_{pre} and E_{post} represent the Error query sets before and after Query Revision, respectively. Then, we define:

$$\mathrm{CI} = \big(CR_{\text{post}} - CR_{\text{pre}}\big) \,/\, CR_{\text{pre}} \tag{8}$$

$$\mathrm{I2C} = |I_{\text{pre}} \cap C_{\text{post}}| \,/\, |I_{\text{pre}}| \tag{9}$$

$$\mathrm{E2C} = |E_{\text{pre}} \cap C_{\text{post}}| \,/\, |E_{\text{pre}}| \tag{10}$$

$$\mathrm{C2I} = |C_{\text{pre}} \cap I_{\text{post}}| \,/\, |C_{\text{pre}}| \tag{11}$$

$$\mathrm{C2E} = |C_{\text{pre}} \cap E_{\text{post}}| \,/\, |C_{\text{pre}}| \tag{12}$$

Among the new metrics, CI, I2C, and E2C capture improvements in query quality, while C2I and C2E identify potential regressions. Together with the primary metrics from the Candidate Generation module, these measures enable a comprehensive assessment of the module's strengths and weaknesses.

3.1.4 Efficiency Metrics. Although an increasing number of novel solutions have achieved impressive performance on NL2SQL leaderboards, most overlook computational cost and latency, which are critical for real-world deployment [34]. This oversight significantly limits their deployment in real-world environments with strict constraints on resources, budgets, and real-time responsiveness.

To bridge this critical gap, we propose explicitly measuring the efficiency of individual NL2SQL modules using two metrics: *#Tokens* (the total number of tokens for prompts and completions) and *#LLM Calls* (the number of LLM invocations)[3]. By systematically quantifying these metrics, our framework enables a detailed evaluation of computational overhead, thereby fostering the development of NL2SQL solutions practically viable for widespread adoption.

3.2 Multi-Agent Benchmarking Framework

To facilitate the benchmarking, we develop a multi-agent benchmarking framework, as shown in the bottom half of Figure 1, comprising three specialized agents:

Result Collecting Agent. This agent collects the results required for evaluation metrics from each module. It gathers selected schemas, candidate SQL queries, and refined SQL queries from the Schema Selection, Candidate Generation, and Query Revision modules, respectively. It also collects LLM usage statistics, such as token costs

[2]The Correct Rate is equivalent to the Execution Accuracy used by Spider and BIRD.

[3]For the methods utilizing self-consistency on n outputs, we compute *#Tokens* and *#LLM Calls* by invoking the language model n times

and the number of LLM calls. All the collected information is consolidated into structured JSON files. We provide an example of the collected JSON format in Appendix A.1 of our extended paper [19].

Postprocessing Agent. This agent transforms raw outputs from upstream modules into a standardized format for benchmarking. Its main functions include extracting gold schema from gold SQL queries as reference for schema selection accuracy, executing both candidate and refined SQL to compare results with gold SQL for correctness and error identification, and serializing evaluation outputs into structured JSON files for further analysis. Additionally, the agent compiles performance statistics, reporting total executions, success and failure counts, and per-question summaries.

Benchmarking Agent. Utilizing the processed datasets, this agent performs a comprehensive and modular evaluation of each module. Specifically, it leverages the previously defined evaluation metrics to compute quantitative performance indicators for each module. For each module, the agent aggregates relevant statistics and analyzes these metrics across varying difficulty levels of questions, enabling a granular assessment. The resulting evaluations are recorded in a structured format, providing detailed performance breakdowns that support comparative analysis across different strategies.

NL2SQLBench is designed for broad adaptability across NL2SQL systems, LLMs, and evaluation datasets. The *Result Postprocessing* and *Benchmarking* agents function identically for all NL2SQL methods, requiring no method-specific changes. Only the *Result Collecting* agent adapts to each system, extracting and standardizing intermediate outputs with minimal wrapper code and configuration. **NL2SQLBench** allows users to switch LLMs or datasets via simple configuration without architectural modifications. Future updates will introduce an Orchestration agent to automate the evaluation workflow, enabling reproducible, end-to-end benchmarking of NL2SQL pipelines with minimal scripting effort.

4 EXPERIMENTS AND EVALUATION

We now present a comprehensive, modularized benchmarking of existing NL2SQL solutions, focusing on their three core modules, using our evaluation metrics and **NL2SQLBench** framework to systematically and thoroughly analyze their effectiveness and efficiency.

4.1 Experiment Settings

Datasets. To ensure a rigorous and comprehensive evaluation of NL2SQL approaches, we utilize the BIRD dataset [28], a large-scale and cross-domain dataset specifically designed to assess the capabilities of NL2SQL systems across diverse and complex scenarios, and the ScienceBenchmark dataset [76], which contains three real-world, highly domain-specific databases[4].

Evaluated Approaches. We choose a list of representative and open-sourced approaches[5] from the BIRD leaderboard: C3-SQL [11], DIN-SQL [48], MAC-SQL [61], TA-SQL [53], CHESS$_{(IR,SS,CG)}$ [60], CHESS$_{(IR,CG,UT)}$ [60], E-SQL [4], RSL-SQL [5], OpenSearch-SQL [68] and GSR [2]. These approaches cover diverse strategies for the three core modules mentioned previously.

Language Models. To ensure a fair and consistent comparison across all approaches while minimizing the experimental costs, we utilize DeepSeek-V3 [9] and GPT-4o-mini [45] as the unified LLMs for all evaluations[6]. For model-related hyperparameters such as temperature, maximum tokens, and prompts, we retain the original configurations used by each evaluated method to ensure objectivity in evaluation. For the approaches that require models for embedding and retrieval, we choose the model `bge-large-en-v1.5` [67].

4.2 Results for Schema Selection Module

Table 4: Analysis results of Schema Selection on BIRD using DeepSeek-V3. The best results are in bold and underlined while the second-best results are underlined.

Approach	Table Selection			Column Selection			Efficiency	
	Precision (%)	Recall (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)	#Tokens	#LLM calls
C3-SQL	50.03	98.80	64.63	27.88	95.03	41.76	15886	20
DIN-SQL	93.27	95.47	93.41	89.10	88.76	**88.01**	7360	**1**
MAC-SQL	32.74	**99.84**	46.70	15.84	**98.70**	26.12	**3179**	**1**
CHESS$_{(IR,SS,CG)}$	**94.50**	95.85	**94.35**	**89.78**	88.00	87.66	307894	78.56
TA-SQL	93.66	95.73	93.77	82.46	90.77	84.89	4249	**1**
RSL-SQL	88.53	97.09	91.21	55.03	93.01	63.37	5790	**1**
OpenSearch-SQL	32.80	99.62	46.72	16.97	97.35	27.49	6698	3

4.2.1 Overall performance. Table 4 reports the overall results of the schema selection module on the BIRD with DeepSeek-V3. The results of the remaining three settings are deferred to the Appendix A.2 in our extended paper [19]. Across all four settings, we observe highly consistent rankings.

Table selection. As whoen in Table 4, CHESS$_{(IR,SS,CG)}$ and TA-SQL achieved the top two F1 scores, followed closely by DIN-SQL and RSL-SQL. This pattern largely persists when swapping the LLM and switching to ScienceBenchmark, with minor shifts. These consistent outcomes show the effectiveness of their Multi-Stage Schema Pruning (as in CHESS$_{(IR,SS,CG)}$) and Preliminary-SQL Enhanced strategies (as in TA-SQL). Few-Shot CoT (DIN-SQL) also delivers strong, often third-best F1, suggesting that lightweight reasoning with carefully curated exemplars can narrow the gap to heavier multi-stage pipelines.

Column selection. On BIRD, DIN-SQL leads the column-level F1, with CHESS and TA-SQL close behind; On ScienceBenchmark, leadership alternates among CHESS and TA-SQL, with DIN-SQL remaining competitive. Compared to BIRD, ScienceBenchmark tends to depress column-selection precision, leading to lower column-F1 across systems. We conjecture this stems from the fact that the schema complexity of ScienceBenchmark datasets is higher than BIRD. Indeed, methods such as MAC-SQL and OpenSearch-SQL often attain very high recall at both table and column levels but suffer from low precision.

***Insight 1**: Multi-Stage Schema Pruning, Pre-SQL cues, and Few-Shot CoT reasoning are consistently effective for both table and column selection. However, their gains depend on careful implementation details. The predominant error is over-selection, especially on complex*

[4] The evaluation is conducted on the dev Set of BIRD and ScienceBenchmark, containing 1534 and 299 test cases respectively.

[5] Due to computational budget, we do not evaluate fine-tuned or RL-trained models

[6] We use the DeepSeek-V3-0324 Release version.

schemas (ScienceBenchmark). This suggests adding precision-oriented controls to curtail long tails of spurious columns.

Insight 2: *Relative rankings are stable across LLMs and datasets, with only mild re-ordering at the top. This robustness indicates the observed advantages stem from algorithmic choices rather than opportunistic synergy with a particular LLM or dataset.*

4.2.2 Cost efficiency and performance on large-scale databases. Across settings, we observe a clear cost–quality frontier. CHESS$_{(IR,SS,CG)}$ attains top table/column selection but incurs substantial token cost and LLM calls pose significant barriers to deployment, where latency and cost matter. By contrast, TA-SQL and DIN-SQL deliver near-top F1 with a single call and low tokens, offering a favorable accuracy–efficiency trade-off.

These trends persist on ScienceBenchmark, which better approximates real-world schema density than academic suites such as Spider [72] and BIRD [28]. On ScienceBenchmark we consistently see lower column-selection precision and stronger over-selection pressure. From a scalability standpoint, schema selection cost often grows with the number of tables/columns included in prompts or retrieval stages. This underscores the need for future research into lightweight pruning strategies that maintain accuracy while minimizing computational overhead.

Insight 3: *Multi-stage pipelines (e.g., CHESS) yield the highest accuracy but at high token/call budgets; single-call pre-SQL/CoT approaches (e.g., TA-SQL, DIN-SQL) provide the strongest production-ready trade-off. ScienceBenchmark stresses these dynamics with denser schemas, revealing precision-cost trade-off as an crucial concern for maintaining accuracy under strict cost/latency constraints.*

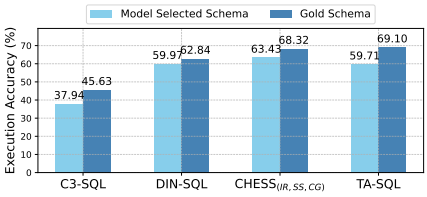


Figure 2: Execution accuracy using different schemas.

4.2.3 Influence of schema selection on overall performance. To assess the influence of accurate schema selection on the overall performance, we conducted experiments comparing execution accuracies using model-selected schemas versus gold schemas on some approaches[7] using BIRD dev set and Deepseek-V3. Figure 2 presents the execution accuracies for each approach under both conditions. The results indicate that the use of gold schemas significantly enhances execution accuracy across all evaluated approaches. Specifically, for C3-SQL, the execution accuracy increased from 37.94% to 45.63%, a significant improvement of 20.27%. TA-SQL's execution accuracy improved from 59.71% to 69.10%, the highest increase among all approaches, with a gain of 15.73%. This observation confirms the critical role of accurate schema selection – the provision of the correct schemas to the subsequent modules eliminates errors arising from incorrect schema information, which is aligned with the observation of [13, 61]. While some recent studies [4, 38] suggest that schema selection is unnecessary with today's strongest LLMs, we contend it remains indispensable for robust NL2SQL systems. Not only are cutting-edge models often inaccessible under strict compute or budget constraints, but feeding entire enterprise schemas—potentially comprising thousands of tables and columns—can exceed model context windows, increase token costs, and ultimately degrade output quality[33].

[7] Due to budget limitations, we only chose C3-SQL, DIN-SQL, CHESS, and TA-SQL.

Table 5: Results of Candidate Generation on BIRD dev with two LLMs. The best results inside each block are bold+underlined, the second-best are underlined. We use `pass@1` for methods producing multiple candidates.

Approach	DeepSeek-V3					GPT-4o-mini				
	CR (%)	IR (%)	ER (%)	#Tok.	#Calls	CR (%)	IR (%)	ER (%)	#Tok.	#Calls
C3-SQL	38.14	50.07	11.80	11786	20	35.66	51.83	12.52	1353	20
DIN-SQL	59.58	35.52	4.89	6680	2	56.65	34.49	8.87	6523	2
MAC-SQL	58.74	33.83	7.43	2829	**1**	52.74	35.92	11.34	3031	**1**
CHESS$_{(IR,SS,CG)}$	59.78	**33.12**	7.11	**1017**	**1**	53.42	41.03	**5.54**	**934**	**1**
CHESS$_{(IR,CG,UT)}$	62.91	33.12	3.98	234394	20	55.08	32.01	12.91	228177	20
TA-SQL	59.71	34.42	5.87	2592	2	53.06	33.57	13.36	2427	2
GSR	57.37	37.09	5.48	3846	2	52.67	40.29	7.04	2006	2
E-SQL	57.50	35.07	7.43	13118	**1**	**58.02**	32.07	9.91	13031	**1**
RSL-SQL	59.00	33.38	7.62	11370	5	55.54	35.01	9.45	2219	5
OpenSearch-SQL	**66.10**	**30.83**	**3.06**	100095	21	56.52	35.33	8.15	8984	21

Insight 4: *Accurate schema selection is critical in improving execution accuracy and preventing error propagation, and remains essential for robust real-world deployment due to practical constraints.*

4.2.4 Practical guide. Our analysis reveals that over-selection is prevalent, while Multi-Stage Schema Pruning achieve superior accuracy, they incur substantial computational costs that may hinder real-world deployment. To address this trade-off, we recommend implementing a filter and refine pipeline that balances effectiveness and efficiency. In the first stage, employ lightweight retrieval methods to broadly filter irrelevant schemas, prioritizing high recall to minimize the risk of excluding relevant schemas. In the second stage, leverage LLMs with carefully designed prompts to perform precision-oriented refinement on the reduced schema space, focusing on eliminating irrelevant elements that would otherwise increase token costs and degrade downstream performance.

For resource-constrained environments, Preliminary-SQL Enhanced and Few-shot CoT strategies are favored, which leverage the inherent generation capabilities of LLMs without multiple calls, offering a practical balance between accuracy and efficiency.

4.3 Results for Candidate Generation Module

4.3.1 Overall performance. We evaluate `pass@1` for comparability. Table 5 and Table 6 show that, unlike schema selection, candidate generation is highly sensitive to both the dataset and the LLM—rankings shift and there is no universal winner.

BIRD. With DeepSeek-V3, OpenSearch-SQL attains the best Correct rate (CR) with the lowest Incorrect Rate (IR), and CHESS$_{(IR,CG,UT)}$ is a close second. With GPT-4o-mini, the ordering changes: E-SQL achieves the highest CR, and simpler pipelines such as DIN-SQL and TA-SQL remain competitive at low token budgets. These shifts indicate candidate generation quality depends strongly on the LLM; the same pipeline can rank differently when the LLM changes.

Table 6: Results of Candidate Generation on ScienceBenchmark benchmark. The best results are bold+underlined and the second-best are underlined. We report pass@1 for methods producing multiple candidates.

Approach	DeepSeek-V3					GPT-4o-mini				
	CR (%)	IR (%)	ER (%)	#Tok.	#Calls	CR (%)	IR (%)	ER (%)	#Tok.	#Calls
C3-SQL	54.18	42.81	3.01	12110	20	50.84	42.81	6.35	1350	20
DIN-SQL	52.84	44.82	2.34	6149	2	42.81	48.49	8.70	5932	2
MAC-SQL	47.16	42.81	10.03	3514	**1**	44.15	41.81	14.05	3734	**1**
CHESS$_{(IR,SS,CG)}$	37.92	55.37	6.71	978	**1**	**53.42**	**41.03**	**5.54**	934	**1**
CHESS$_{(IR,CG,UT)}$	54.85	39.46	5.69	259569	20	40.80	46.82	12.37	247969	20
TA-SQL	**59.53**	38.46	**2.01**	2712	2	50.17	43.81	6.02	2523	2
GSR	40.47	47.49	12.04	**824**	2	40.47	52.17	7.36	**766**	2
E-SQL	54.52	40.47	5.02	6542	**1**	38.80	45.82	15.38	12736	**1**
RSL-SQL	45.48	39.46	15.05	2877	5	40.47	52.84	6.69	1532	5
OpenSearch-SQL	38.46	**27.09**	34.45	103392	21	42.14	46.82	11.04	8355	21

ScienceBenchmark. With DeepSeek-V3, TA-SQL yields the strongest CR at low Error Rate (ER), while breadth-oriented systems like OpenSearch-SQL see degraded CR due to increased fragility on complex schemas. With GPT-4o-mini, CHESS$_{(IR,SS,CG)}$ becomes the most effective at pass@1 with *one* call and sub-1K tokens.

Error profile. Across all four blocks, the dominant failure mode is IR: syntactically valid but *semantically misaligned* SQL substantially outnumbers execution errors. Typical IR patterns include (i) wrong join path or missing bridge table, (ii) predicate misalignment, (iii) aggregation/GROUP BY/HAVING mismatches, and (iv) projection-level omissions or extra columns. ER is non-negligible for some complex databases on ScienceBenchmark, but even there IR constitutes the majority of errors; improving *semantic fidelity* is therefore the primary lever for improving the accuracy.

Cost–efficiency. A clear cost–quality frontier emerges. Multi-candidate pipelines (e.g., OpenSearch-SQL, CHESS$_{(IR,CG,UT)}$) can lead on BIRD but often require tens of calls and very high token budgets; on ScienceBenchmark, their advantage weakens. In contrast, token-efficient pipelines (e.g., CHESS$_{(IR,SS,CG)}$, TA-SQL, sometimes DIN-SQL) achieve competitive CR at a fraction of the cost.

Insight 5: *The dominant failure mode is semantic mismatch. Adding semantic checks that align the SQL with the question will be helpful. Results shift between LLMs and datasets, indicating that performing evaluation on targeting databases and backbone LLMs before adopting specific NL2SQL approaches is necessary.*

In addition to the overall performance, we also conducted a detailed error analysis and result analysis on different difficulty level of questions. Due to page limits, we present the full results in Appendix A.3.1 and Appendix A.3.2 in our extended paper [19].

4.3.2 Pass@k *evaluation for multiple candidates.* To evaluate methods that produce multiple candidate queries, we report Pass@k for $k \in \{1, 5, 10, 15, 20\}$ across four settings: BIRD with DeepSeek-V3 and GPT-4o-mini, and ScienceBenchmark with DeepSeek-V3 and GPT-4o-mini. As illustrated in Figure 3, across all subfigures, the curves show the same shape: execution accuracy rises with larger k, confirming the benefit of generating multiple candidates. The marginal gains, however, diminish as k increases, indicating limited returns beyond a moderate number of candidates. The consistency among all subfigures indicates that our conclusion is robust to both the backbone LLM and the evaluation dataset. Since Pass@k represents an upper bound on achievable accuracy, these results suggest that a well-designed Query Revision module—capable of reliably selecting the optimal query—has the potential to elevate the accuracy close to this theoretical limit.

Insight 6: *Increasing the number of candidates raises potential accuracy across all LLMs/datasets, but the gains diminish rapidly; if the Query Revision module can consistently select the best query from the candidates, overall accuracy can nearly reach the upper bound.*

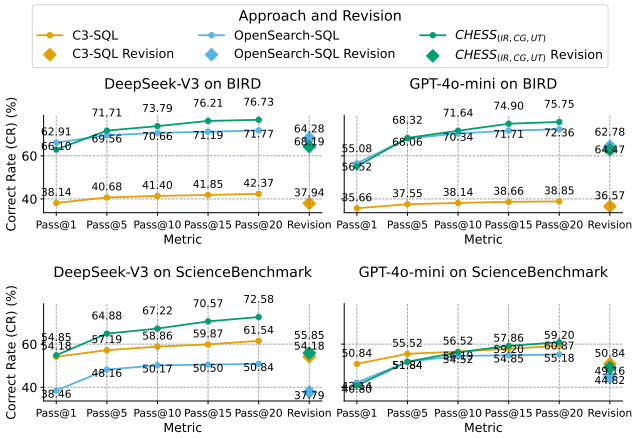


Figure 3: Pass@k results for multiple-candidate approaches.

4.3.3 Practical guide. Our evaluation demonstrates that Incorrect queries-syntactically valid but semantically misaligned queries-constitute the dominant failure case, far exceeding execution errors. This finding suggests that current candidate generation modules should reorient their validation priorities from syntax checking to semantic verification. We recommend implementing execution-result-based semantic validation during the generation phase rather than deferring all validation to the revision stage. This early detection mechanism can prevent the propagation of semantic errors to downstream modules and reduce the burden on query revision.

For systems generating multiple candidates, our Pass@k analysis reveals that increasing diversity ($k > 10$) yields diminishing returns; instead, practitioners should focus on improving the quality of top-ranked candidates through better strategies. In addition, using early-exit, and expanding to larger k only for high-uncertainty queries is a more cost-efficient approach.

4.4 Results for Query Revision Module

4.4.1 Overall performance. Figure 4 and Figure 5 compare before vs. after the Revision stage on Correct Rate (CR), Incorrect Rate (IR), and Error Rate (ER) for all approaches.

BIRD. Across most systems, Revision raises CR while reducing IR and keeping ER low. The gains are largest for pipelines that already provide reasonably good candidates: Revision acts mainly as a selector/repair step that fixes small semantic gaps and filters poor candidates. Methods that rank/select the best SQL query from multiple SQL candidates also improve, but their CR uplift depends on revision policy's ability to reliably select the optimal query.

ScienceBenchmark. The improvements are more heterogeneous across systems. Some systems still obtain clear CR gains, but

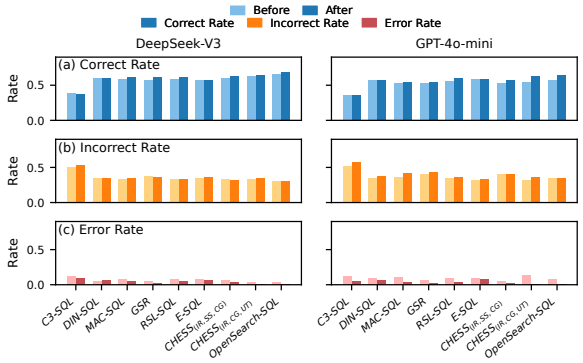


Figure 4: Analysis of the Query Revision module on BIRD.

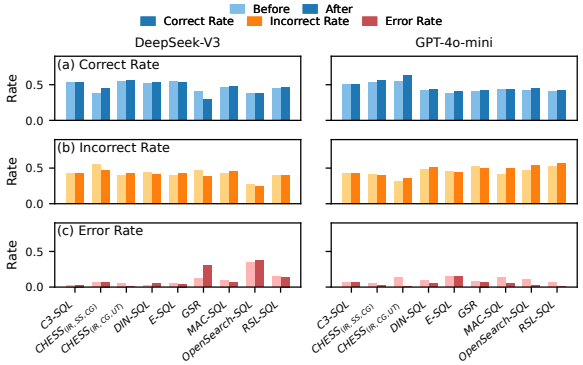
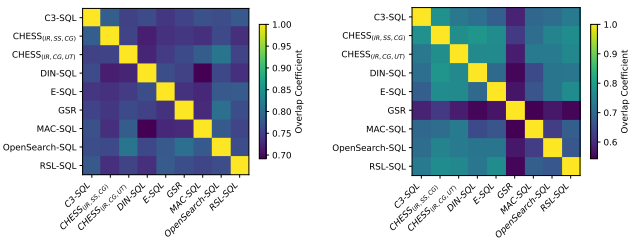


Figure 5: Analysis of the Query Revision module on ScienceBenchmark.



(a) **Overlap on the BIRD (using DeepSeek-V3)** (b) **Overlap on the ScienceBenchmark (using DeepSeek-V3)**

Figure 6: The coefficient heatmap of different solutions on Incorrect cases.

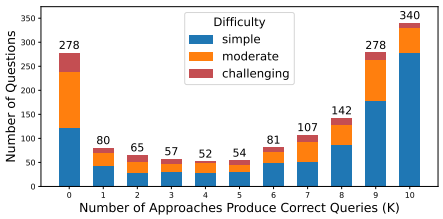


Figure 7: Breakdown of questions by number of approaches producing Correct queries (for different difficulty levels).

others see smaller gains or even a mild rise in IR. Furthermore, the extent of performance improvement varies depending on the backbone LLM employed, even when using the same NL2SQL approach. The divergance can be attributed to two compounding factors. First, the ScienceBenchmark dataset generally features more complex schemas and more challenging questions than the BIRD dataset, resulting in a more demanding evaluation of query revision strategies. Second, the query revision strategies of most NL2SQL approaches are, to some extent, specifically tailored to the BIRD dataset and therefore do not fully generalize to the ScienceBenchmark scenarios, struggling to address the unique challenges that the latter presents.

Error profile. In all settings, the predominant failure type after revision is still IR (syntactically valid yet semantically misaligned SQL), which greatly outnumbers ER. Therefore, the upper bound for further improvements in CR primarily depends on mitigating IR, while reducing ER is still necessary.

4.4.2 An in-depth analysis of incorrect cases. Through an in-depth analysis of incorrect cases, we observed an unexpectedly high degree of overlap in incorrect queries generated by different methods using DeepSeek-V3, as illustrated in Figure 6. Similar results using GPT-4o-mini are provided in the Appendix A.4.2 of our extended paper [19]. Furthermore, as shown in Figure 7, a significant portion – 278 out of the total 1,534 questions from the BIRD Dev Set – could not be correctly solved by any of the evaluated approaches (the first vertical bar, which indicates no approach can produce correct SQL queries). This substantial overlap suggests a shared set of underlying limitations across existing NL2SQL methods, potentially involving insufficient semantic understanding, inadequate query generation strategies, or ineffective query revision mechanisms.

To uncover the root causes behind this, we conducted an additional human evaluation of these 278 problematic questions. Remarkably, the results revealed that a considerable fraction of these questions were paired with inaccurate gold SQL queries, which is aligned with the observation of [35]. This discovery highlights the often overlooked but critical issue of benchmark dataset quality and raises concerns that current assessments of NL2SQL methods on the BIRD dataset may have been compromised or misled by inaccuracies inherent in the benchmark. For a more detailed analysis of the BIRD dataset, please see Section 4.6.

Insight 7: *Our analysis shows significant overlap in incorrect queries across NL2SQL approaches, highlighting shared limitations. Many unresolved errors arise from inaccuracies in the BIRD dataset's gold SQL annotations. Future work should focus on improving benchmark annotation quality to enable more reliable evaluations and meaningful progress.*

4.4.3 Detailed analysis on Query Revision performance. We evaluate the Query Revision module using Correct-Improvement (CI), Incorrect-to-Correct (I2C), and Error-to-Correct (E2C) along with efficiency (#tokens/#LLM calls). The consolidated results for BIRD and ScienceBenchmark are reported in Table 7 and Table 8, while the full details in Appendix A.4.3 of our extended paper [19].

BIRD. Using DeepSeek-V3, methods that employ with light, execution-guided strategies—notably GSR and $\text{CHESS}_{(\text{IR,SS,CG})}$—obtain the highest CI with few calls and low tokens, while still posting healthy I2C/E2C. In contrast, unit-test-enabled pipelines such as $\text{CHESS}_{(\text{IR,CG,UT})}$ achieve high I2C/E2C but only modest CI and at

Table 7: Effectiveness and efficiency on Query Revision on BIRD dev with DeepSeek-V3 vs GPT-4o-mini. We report CI/I2C/E2C and #Tokens, #LLM Calls. The best/second-best results are bold+underlined/underlined within each LLM.

Approach	DeepSeek-V3					GPT-4o-mini				
	CI (%)	I2C (%)	E2C (%)	#Tok.	#Calls	CI (%)	I2C (%)	E2C (%)	#Tok.	#Calls
C3-SQL	-0.51	1.43	4.97	–	–	2.56	2.52	8.33	–	–
DIN-SQL	0.66	2.02	21.33	4948	**1**	-0.11	4.16	19.12	4791	**1**
MAC-SQL	3.55	3.66	13.16	**1566**	2	4.45	1.45	21.84	1729	2
CHESS$_{(IR,SS,CG)}$	6.11	6.89	29.36	1844	**1**	6.72	10.49	35.29	**1683**	**1**
CHESS$_{(IR,CG,UT)}$	2.18	**15.16**	**47.54**	106126	21	14.04	16.09	38.38	115055	23.19
GSR	**7.27**	9.49	38.10	2220	2	2.60	4.85	21.3	2449	2
E-SQL	0.00	3.53	14.04	24924	2	1.91	7.52	28.29	25023	2
RSL-SQL	4.97	4.69	17.95	3315	2	9.15	10.06	35.86	2802	2
OpenSearch-SQL	3.16	8.67	34.04	6652	4.6	**14.07**	**17.71**	**51.20**	12969	6.64

Table 8: Effectiveness and efficiency on Query Revision on ScienceBenchmark dev with DeepSeek-V3 vs GPT-4o-mini. We report CI/I2C/E2C and #Tokens, #LLM Calls. The best/second-best results are bold+underlined/underlined.

Approach	DeepSeek-V3					GPT-4o-mini				
	CI (%)	I2C (%)	E2C (%)	#Tok.	#Calls	CI (%)	I2C (%)	E2C (%)	#Tok.	#Calls
C3-SQL	0	0	0	–	–	0	0	0	–	–
DIN-SQL	1.27	2.99	14.29	4349	**1**	2.34	1.38	15.38	4156	**1**
MAC-SQL	1.42	2.34	10	2395	2	0.76	3.20	9.52	2676	2
CHESS$_{(IR,SS,CG)}$	19.47	**14.55**	0	2319	**1**	**26.32**	**13.30**	14.29	1683	**1**
CHESS$_{(IR,CG,UT)}$	1.83	9.32	**52.94**	201372	23.42	20.49	12.14	**43.24**	115055	23.19
GSR	-24.8	14.08	33.33	**1757**	2	4.96	5.13	18.18	**1498**	2
E-SQL	-2.45	4.96	13.33	15088	2	5.17	6.57	4.35	21360	2
RSL-SQL	1.47	0.00	4.44	6616	2	5.79	0	35	3720	2
OpenSearch-SQL	-1.74	7.41	3.88	18575	7.17	6.35	7.14	33.33	34752	11.33

very high cost. However, under GPT-4o-mini, the ranking shifts: breadth-oriented systems (OpenSearch-SQL, CHESS$_{(IR,CG,UT)}$) show the best CI on BIRD, indicating that GPT-4o-mini responds better to selection or unit-test style prompts than DeepSeek-V3. Compact pipelines (DIN-SQL, TA-SQL, CHESS$_{(IR,SS,CG)}$) remain cost-efficient with smaller but steady CI.

ScienceBenchamrk. The results are polarized. CHESS$_{(IR,SS,CG)}$ delivers large CI with a single call and very low tokens. CHESS$_{(IR,CG,UT)}$ also posts high CI but at very high budget. Several systems—even with decent I2C/E2C—show small or negative CI, meaning they flip correct queries to wrong during revision (i.e., high C2I risk).

Error Analysis. Across all four blocks, I2C/E2C improvements do not guarantee a high CI. When revision policies are aggressive, they may "fix" incorrect/error queries but also damage originally correct ones (C2I), netting a small or negative CI, as shown in the full detailed results in Appendix A.4.3 of our extended paper [19]. This explains the divergence we observe between good I2C/E2C figures and underwhelming CI on ScienceBenchmark/DeepSeek-V3 for several methods. Since our candidate generation analysis showed that IR (semantically misaligned yet executable SQL) dominates remaining errors, revision should prioritize semantic alignment and de-risk edits on already-correct SQL.

Insight 8: *Future exploration should focus on strategies not only to correct wrong answers, but also to avoid altering originally correct queries into incorrect ones. Methods that incorporate robust "do-no-harm" safeguards for originally correct SQL can be beneficial.*

Note that even the highest CI, achieved in the Query Revision section, is just merely 26.32%, which falls significantly short of a satisfactory threshold. Moreover, methods such as CHESS$_{(IR,CG,UT)}$ entail substantial token consumption and frequent LLM invocations, rendering them impractical for real-world deployment.

Insight 9: *All of the current Query Revision strategies exhibit limited effectiveness in improving the Correct Rate, underscoring a crucial need to develop methods that are both genuinely effective—achieving significant accuracy improvements—and practically efficient, minimizing cost and latency.*

Similar to the Candidate Generation module, we conduct an additional experimental analysis for approaches that produce multiple queries. As shown in Figure 3, while Correct Rates consistently improve with increasing `Pass@k`, the final Correct Rates after Query Revision remain notably lower, even below those at `Pass@5`. This indicates that the Query Revision module currently fails to approach the upper-bound effectiveness suggested by `Pass@5`.

Insight 10: *The gap between* `Pass@k` *upper bounds and post-revision results identifies Query Revision as a key bottleneck, highlighting the need for more effective strategies to fully capitalise on the diversity of multiple candidates.*

4.4.4 Practical guide. Our analysis reveals that current Query Revision strategies achieve limited effectiveness, with the predominant failure being the inability to correct Incorrect queries that are syntactically valid but semantically misaligned. To address this critical bottleneck, we recommend a dual-strategy revision approach that combines execution-guided feedback with conservative validation, detailed explanation are presented in the Appendix A.5 of our extended paper [19] For systems with multiple-candidate generation, query selection strategies are crucial. We recommend implementing ensemble-based selection that combines multiple signals: execution success, result plausibility, query complexity metrics, and LLM-based confidence scores. Finally, for real-world deployment, given the substantial token costs of iterative revision, practitioners should implement early stopping mechanisms based on execution feedback, to avoid unnecessary costs.

4.5 End-to-End Overall Evaluation

We summarize the end-to-end overall evaluation of all approaches in Table 9 and Table 10. The financial cost per task is estimated based on the pricing guidelines of DeepSeek-V3[8] and GPT-4o-mini[9]. To simulate a realistic pricing scenario, we assume that half of the prompt tokens result in cache hits and the other half in cache misses.

BIRD. With DeepSeek-V3, OpenSearch-SQL attains the highest CR (68%), about 29 calls and 113K tokens per query. CHESS variants are similarly accurate yet even more expensive. In contrast, compact pipelines—TA-SQL, MAC-SQL, GSR—reach 59–61% CR with 1–4 calls and 6–8K tokens, yielding much lower cost. The results obtained using GPT-4o-mini are highly similar, indicates the performance of evaluated approaches are stable across LLMs.

[8] https://api-docs.deepseek.com/
[9] https://platform.openai.com/docs/pricing

Table 9: End-to-end results on BIRD dev with two LLMs. We report *Correct Rate*, efficiency (#Tokens, #LLM Calls) and *Cost*. The best/second-best results are bold+underlined/underlined within each LLM block.

Approach	DeepSeek-V3				GPT-4o-mini			
	Correct Rate (%)	#Tokens	#LLM Calls	Cost ($)	Correct Rate (%)	#Tokens	#LLM Calls	Cost ($)
C3-SQL	37.94	27673	40	0.0126	36.57	12332	40	0.0067
DIN-SQL	59.97	18988	**3**	0.0038	56.58	18511	**3**	0.0024
MAC-SQL	60.82	7574	4	0.0019	55.08	7867	4	0.0013
CHESS$_{(IR,SS,CG)}$	63.43	310756	80.6	0.0604	57.01	300061	80.5	0.0364
CHESS$_{(IR,CG,UT)}$	64.28	340601	41.8	0.0816	62.82	343232	43.2	0.0525
TA-SQL	59.71	6842	**3**	0.0017	53.06	6546	**3**	**0.0007**
GSR	61.54	**6066**	4	**0.0015**	54.04	**4455**	4	**0.0007**
E-SQL	57.50	38903	**3**	0.0074	59.13	38054	**3**	0.0047
RSL-SQL	61.93	20475	8	0.0041	60.63	10549	8	0.0013
OpenSearch-SQL	**68.19**	113446	28.6	0.0242	**64.47**	28569	10.7	0.0057

Table 10: End-to-end results on ScienceBenchmark dev with two LLMs. We report *Correct Rate*, efficiency (#Tokens, #LLM Calls) and *Cost*. The best/second-best results are bold+underlined/underlined within each LLM block.

Approach	DeepSeek-V3				GPT-4o-mini			
	Correct Rate (%)	#Tokens	#LLM Calls	Cost ($)	Correct Rate (%)	#Tokens	#LLM Calls	Cost ($)
C3-SQL	54.18	30979	40	0.0152	50.84	14179	40	0.0078
DIN-SQL	53.51	17299	**3**	0.0035	43.81	16660	**3**	0.0022
MAC-SQL	47.83	9290	4	0.0023	44.48	9577	4	0.0015
CHESS$_{(IR,SS,CG)}$	45.30	355983	92.59	0.0677	**57.01**	300061	80.5	0.0364
CHESS$_{(IR,CG,UT)}$	55.85	460941	43.42	0.1018	49.16	393208	47.6	0.0428
TA-SQL	**59.53**	7867	**3**	0.0018	50.17	7572	**3**	0.0008
GSR	30.43	**2581**	4	**0.0004**	42.47	**2264**	4	**0.0004**
E-SQL	53.18	21630	**3**	0.0044	40.80	34096	**3**	0.0042
RSL-SQL	46.15	10546	8	0.0021	42.81	11990	8	0.0018
OpenSearch-SQL	37.79	128204	31.08	0.0272	44.82	49371	15.34	0.0084

ScienceBenchamrk. With DeepSeek-V3, leadership shifts to TA-SQL (60% CR) using only 3 calls and 8K tokens. Breadth-heavy systems (CHESS$_{(IR,CG,UT)}$, OpenSearch-SQL) consume 40–90+ calls and 300K–460K tokens yet do not dominate CR. With GPT-4o-mini, CHESS$_{(IR,SS,CG)}$ achieves the best CR (57%) but at very high cost (80 calls and 300K tokens). TA-SQL retains strong cost efficiency. The ranking shift indicates there is no universal winner on ScienceBenchmark dataset: the best method depends on both the dataset and the backbone LLM: the best method depends on backbone LLM.

Insight 11: *End-to-end rankings are backbone and dataset-dependent; validate on the target database(s) with the intended LLM before adoption is necessary.*

Cost-efficiency While OpenSearch-SQL, CHESS$_{(IR,CG,UT)}$ and CHESS$_{(IR,SS,CG)}$ achieve superior accuracy on some workloads, they incur substantial token costs and LLM invocations. Recent studies, such as CHASE-SQL [47] and MCS-SQL [21] also shown outstanding performance by increasing LLM calls to generate a large pool of candidate queries. However, we believe that while increasing the number of LLM calls can improve accuracy, this approach may lack adaptability in real-world deployment.

Insight 12: *While scaling LLM calls and token usage can enhance accuracy on some datasets, such strategies may undermine scalability and cost-effectiveness. Future work should aim to achieve high accuracy without proportional growth in token consumption.*

4.5.1 Practical guide. We recommend reporting CR@*budget* (e.g., CR at a fixed call or token cap) and selecting methods along the Pareto frontier instead of by CR alone. To achieve high accuracy without huge token consumption, we recommend implementing adaptive module selection that dynamically adjusts computational strategies based on query characteristics and system confidence signals. See Appendix A.5 of our extended paper [19] for full details.

4.6 New Discoveries on Existing Datasets

We conduct an in-depth analysis on BIRD and identified three key limitations that compromise its reliability as below. Detailed examples are provided in the Appendix A.6 of our extended paper [19].

4.6.1 Inaccurate Gold SQL Queries. As shown in Section 4.4, some NL queries in BIRD are paired with the wrong Gold SQL queries. Although [35] recently identified 106 incorrectly annotated queries in the BIRD dataset, their results still haven't exhaustively covered all annotation errors. The prevalence of such annotation errors raises concerns about the reliability and validity of existing datasets.

4.6.2 Strict Evaluation Rules. The BIRD benchmark currently assesses each NL question against a single gold SQL query, an assumption that proves overly restrictive in many realistic scenarios [13]. This over-strictness calls for more flexible and semantically aware evaluation rules—such as result-set equivalence or graded relevance metrics—to foster fairer assessments of future NL2SQL solutions.

4.6.3 Semantic Ambiguity. Existing datasets inevitably contain questions that exhibit a certain degree of semantic ambiguity [13]. Mitigating the effect of such ambiguity will require either (i) rewriting questions to eliminate ambiguity or (ii) devising evaluation rules that recognize ambiguity and reward semantically equivalent answers. Developing formal methods to define, detect, and quantify semantic ambiguity remains an open problem and constitutes a promising avenue for future research.

Insight 13: *Systematic auditing of gold annotations, adoption of evaluation rules that acknowledge multiple semantically equivalent queries, and explicit modeling of semantic ambiguity are crucial for reliable assessment and methodological progress. Tackling these issues, notably present in the BIRD dataset, will enable more accurate evaluations and advance robust NL2SQL solutions.*

4.6.4 Practical guide. We recommend developing semantically-aware evaluation frameworks that recognize and credit semantically equivalent but syntactically different SQL queries. Such improvements are vital for fair benchmarking and for training NL2SQL models that truly understand semantics rather than replicating annotation errors from noisy datasets. Detailed recommendations are provided in the Appendix A.6 of our extended paper [19].

4.7 Leveraging NL2SQLBench for NL2SQL System Development

The modular architecture of **NL2SQLBench** enables practitioners to adopt a **compositional optimization** approach to NL2SQL development, analogous to assembling building blocks where each module can be independently diagnosed, optimized, and composed. To leverage this capability effectively, we recommend the following systematic workflow. First, establish a performance profile by running **NL2SQLBench** with different baselines on your target dataset to generate a comprehensive performance profile across all three core modules. This profile identifies the primary bottleneck. Second, apply targeted optimizations to the bottleneck module while monitoring cross-module impacts. Third, conduct cost-benefit analysis at the module level using our effectiveness and efficiency metrics. Finally, leverage **NL2SQLBench** for A/B testing and continuous improvement. By treating NL2SQL as a modular optimization problem where each component can be independently analyzed, improved, and composed, rather than a monolithic system design challenge, practitioners can systematically navigate the accuracy-efficiency trade-off space and construct systems tailored to their specific deployment requirements. We provide a case study of leveraging **NL2SQLBench** in Appendix A.7 of our extended paper [19].

5 RELATED WORK

LLM-based NL2SQL. Since LLMs have demonstrated distinctive emergent abilities [64], LLM-based NL2SQL methods have become the most prominent solutions in the current NL2SQL landscape, as described in recent surveys [18, 34, 44, 58, 80]. Detailed discussion are presented in the Appendix A.9 of our extended paper [19].

NL2SQL Datasets and Dataset Evaluation. Several datasets have been proposed recently to facilitate the development and evaluation of NL2SQL solutions. WikiSQL [79] and Spider [72] published several cross-domain datasets, which mainly contain lightweight databases and focus on database schema. BIRD [28] focuses on large databases and external knowledge, and has become a widely used NL2SQL dataset nowadays. ScienceBenchmark [76] is a highly domain-specific NL2SQL dataset in the field of scientific data analysis, which contains real-world complex queries and large databases. Recently proposed Spider-2.0 [22] provides a more realistic enterprise-level NL2SQL benchmark, encompassing multiple database systems, diverse SQL dialects, and numerous challenging tasks from real data engineering pipelines. In the line of related work that evaluates NL2SQL datasets, Mitsopoulou and Koutrika proposed a comprehensive analysis of existing NL2SQL datasets, and provided valuable insights into their capabilities and limitations, and how they affect training and evaluation of NL2SQL systems [43]. Liu et al. [35] further developed a benchmark for detecting and categorizing semantic errors in NL2SQL. Yang et al. [71] presents a method for detecting and fixing the errors in NL2SQL translation.

NL2SQL System Evaluations. Several experimental studies have been proposed to evaluate NL2SQL solutions. For example, Chang et al. [6] evaluated the robustness of different NL2SQL models. Pourreza and Rafiei [49] studied the limitations of existing evaluation metrics and conducted a benchmark through human evaluation and query rewriting. Gao et al. [15] undertook an assessment examining multiple prompting strategies and fine-tuning methods. Zhang et al. [74] evaluated the performance of different LLMs in sub-tasks of NL2SQL pipeline. Li et al. [24] proposed a testbed for evaluating NL2SQL systems from different perspectives.

Unlike previous efforts, our work, **NL2SQLBench**, is the first that introduces a set of fine-grained evaluation metrics for each module of the NL2SQL pipeline, develops a benchmarking framework, and conducts a comprehensive benchmarking of diverse strategies at the modular level. We summarize the differences between **NL2SQLBench** and existing NL2SQL evaluation frameworks in Table 11 .

Table 11: Comparison of evaluation frameworks for NL2SQL datasets and systems.

Benchmarking Framework	Evaluation Objectives	E2E Eval	Modular Eval	Fine-grained Metrics	Error Analysis	Practical Guides
Text2sql Benchmarks [43]	Datasets	–	–	–	**Yes**	–
SQLDriller [71]	Datasets	–	–	–	**Yes**	–
NL2SQL-BUGs [35]	Datasets	–	–	–	**Yes**	–
DR-Spider [6]	Systems	**Yes**	No	No	**Yes**	Partial
Cross-Domain Text2sql [49]	Systems	**Yes**	No	No	**Yes**	Partial
Text2sql by LLMs [15]	Systems	**Yes**	No	No	No	Partial
Benchmarking Text2sql [74]	Systems	**Yes**	No	No	**Yes**	Partial
NL2SQL360 [24]	Systems	**Yes**	No	No	No	Partial
NL2SQLBench (ours)	Systems	**Yes**	**Yes**	**Yes**	**Yes**	**Detailed**

6 LIMITATIONS

Limited Scope of Evaluated Approaches. Our evaluation encompasses a selection of representative NL2SQL approaches from the BIRD leaderboard. However, due to code availability constraints and limited computational resources, not all existing NL2SQL methods were included. For instance, the studies [8, 16, 25, 27, 47, 51, 56] that fine-tune specialized models for specific tasks within the NL2SQL pipeline were excluded from our benchmarking. Future work should incorporate a broader array of approaches to enable a more comprehensive evaluation of the current landscape of NL2SQL systems.

Lack of Industry-Level Evaluation. Current benchmarks may not fully reflect the complexities of real-world industry workloads, including schema intricacy and query ambiguity. Future research should incorporate more realistic, industry-level evaluations to better capture practical deployment challenges.

7 CONCLUSIONS

We have systematically addressed the critical need for a unified, modular evaluation of LLM-enabled NL2SQL approaches. Specifically, we conducted a comprehensive review of the three core modules *Schema Selection*, *Candidate Generation*, and *Query Revision*. We proposed a novel set of fine-grained metrics and developed **NL2SQLBench**, a modular, multi-agent benchmarking framework. Our evaluation of representative NL2SQL approaches highlights substantial opportunities for improvement in both accuracy and computational efficiency Our in-depth analysis exposed critical shortcomings in current benchmark datasets and evaluation rules, underscoring the urgency for developing more accurate, robust, and standardized evaluation resources. By establishing this foundational benchmarking framework, our work aims to provide useful insights, practical guides, for future NL2SQL development, ultimately benefiting both academic research and enterprise applications.

REFERENCES

[1] Anastasia Ailamaki, Samuel Madden, Daniel Abadi, Gustavo Alonso, Sihem Amer-Yahia, Magdalena Balazinska, Philip A Bernstein, Peter Boncz, Michael Cafarella, Surajit Chaudhuri, et al. 2025. The Cambridge Report on Database Research. *arXiv preprint arXiv:2504.11259* (2025).

[2] Anonymous. 2025. LLM Prompting for Text2SQL via Gradual SQL Refinement. In *Submitted to ACL Rolling Review - February 2025.* https://openreview.net/forum?id=ozUJOijVTJ under review.

[3] Arian Askari, Christian Poelitz, and Xinye Tang. 2025. Magic: Generating self-correction guideline for in-context text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 23433–23441.

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

[5] Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, and Wei Chen. 2024. RSL-SQL: Robust Schema Linking in Text-to-SQL Generation. *arXiv preprint arXiv:2411.00073* (2024).

[6] Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Yang Wang, Zhiguo Wang, Vittorio Castelli, Patrick Ng, and Bing Xiang. 2023. Dr.Spider: A Diagnostic Evaluation Benchmark towards Text-to-SQL Robustness. In *ICLR*.

[7] Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. 2024. Navigate through Enigmatic Labyrinth A Survey of Chain of Thought Reasoning: Advances, Frontiers and Future. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 1173–1203. https://doi.org/10.18653/v1/2024.acl-long.65

[8] Team Cohere, Arash Ahmadian, Marwan Ahmed, Jay Alammar, Milad Alizadeh, Yazeed Alnumay, Sophia Althammer, Arkady Arkhangorodsky, Viraat Aryabumi, Dennis Aumiller, et al. 2025. Command A: An enterprise-ready large language model. arXiv:2504.00698 [cs.CL] https://arxiv.org/abs/2504.00698

[9] DeepSeek-AI. 2025. DeepSeek-V3-0324 Release. https://api-docs.deepseek.com/news/news250325.

[10] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. 2024. A Survey on In-context Learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 1107–1128. https://doi.org/10.18653/v1/2024.emnlp-main.64

[11] Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, lu Chen, Jinshu Lin, and Dongfang Lou. 2023. C3: Zero-shot Text-to-SQL with ChatGPT. arXiv:2307.07306 [cs.CL] https://arxiv.org/abs/2307.07306

[12] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining Small Language Models and Large Language Models for Zero-Shot NL2SQL. *Proc. VLDB Endow.* 17, 11 (July 2024), 2750–2763. https://doi.org/10.14778/3681954.3681960

[13] Avrilia Floratou, Fotis Psallidas, Fuheng Zhao, Shaleen Deep, Gunther Hagleither, Wangda Tan, Joyce Cahoon, Rana Alotaibi, Jordan Henkel, Abhik Singla, Alex Van Grootel, Brandon Chow, Kai Deng, Katherine Lin, Marcos Campos, K. Venkatesh Emani, Vivek Pandit, Victor Shnayder, Wenjing Wang, and Carlo Curino. 2024. NL2SQL is a solved problem... Not!

[14] Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. 2021. Natural SQL: Making SQL Easier to Infer from Natural Language Specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Punta Cana, Dominican Republic, 2030–2042. https://doi.org/10.18653/v1/2021.findings-emnlp.174

[15] 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. *Proc. VLDB Endow.* 17, 5 (January 2024), 1132–1145. https://www.vldb.org/pvldb/vol17/p1132-gao.pdf

[16] Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, Jinyang Gao, Liyu Mou, and Yu Li. 2024. A Preview of XiYan-SQL: A Multi-Generator Ensemble Framework for Text-to-SQL. *arXiv preprint arXiv:2411.08599* (2024). https://arxiv.org/abs/2411.08599

[17] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Anna Korhonen, David Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, Florence, Italy, 4524–4535. https://doi.org/10.18653/v1/P19-1444

[18] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. Next-Generation Database Interfaces: a Survey of LLM-based Text-to-SQL. *IEEE Transactions on Knowledge and Data Engineering* (2025), 1–20. https://doi.org/10.1109/TKDE.2025.3609486

[19] Shizheng Hou, Wenqi Pei, Nuo Chen, Quang-Trung Ta, Peng Lu, and Beng Chin Ooi. 2025. A Modular Benchmarking Framework for LLM-Enabled NL2SQL Solutions [Experiment, Analysis & Benchmark] (Extended Version). https://github.com/HOU-SZ/NL2SQLBench.

[20] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169* (2023).

[21] Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025. MCS-SQL: Leveraging Multiple Prompts and Multiple-Choice Selection For Text-to-SQL Generation. In *Proceedings of the 31st International Conference on Computational Linguistics*, Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert (Eds.). Association for Computational Linguistics, Abu Dhabi, UAE, 337–353. https://aclanthology.org/2025.coling-main.24/

[22] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin SU, ZHAOQING SUO, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. 2025. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=XmProj9cPs

[23] Wenqiang Lei, Weixin Wang, Zhixin Ma, Tian Gan, Wei Lu, Min-Yen Kan, and Tat-Seng Chua. 2020. Re-examining the Role of Schema Linking in Text-to-SQL. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 6943–6954. https://doi.org/10.18653/v1/2020.emnlp-main.564

[24] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? *Proc. VLDB Endow.* 17, 11 (July 2024), 3318–3331. https://doi.org/10.14778/3681954.3682003

[25] Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. 2025. OmniSQL: Synthesizing High-Quality Text-to-SQL Data at Scale. arXiv:2503.02240 [cs.CL]

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

[27] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. CodeS: Towards Building Open-source Language Models for Text-to-SQL. *Proc. ACM Manag. Data* 2, 3, Article 127 (May 2024), 28 pages. https://doi.org/10.1145/3654930

[28] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C.C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM already Serve as a Database Interface? A big Bench for Large-Scale Database Grounded Text-To-SQLs. *Advances in Neural Information Processing Systems* 36 (2023), 42330–42357.

[29] Xiuwen Li, Qifeng Cai, Yang Shu, Chenjuan Guo, and Bin Yang. 2025. AID-SQL: Adaptive In-Context Learning of Text-to-SQL with Difficulty-Aware Instruction and Retrieval-Augmented Generation . In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 3945–3957. https://doi.org/10.1109/ICDE65448.2025.00294

[30] Yinheng Li. 2023. A Practical Survey on Zero-Shot Prompt Design for In-Context Learning. In *Proceedings of the 14th International Conference on Recent Advances in Natural Language Processing*, Ruslan Mitkov and Galia Angelova (Eds.). INCOMA Ltd., Shoumen, Bulgaria, Varna, Bulgaria, 641–647. https://aclanthology.org/2023.ranlp-1.69/

[31] Zhishuai Li, Xiang Wang, Jingjing Zhao, Sun Yang, Guoqing Du, Xiaoru Hu, Bin Zhang, Yuxiao Ye, Ziyue Li, Rui Zhao, and Hangyu Mao. 2024. PET-SQL: A Prompt-enhanced Two-stage Text-to-SQL Framework with Cross-consistency.

[32] Jinqing Lian, Xinyi Liu, Yingxia Shao, Yang Dong, Ming Wang, Zhang Wei, Tianqi Wan, Ming Dong, and Hailin Yan. 2024. ChatBI: Towards Natural Language to Complex Business Intelligence SQL. *arXiv preprint arXiv:2405.00527* (2024).

[33] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. https://doi.org/10.1162/tacl_a_00638

[34] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025. A Survey of Text-to-SQL in the Era of LLMs: Where Are We, and Where Are We Going? *IEEE Trans. Knowl. Data Eng.* 37, 10 (2025), 5735–5754. https://doi.org/10.1109/TKDE.2025.3592032

[35] Xinyu Liu, Shuyu Shen, Boyan Li, Nan Tang, and Yuyu Luo. 2025. Nl2sql-bugs: A benchmark for detecting semantic errors in NL2SQL translation. *arXiv preprint arXiv:2503.11984* (2025).

[36] Lin Long, Xijun Gu, Xinjie Sun, Wentao Ye, Haobo Wang, Sai Wu, Gang Chen, and Junbo Zhao. 2025. Bridging the Semantic Gap Between Text and Table: A Case Study on NL2SQL. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=qmsX2R19p9

[37] Shuai Lyu, Haoran Luo, Zhonghong Ou, Yifan Zhu, Xiaoran Shang, Yang Qin, and Meina Song. 2025. SQL-o1: A Self-Reward Heuristic Dynamic Search Method for Text-to-SQL. *arXiv preprint arXiv:2502.11741* (2025).

[38] 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*. https://openreview.net/forum?id=fglyh5pa7d

[39] Karime Maamari and Amine Mhedhbi. 2024. End-to-end text-to-sql generation within an analytics insight engine. *arXiv preprint arXiv:2406.12104* (2024).

[40] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems* 36 (2023), 46534–46594.

[41] 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*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 2009–2024. https://doi.org/10.18653/v1/2024.findings-acl.120

[42] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2025. Large language models: A survey. arXiv:2402.06196 [cs.CL] https://arxiv.org/abs/2402.06196

[43] Anna Mitsopoulou and Georgia Koutrika. 2025. Analysis of text-to-SQL benchmarks: limitations, challenges and opportunities. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025*. 199–212.

[44] Ali Mohammadjafari, Anthony S. Maida, and Raju Gottumukkala. 2024. From Natural Language to SQL: Review of LLM-based Text-to-SQL Systems. arXiv:2410.01066 [cs.CL]

[45] OpenAI. 2025. GPT-4o mini: Fast, affordable small model for focused tasks. https://platform.openai.com/docs/models/gpt-4o-mini

[46] Wenqi Pei, Hailing Xu, Henry Hengyuan Zhao, CHEN Han, Zining Zhang, Shizheng Hou, Luo Pingyi, and Bingsheng He. 2025. Optimizing Small Language Models for NL2SQL. In *ICLR 2025 Third Workshop on Deep Learning for Code*. https://openreview.net/forum?id=xGkxWP2wE4

[47] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2025. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=CvGqMD5OtX

[48] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=p53QDxSIc5

[49] Mohammadreza Pourreza and Davood Rafiei. 2023. Evaluating Cross-Domain Text-to-SQL Models and Benchmarks. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 1601–1611. https://doi.org/10.18653/v1/2023.emnlp-main.99

[50] Mohammadreza Pourreza and Davood Rafiei. 2024. DTS-SQL: Decomposed Text-to-SQL with Small Large Language Models. *arXiv:2402.01117* (2024).

[51] Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wan, Hailong Li, Azalia Mirhoseini, Amin Saberi, and Sercan "O. Arik. 2025. Reasoning-SQL: Reinforcement Learning with SQL Tailored Partial Rewards for Reasoning-Enhanced Text-to-SQL. *arXiv preprint arXiv:2503.23157* (2025).

[52] Yang Qin, Chao Chen, Zhihang Fu, Ze Chen, Dezhong Peng, Peng Hu, and Jieping Ye. 2025. ROUTE: Robust Multitask Tuning and Collaboration for Text-to-SQL. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=BAglD6NGy0

[53] 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*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 5456–5471. https://doi.org/10.18653/v1/2024.findings-acl.324

[54] 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 *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 15–28. https://doi.org/10.1109/ICDE60146.2024.00009

[55] Jaydeep Sen, Fatma Ozcan, Abdul Quamar, Greg Stager, Ashish Mittal, Manasa Jammi, Chuan Lei, Diptikalyan Saha, and Karthik Sankaranarayanan. 2019. Natural Language Querying of Complex Business Intelligence Queries. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1997–2000. https://doi.org/10.1145/3299869.3320248

[56] Lei Sheng and Shuai-Shuai Xu. 2025. CSC-SQL: Corrective Self-Consistency in Text-to-SQL via Reinforcement Learning. *arXiv preprint arXiv:2505.13271* (2025).

[57] 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:2407.15186 [cs.CL]

[58] Liang Shi, Zhengju Tang, Nan Zhang, Xiaotong Zhang, and Zhi Yang. 2025. A Survey on Employing Large Language Models for Text-to-SQL Tasks. *ACM Comput. Surv.* 58, 2, Article 54 (Sept. 2025), 37 pages.

[59] Yewei Song, Saad Ezzini, Xunzhu Tang, Cedric Lothritz, Jacques Klein, Tegawendé Bissyandé, Andrey Boytsov, Ulrick Ble, and Anne Goujon. 2024. Enhancing Text-to-SQL translation for financial system design. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. 252–262.

[60] 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* (2024).

[61] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. 2025. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. In *Proceedings of the 31st International Conference on Computational Linguistics*. 540–557.

[62] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (Eds.). Association for Computational Linguistics, Online, 7567–7578. https://doi.org/10.18653/v1/2020.acl-main.677

[63] 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 *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=1PL1NIMMrw

[64] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent Abilities of Large Language Models. *Transactions on Machine Learning Research* (2022). https://openreview.net/forum?id=yzkSU5zdwD Survey Certification.

[65] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=_VjQlMeSB_J

[66] Luoxuan Weng, Yinghao Tang, Yingchaojie Feng, Zhuo Chang, Ruiqin Chen, Haozhe Feng, Chen Hou, Danqing Huang, Yang Li, Huaming Rao, Haonan Wang, Canshi Wei, Xiaofeng Yang, Yuhui Zhang, Yifeng Zheng, Xiuqi Huang, Minfeng Zhu, Yuxin Ma, Bin Cui, Peng Chen, and Wei Chen. 2024. DataLab: A Unified Platform for LLM-Powered Business Intelligence. *arXiv preprint arXiv:2412.02205* (2024).

[67] Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. 2024. C-Pack: Packed Resources For General Chinese Embeddings. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Washington DC, USA) *(SIGIR '24)*. Association for Computing Machinery, New York, NY, USA, 641–649. https://doi.org/10.1145/3626772.3657878

[68] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. OpenSearch-SQL: Enhancing Text-to-SQL with Dynamic Few-shot and Consistency Alignment. arXiv:2502.14913 [cs.CL] https://arxiv.org/abs/2502.14913

[69] Yuanzhen Xie, Xinzhou Jin, Tao Xie, Matrixmxlin Matrixmxlin, Liang Chen, Chenyun Yu, Cheng Lei, Chengxiang Zhuo, Bo Hu, and Zang Li. 2024. Decomposition for Enhancing Attention: Improving LLM-based Text-to-SQL through Workflow Paradigm. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 10796–10816. https://doi.org/10.18653/v1/2024.findings-acl.641

[70] Siqiao Xue, Danrui Qi, Caigao Jiang, Fangyin Cheng, Keting Chen, Zhiping Zhang, Hongyang Zhang, Ganglin Wei, Wang Zhao, Fan Zhou, Hong Yi, Shaodong Liu, Hongjun Yang, and Faqiang Chen. 2024. Demonstration of DB-GPT: Next Generation Data Interaction System Empowered by Large Language Models. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4365–4368. https://doi.org/10.14778/3685800.3685876

[71] Yicun Yang, Zhaoguo Wang, Yu Xia, Zhuoran Wei, Haoran Ding, Ruzica Piskac, Haibo Chen, and Jinyang Li. 2025. Automated Validating and Fixing of Text-to-SQL Translation with Execution Consistency. In *SIGMOD*.

[72] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 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*, Ellen Riloff,

David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, Brussels, Belgium, 3911–3921. https://doi.org/10.18653/v1/D18-1425

[73] Hongwei Yuan, Xiu Tang, Ke Chen, Lidan Shou, Gang Chen, and Huan Li. 2025. CogSQL: A Cognitive Framework for Enhancing Large Language Models in Text-to-SQL Translation. In *AAAI*. 25778–25786. https://doi.org/10.1609/aaai.v39i24.34770

[74] Bin Zhang, Yuxiao Ye, Guoqing Du, Xiaoru Hu, Zhishuai Li, Sun Yang, Chi Harold Liu, Rui Zhao, Ziyue Li, and Hangyu Mao. 2024. Benchmarking the Text-to-SQL Capability of Large Language Models: A Comprehensive Evaluation. arXiv:2403.02951 [cs.CL] https://arxiv.org/abs/2403.02951

[75] Meihui Zhang, Zhaoxuan Ji, Zhaojing Luo, Yuncheng Wu, and Chengliang Chai. 2024. Applications and Challenges for Large Language Models: From Data Management Perspective. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 5530–5541. https://doi.org/10.1109/ICDE60146.2024.00441

[76] Yi Zhang, Jan Deriu, George Katsogiannis-Meimarakis, Catherine Kosten, Georgia Koutrika, and Kurt Stockinger. 2023. ScienceBenchmark: A Complex Real-World Benchmark for Evaluating Natural Language to SQL Systems. *Proc. VLDB Endow.* 17, 4 (2023), 685–698.

[77] Fuheng Zhao, Shaleen Deep, Fotis Psallidas, Avrilia Floratou, Divyakant Agrawal, and Amr El Abbadi. 2025. Sphinteract: Resolving Ambiguities in NL2SQL through User Interaction. *Proc. VLDB Endow.* 18, 4 (2025), 1145–1158. https://doi.org/10.14778/3717755.3717772

[78] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2025. A survey of large language models. arXiv:2303.18223 [cs.CL] https://arxiv.org/abs/2303.18223

[79] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* (2017). arXiv:1709.00103

[80] Xiaohu Zhu, Qian Li, Lizhen Cui, and Yongkang Liu. 2024. Large Language Model Enhanced Text-to-SQL Generation: A Survey. arXiv:2410.06011 [cs.DB]

A APPENDIX

A.1 JSON Structure Used to Collect Results from NL2SQL Solutions

We present an example of the JSON format used by our Result Collecting Agent to collect the results of each NL2SQL solution as follows:

```
[{
 "node_type": "schema_selection",
 "question": "How many cards are there with toughness of 99?",
 "extracted_schema": { "cards": ["id", "toughness"] }
 "token_cost": 1200,
 "llm_calls": 1
 },
 {
 "node_type": "candidate_generation",
 "question": "How many cards are there with toughness of 99?",
 "SQL": "SELECT COUNT(id) FROM cards WHERE toughness = 99",
 "token_cost": 1200,
 "llm_calls": 1
 },
 {
 "node_type": "query_revision",
 "question": "How many cards are there with toughness of 99?",
 "SQL": "SELECT COUNT(id) FROM cards WHERE toughness = 99",
 "token_cost": 1200,
 "llm_calls": 1
 }]
```

The meanings of JSON fields in the collected files are:

- `node_type`: the module that produced this record.
- `question`: the original natural language query Q.
- `extracted_schema`: a mapping of selected tables and columns.
- `SQL`: the SQL generated by the model.
- `token_cost`: the token cost for this module.
- `llm_calls`: the number of of LLMs invocations.

A.2 Additional Results for Schema Selection Module

In this section, we present the results for the Schema Selection module obtained using GPT-4o-mini on the BIRD dataset, using DeepSeek-V3 on the ScienceBenchmark dataset, amd using GPT-4o-mini on the ScienceBenchmark dataset, as shown in Table 12, Table 13, and Table 14 respectively.

Table 12: Analysis results of Schema Selection on BIRD dev set using GPT-4o-mini. The best results are in bold and underlined while the second-best results are underlined

Approach	Table Selection			Column Selection			Efficiency	
	Precision (%)	Recall (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)	#Tokens	#LLM calls
C3-SQL	49.79	98.46	64.35	26.80	93.75	40.30	10979	20
DIN-SQL	92.08	95.89	92.83	87.49	85.51	**85.47**	7197	**1**
MAC-SQL	32.74	**99.84**	46.70	14.83	96.63	24.49	**3107**	**1**
CHESS$_{(IR,SS,CG)}$	90.96	95.80	92.17	**88.74**	74.14	78.96	297443	78.53
TA-SQL	**92.56**	95.38	**92.88**	81.78	88.44	83.25	4119	**1**
RSL-SQL	81.64	97.18	86.57	46.58	80.95	52.39	5528	**1**
OpenSearch-SQL	32.76	99.49	46.66	16.88	**97.14**	27.34	6616	3

Table 13: Analysis results of Schema Selection on ScienceBenchmark dev set using DeepSeek-V3. The best results are in bold and underlined while the second-best results are underlined

Approach	Table Selection			Column Selection			Efficiency	
	Precision (%)	Recall (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)	#Tokens	#LLM calls
C3-SQL	49.50	97.06	62.98	30.69	94.30	43.64	18869	20
DIN-SQL	89.58	81.15	83.48	**79.80**	73.20	73.92	6802	**1**
MAC-SQL	18.56	**100**	28.93	11.15	**96.39**	18.42	**3380**	**1**
CHESS$_{(IR,SS,CG)}$	91.23	82.44	84.86	76.48	73.51	72.42	352686	92.59
TA-SQL	**92.72**	90.20	**90.13**	74.16	85.27	**76.39**	5155	**1**
RSL-SQL	86.53	91.76	87.01	40.59	88.17	50.39	7669	**1**
OpenSearch-SQL	12.35	67.89	19.28	7.91	66.79	13.44	6237	3

Table 14: Analysis results of Schema Selection on ScienceBenchmark dev set using GPT-4o-mini. The best results are in bold and underlined while the second-best results are underlined

Approach	Table Selection			Column Selection			Efficiency	
	Precision (%)	Recall (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)	#Tokens	#LLM calls
C3-SQL	49.25	95.39	62.19	29.17	91.50	41.51	12829	20
DIN-SQL	90.18	81.71	83.65	77.23	68.77	70.65	6571	**1**
MAC-SQL	18.56	**100**	28.93	10.48	96.22	17.26	**3167**	**1**
CHESS$_{(IR,SS,CG)}$	**90.96**	95.80	**92.17**	**88.74**	74.14	**78.96**	341059	92.59
TA-SQL	90.17	86.36	86.38	67.98	79.63	70.19	5049	**1**
RSL-SQL	74.40	85.65	76.88	39.10	70.97	42.73	7453	**1**
OpenSearch-SQL	18.56	**100**	28.93	12.30	**97.90**	20.77	6264	3

A.3 Additional Results for Candidate Generation Module

A.3.1 Error Analysis on Candidate Generation. We break execution errors (ER) into five categories: No Table/Column, No Function, Syntax Error, Timeout, and Others. Although our earlier analysis shows that incorrect-but-executable (IR) dominates overall failures, understanding the composition of ER is important for robustness and cost. The detailed results are shown in Table 15, Table 16, Table 17 and Table 18.

BIRD. With DeepSeek-V3, OpenSearch-SQL exhibits the lowest total ER, with small contributions from all buckets; CHESS$_{(IR,SS,CG)}$ is also strong but sees a slightly higher Timeout share. With GPT-4o-mini, the lowest ER shifts to CHESS$_{(IR,SS,CG)}$, while several methods show noticeably higher No Table/Column errors than with DeepSeek-V3. Overall, on BIRD, execution failures are already rare for the top systems; the main movable pieces are schema-linking lapses (No Table/Column) and light syntax/dialect issues.

ScienceBenchmark. With DeepSeek-V3, patterns diverge: TA-SQL keeps ER very low, while OpenSearch-SQL shows a dramatic spike in Others. With GPT-4o-mini, CHESS$_{(IR,SS,CG)}$ achieves the lowest ER, while several methods suffer increased No Table/Column rates. Overall, on denser, more realistic schemas, resource and plan-related failures (Timeout/Others) become salient for breadth-heavy generation, whereas IR/SS-style pipelines remain execution-stable.

Table 15: Error analysis on BIRD dev set with DeepSeek-V3 for Candidate Generation. The best results are in bold and underlined while the second-best results are underlined. We use `pass@1` results for approaches producing multiple candidates.

Approach	Error Rates (%)					Total (%)
	No Tabl/Col	No Func	Syntax Err	Timeout	Others	
C3-SQL	3.32	**0**	**0.65**	0.52	7.30	11.80
DIN-SQL	0.98	0.13	3.12	0.26	0.39	4.89
MAC-SQL	1.11	0.07	1.83	0.26	4.17	7.43
CHESS$_{(IR,SS,CG)}$	0.46	1.43	1.69	3.32	**0.20**	7.11
CHESS$_{(IR,CG,UT)}$	1.30	**0**	2.28	**0.20**	**0.20**	3.98
TA-SQL	1.30	1.69	2.54	0.13	**0.20**	5.87
GSR	2.41	0.58	1.63	0.52	0	5.48
E-SQL	0.65	0.13	2.87	0.26	3.52	7.43
RSL-SQL	**0.39**	0.91	1.37	0.33	4.62	7.62
OpenSearch-SQL	0.78	0.07	0.84	**0.07**	1.30	**3.06**

Table 16: Error analysis on BIRD Dev using GPT-4o-mini for Candidate Generation. The best results are in bold and underlined while the second-best results are underlined. We use `pass@1` results for approaches producing multiple candidates.

Approach	Error Rates (%)					Total (%)
	No Tabl/Col	No Func	Syntax Err	Timeout	Others	
C3-SQL	9.19	**0**	1.96	0.07	1.3	12.52
DIN-SQL	5.48	0.46	2.41	0.2	0.33	8.87
MAC-SQL	6.26	1.43	0.85	0.33	2.48	11.34
CHESS$_{(IR,SS,CG)}$	**2.54**	0.72	1.11	0.91	0.26	**5.54**
CHESS$_{(IR,CG,UT)}$	9.32	1.76	1.63	**0**	0.2	12.91
TA-SQL	7.37	4.82	0.78	0.26	**0.13**	13.36
GSR	**2.54**	0.39	1.43	0.07	2.61	7.04
E-SQL	5.48	0.72	1.17	0.07	2.48	9.91
RSL-SQL	4.24	1.5	0.72	0.13	2.87	9.45
OpenSearch-SQL	6.13	0.85	**0.46**	0.2	0.52	8.15

Table 17: Error analysis on ScienceBenchmark Dev using DeepSeek-V3 for Candidate Generation. The best results are in bold and underlined while the second-best results are underlined. We use `pass@1` results for approaches producing multiple candidates.

Approach	Error Rates (%)					Total (%)
	No Tabl/Col	No Func	Syntax Err	Timeout	Others	
C3-SQL	3.01	**0**	**0**	**0**	**0**	3.01
DIN-SQL	2.01	**0**	**0**	**0**	0.33	2.34
MAC-SQL	4.35	**0**	0.33	**0**	5.35	10.03
CHESS$_{(IR,SS,CG)}$	**0.67**	**0**	**0**	6.02	**0**	6.69
CHESS$_{(IR,CG,UT)}$	3.01	**0**	**0**	**0**	2.68	5.69
TA-SQL	2.01	**0**	**0**	**0**	**0**	**2.01**
GSR	11.04	**0**	0.67	**0**	0.33	12.04
E-SQL	1.34	**0**	0.33	**0**	3.34	5.02
RSL-SQL	**0.67**	**0**	**0**	**0**	14.38	15.05
OpenSearch-SQL	2.01	**0**	**0**	**0**	32.44	34.45

Table 18: Error analysis on ScienceBenchmark Dev using GPT-4o-mini for Candidate Generation. The best results are in bold and underlined while the second-best results are underlined. We use `pass@1` results for approaches producing multiple candidates.

Approach	Error Rates (%)					Total (%)
	No Tabl/Col	No Func	Syntax Err	Timeout	Others	
C3-SQL	4.68	**0**	**0**	**0**	1.67	6.35
DIN-SQL	8.36	**0**	**0**	0.33	**0**	8.70
MAC-SQL	8.70	**0**	**0**	0.33	5.02	14.05
CHESS$_{(IR,SS,CG)}$	**2.01**	**0**	**0**	0.33	**0**	**2.34**
CHESS$_{(IR,CG,UT)}$	12.37	**0**	**0**	**0**	**0**	12.37
TA-SQL	6.02	**0**	**0**	**0**	**0**	6.02
GSR	6.35	**0**	**0**	0.33	0.67	7.36
E-SQL	2.34	**0**	**0**	0.33	12.71	15.38
RSL-SQL	6.35	**0**	**0**	0.33	**0**	6.69
OpenSearch-SQL	10.37	**0**	**0**	0.33	0.33	11.04

Backbone LLM effects. Switching from DeepSeek-V3 to GPT-4o-mini increases No Table/Column errors for multiple methods on both datasets, suggesting backbone-specific schema-linking calibration matters. By contrast, Syntax Error remains low across all settings, indicating that most generators respect SQL grammar.

A.3.2 Result analysis on difficulty level. Figure 8 9 10 11 present the *Correct Rates*, *Incorrect Rates*, and *Error Rates* for the Candidate Generation module across three difficulty levels—Simple, Moderate, and Challenging—for each evaluated approach under the four experiment settings.

Overall trend. As we can see, when the difficulty level increases gradually, the Correct Rates show of declining trend, while IR increases and becomes the dominant mass of errors; ER remains comparatively small but shows a mild uptick on Challenging. This confirms that most failures at higher difficulty are semantically wrong yet executable SQL (e.g., wrong join path, boundary/filter mismatch, misused grouping), rather than parser/dialect errors.

Dataset effect. Within each difficulty band, ScienceBenchmark yields lower CR and higher IR than BIRD for many methods. The gap widens on Challenging, reflecting ScienceBenchmark's denser schemas. Some systems (e.g., OpenSearch-SQL) have done targeted alignment for BIRD dataset, thus show good result on BIRD, while on ScienceBenchmark their advantage narrows. Conversely, concise pipelines remain comparatively stable across backbones.

Backbone LLM effect. Changing backbone LLMs reorders methods within each difficulty band, but it does not change the global trend: CR decreases and IR increases with difficulty in both LLMs.

A.4 Additional Results for Query Revision Module

A.4.1 Result analysis on difficulty level. Figure 12, Figure 13, Figure 14 and Figure 15 present the *Correct Rates*, *Incorrect Rates*, and *Error Rates* across three difficulty levels for each approach after applying the Query Revision module.

Overall trend. Similar to the trend previously, the correct rate gradually decreases as query difficulty progresses from Simple to Challenging. Notably, Incorrect queries continue to constitute the majority of unsuccessful results, maintaining relatively high Incorrect Rates post-revision. These queries are syntactically valid yet fail to reproduce results identical to the ground truth SQL, revealing a persistent semantic mismatch. Their prevalence, therefore,

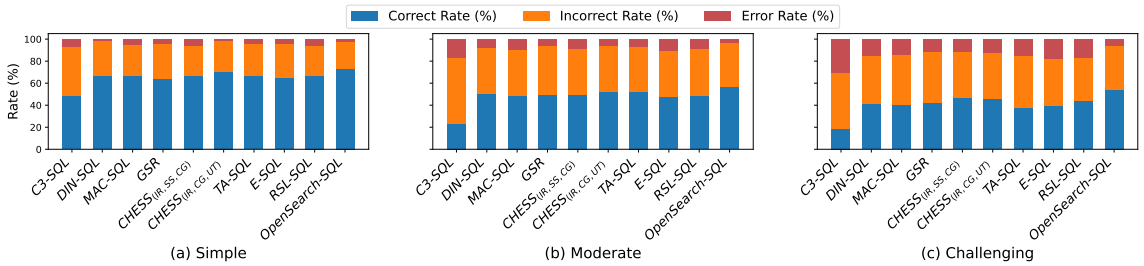


Figure 8: Correct Rates, Incorrect Rates and Error Rates on BIRD dev set using DeepSeek-V3 for the Candidate Generation module across difficulty levels.

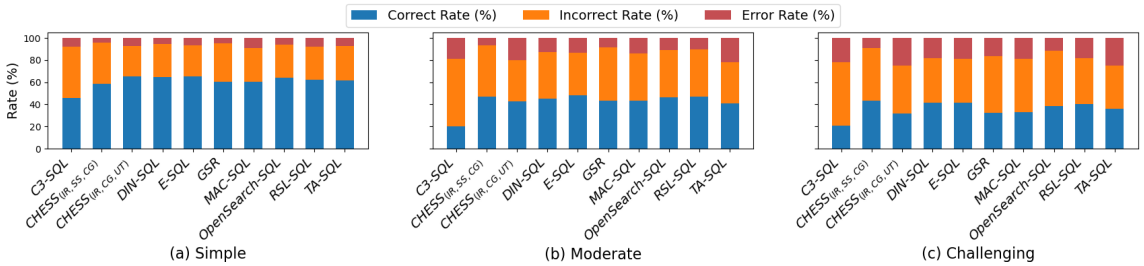


Figure 9: Correct Rates, Incorrect Rates and Error Rates on BIRD dev set using GPT-4o-mini for the Candidate Generation module across difficulty levels.

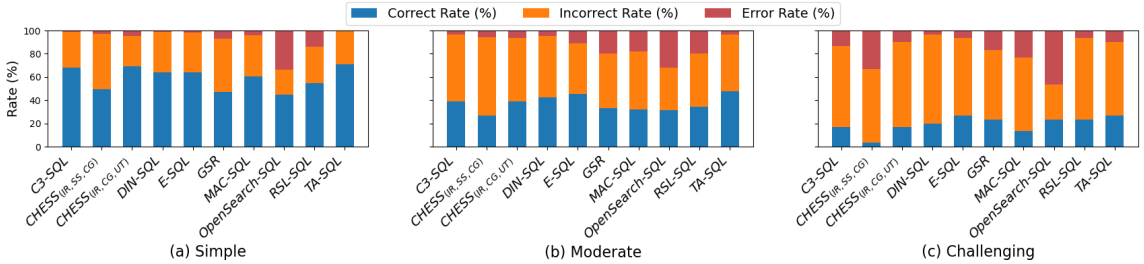


Figure 10: Correct Rates, Incorrect Rates and Error Rates on ScienceBenchmark dev set using DeepSeek-V3 for the Candidate Generation module across difficulty levels.

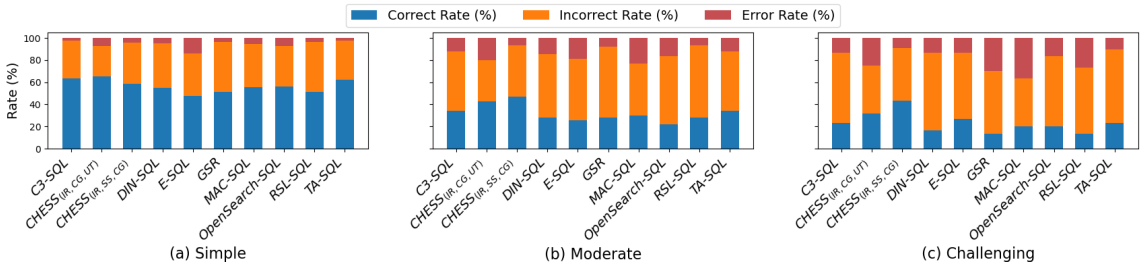


Figure 11: Correct Rates, Incorrect Rates and Error Rates on ScienceBenchmark dev set using GPT-4o-mini for the Candidate Generation module across difficulty levels.

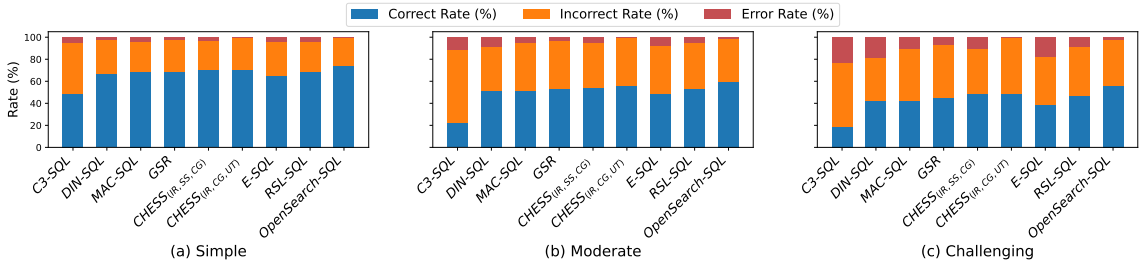


Figure 12: Correct Rates, Incorrect Rates and Error Rates on BIRD dev set using DeepSeek-V3 for Query Revision module across difficulty levels.

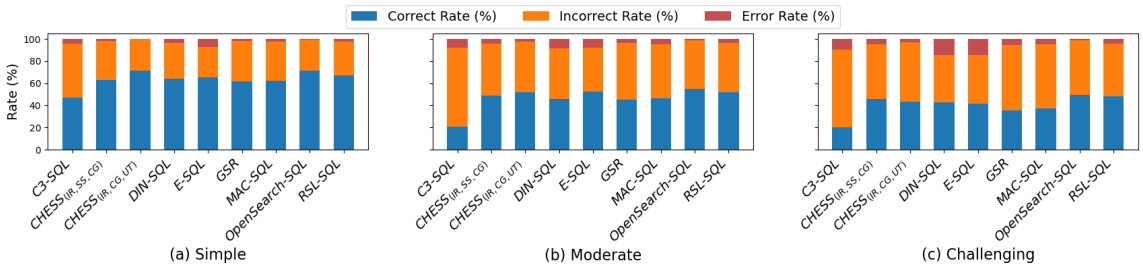


Figure 13: Correct Rates, Incorrect Rates and Error Rates on BIRD dev set using GPT-4o-mini for Query Revision module across difficulty levels.

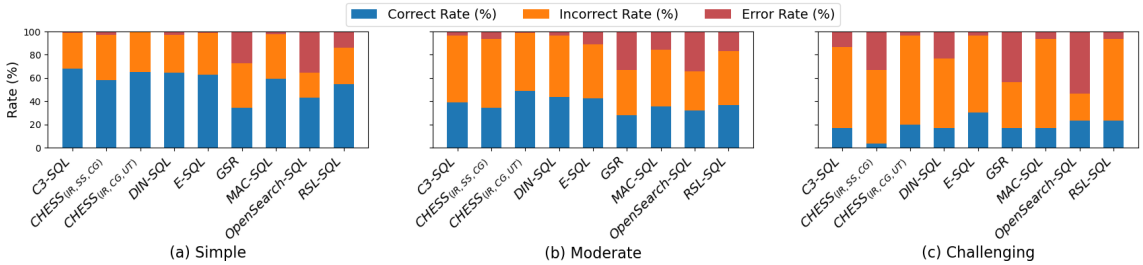


Figure 14: Correct Rates, Incorrect Rates and Error Rates on ScienceBenchmark dev set using DeepSeek-V3 for Query Revision module across difficulty levels.

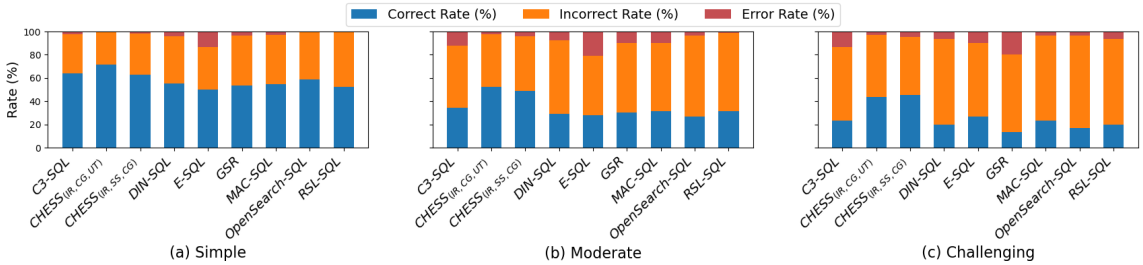


Figure 15: Correct Rates, Incorrect Rates and Error Rates on ScienceBenchmark dev set using GPT-4o-mini for Query Revision module across difficulty levels.

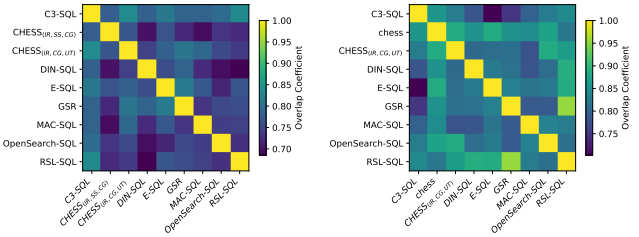
indicates that current revision strategies are still insufficient for faithfully capturing and accurately aligning with the semantic intentions expressed in users' NL queries.

Dataset effect. On BIRD, several systems preserve a reasonable CR margin from Simple to Moderate and achieve visible IR reduction after revision. On ScienceBenchmark, however, the CR drop is not as obvious as BIRD. Notably, OpenSearch-SQL, which has undergone extensive alignment optimization for BIRD, does not transfer well to ScienceBenchmark: its CR advantage narrows, indicating that dataset-specific alignment does not generalize and must be re-designed for new schema distributions.

Backbone LLM effect. Changing the backbone reorders methods within each difficulty band but does not alter the global shape.

Insight: *Current query revision approaches remain inadequate at identifying and rectifying Incorrect queries, which represent the predominant error type. Since these queries are syntactically correct, they are more subtle and challenging to detect, reflecting a persistent misalignment with users' intended semantics.*

Practical guide. Boosting CR requires semantic alignment rather than additional surface rewrites. Alignment strategies must generalize across datasets to ensure robustness and generalization.



(a) **Overlap on the BIRD (using GPT-4o-mini)** (b) **Overlap on the ScienceBenchmark (using GPT-4o-mini)**

Figure 16: The coefficient heatmap of different solutions on Incorrect cases (using GPT-4o-mini).

A.4.2 An in-depth analysis of incorrect cases. Similar to the results shown in Section 4.4.2, we observed an unexpectedly high degree of overlap in incorrect queries generated by different methods using GPT-4o-mini on the two datasets, as illustrated in Figure 16.

A.4.3 Detailed analysis on Query Revision performance. We present the full results of Query Revision module using Correct-Improvement (CI), Incorrect-to-Correct (I2C), and Error-to-Correct (E2C), Correct-to-Incorrect(C2I), Correct-to-Error(C2E) along with efficiency (#tokens/#LLM calls), as shown in Table 19, Table 20, Table 21 and Table 22.

Table 19: Effectiveness and efficiency on Query Revision on BIRD dev with DeepSeek-V3. The best results are in bold and underlined while the second-best results are underlined.

Approach	CI (%)	I2C (%)	E2C (%)	C2I (%)	C2E (%)	#Tokens	#LLM Calls
C3-SQL	-0.51	1.43	4.97	3.93	**0.00**	–	–
DIN-SQL	0.66	2.02	21.33	0.98	1.31	4948	**1**
MAC-SQL	3.55	3.66	13.16	0.22	**0.00**	**1566**	2
CHESS$_{(IR,SS,CG)}$	6.11	6.89	29.36	0.65	0.55	1844	**1**
CHESS$_{(IR,CG,UT)}$	2.18	**15.16**	**47.54**	8.81	**0.00**	106126	21
GSR	**7.27**	9.49	38.10	2.16	0.34	2220	2
E-SQL	0.00	3.53	14.04	3.06	0.91	24924	2
RSL-SQL	4.97	4.69	17.95	**0.00**	**0.00**	3315	2
OpenSearch-SQL	3.16	8.67	34.04	2.47	**0.00**	6652	4.6

Table 20: Effectiveness and efficiency on Query Revision on BIRD dev with GPT-4o-mini. The best results are in bold and underlined while the second-best results are underlined.

Approach	CI (%)	I2C (%)	E2C (%)	C2I (%)	C2E (%)	#Tokens	#LLM Calls
C3-SQL	2.56	2.52	8.33	4.02	**0**	–	–
DIN-SQL	-0.11	4.16	19.12	4.37	1.27	4791.23	**1**
MAC-SQL	4.45	1.45	21.84	**1.24**	**0**	1729.12	2
CHESS$_{(IR,SS,CG)}$	6.72	10.49	35.29	4.64	0.37	**1683.28**	**1**
CHESS$_{(IR,CG,UT)}$	14.04	16.09	38.38	4.26	0.12	115055.19	23.19
GSR	2.6	4.85	21.3	3.71	0.25	2448.84	2
E-SQL	1.91	7.52	28.29	4.04	3.03	25022.94	2
RSL-SQL	9.15	10.06	35.86	2.82	0.47	2802.02	2
OpenSearch-SQL	**14.07**	**17.71**	**51.20**	4.38	**0**	12968.94	6.64

Table 21: Effectiveness and efficiency on Query Revision on ScienceBenchmark dev with DeepSeek-V3. The best results are in bold and underlined while the second-best results are underlined.

Approach	CI (%)	I2C (%)	E2C (%)	C2I (%)	C2E (%)	#Tokens	#LLM Calls
C3-SQL	0	0	0	**0**	**0**	–	–
DIN-SQL	1.27	2.99	14.29	0.63	1.27	4349.08	**1**
MAC-SQL	1.42	2.34	10	2.84	**0**	2395.41	2
CHESS$_{(IR,SS,CG)}$	**19.47**	**14.55**	0	1.77	**0**	2319.17	**1**
CHESS$_{(IR,CG,UT)}$	1.83	9.32	**52.94**	10.37	**0**	201372.44	23.42
GSR	-24.8	14.08	33.33	23.14	28.1	**1756.85**	2
E-SQL	-2.45	4.96	13.33	6.13	1.23	15088.02	2
RSL-SQL	1.47	0.00	4.44	**0**	**0**	6616.00	2
OpenSearch-SQL	-1.74	7.41	3.88	4.35	6.09	18574.91	7.17

Table 22: Effectiveness and efficiency on Query Revision on ScienceBenchmark dev with GPT-4o-mini. The best results are in bold and underlined while the second-best results are underlined.

Approach	CI (%)	I2C (%)	E2C (%)	C2I (%)	C2E (%)	#Tokens	#LLM Calls
C3-SQL	0	0	0	**0**	**0**	–	–
DIN-SQL	2.34	1.38	15.38	1.56	0.78	4156.48	**1**
MAC-SQL	0.76	3.20	9.52	5.30	**0**	2675.69	2
CHESS$_{(IR,SS,CG)}$	**26.32**	**13.30**	14.29	2.11	**0**	1683.28	**1**
CHESS$_{(IR,CG,UT)}$	20.49	12.14	**43.24**	6.56	**0**	115055.19	23.19
GSR	4.96	5.13	18.18	4.13	0.83	**1498.20**	2
E-SQL	5.17	6.57	4.35	2.59	1.72	21360.30	2
RSL-SQL	5.79	0.00	35	**0**	**0**	3720.00	2
OpenSearch-SQL	6.35	7.14	33.33	10.32	**0**	34752.35	11.33

A.5 Detailed Explanation of Practical Guide

Schema Selection Module. We recommend implementing a two-stage schema pruning pipeline that balances effectiveness and efficiency for schema selection. In the first stage, employ lightweight retrieval methods (e.g., embedding-based similarity search or keyword matching) to broadly filter irrelevant tables and columns, prioritizing high recall to minimize the risk of excluding relevant schema elements. In the second stage, leverage LLMs with carefully designed prompts to perform precision-oriented refinement on the reduced schema space, focusing on eliminating irrelevant elements that would otherwise increase token costs and degrade downstream performance.

Additionally, for production systems handling ultra-large databases, we recommend maintaining a schema cache indexed by query embeddings to avoid redundant schema selection for similar queries, and implementing adaptive schema expansion that dynamically adjusts the schema scope based on initial query execution feedback.

Candidate Generation Module. We recommend implementing execution-result-based semantic validation during the generation phase rather than deferring all validation to the revision stage. Specifically, for each generated candidate query, execute it against the database and analyze the result schema (column names, data types) and sample result values to detect potential semantic mismatches—such as selecting ID columns instead of name columns, or aggregating over inappropriate attributes. This early detection mechanism can prevent the propagation of semantic errors to downstream modules and reduce the burden on query revision.

For systems generating multiple candidates, our Pass@k analysis reveals that increasing diversity (k > 10) yields diminishing returns; instead, practitioners should focus on improving the quality of top-ranked candidates through better prompt engineering and schema-aware generation strategies. Intermediate Representations (e.g., NatSQL, Pandas-like code) show promise in improving correctness, but introduce syntax translation overhead—practitioners should carefully weigh this trade-off based on their target SQL dialect complexity.

For production deployment, we recommend implementing progressive generation that starts with simpler query structures and incrementally adds complexity only when necessary, as our difficulty-level analysis shows that accuracy degrades sharply for challenging

queries, suggesting that avoiding unnecessary complexity can improve overall robustness.

Query Revision Module. We present the dual-strategy revision approach as follows. First, implement execution-result-based semantic diagnosis that analyzes not just error messages but also the structure and content of returned results—comparing result schemas, row counts, and sample values against expected patterns derived from the natural language query. Second, adopt a conservative revision policy that explicitly validates whether originally correct queries remain correct after revision. Concretely, before committing a revision, execute both the original and revised queries and apply a consistency check: if the original query succeeded and the revised query produces different results, require additional validation steps (e.g., LLM-based semantic equivalence checking or unit test verification) before accepting the revision.

For systems employing multiple-candidate generation, our findings suggest that the query selection problem is as critical as the revision problem itself—the gap between Pass@5 upper bounds and post-revision performance indicates that current selection mechanisms fail to identify the best candidate reliably. We recommend implementing ensemble-based selection that combines multiple signals: execution success, result plausibility (e.g., non-empty results for queries expecting data), query complexity metrics, and LLM-based confidence scores.

Finally, given the substantial token costs of iterative revision (up to 100K+ tokens for multi-turn approaches), practitioners should implement early stopping mechanisms based on execution feedback: if a query executes successfully and returns non-empty results with expected schema, terminate revision to avoid unnecessary costs and potential degradation.

End-to-end Evaluation. We present the details of the adaptive module selection we recommend as follows. Specifically, design a query complexity classifier (which can be a lightweight model or rule-based heuristic) that categorizes incoming queries into simple, moderate, and challenging classes based on features such as question length, number of entities mentioned, presence of aggregations or joins, and schema size. For simple queries, employ streamlined pipelines that skip expensive multi-stage schema pruning and multi-candidate generation, directly applying single-pass methods For moderate queries, selectively enable key optimization strategies—such as two-stage schema pruning and single-turn execution-guided revision—that hopefully provide acceptable accuracy at reasonable cost. Reserve expensive strategies like multi-candidate generation with consistency-based selection and iterative multi-turn revision exclusively for challenging queries. Additionally, implement confidence-based early termination: if a candidate query executes successfully, returns non-empty results with expected schema structure, and receives high LLM confidence scores, terminate the pipeline early to avoid unnecessary revision costs. This adaptive approach can potentially reduce average token consumption while maintaining the accuracy of always-expensive strategies, thereby enabling practical deployment at scale.

A.6 Example of New Discoveries on Inaccurate Gold SQL Queries and Evaluations

During our evaluation of the BIRD dataset, we identified several gold SQL queries that are inaccurate and have not been discovered previously by existing works, such as [35]. One example of our findings is the question: *"Name the foreign name of the card that has abzan watermark? List out the type of this card"* (BIRD question_id = 448) is currently aligned with an *incorrect* gold SQL query:

```
SELECT DISTINCT T1.name, T1.type FROM cards AS T1
INNER JOIN foreign_data AS T2 ON T2.uuid = T1.uuid
WHERE T1.watermark = 'abzan';
```

In the above query, the column `T1.name` is incorrectly selected. The correct SQL query should instead select `T2.name` like below:

```
SELECT DISTINCT T2.name, T1.type FROM cards AS T1
INNER JOIN foreign_data AS T2 ON T2.uuid = T1.uuid
WHERE T1.watermark = 'abzan';
```

Another example of a provided inaccurate query is shown below.

Question: Name all cards with 2015 frame style ranking below 100 on EDHRec?

- ▷ *Incorrect Gold SQL query:* `SELECT id FROM cards WHERE edhrecRank < 100 AND frameVersion = 2015;`
- ▷ *Correct SQL query:* `SELECT name FROM cards WHERE edhrecRank < 100 AND frameVersion = 2015;`

Over-Strict Evaluation Rule. Consider the question: *"Which user added a bounty amount of 50 to the post title mentioning variance?"* (BIRD question_id = 586). BIRD's Gold SQL query returns both the `DisplayName` and `Title` columns. An alternative query that retrieves only the `DisplayName` column still satisfies most information needs. However, BIRD's existing evaluation rules label it as incorrect.

The correct SQL query should return the `name` column, but the given gold SQL only returns the `id` column. Given that the gold SQL serves as the critical reference for evaluating the correctness and effectiveness of NL2SQL methods, inaccuracies in these gold queries can significantly undermine the validity and reliability of the benchmark results.

Semantic Ambiguity. Consider the question: *"Which school in Contra Costa has the highest number of test takers?"* (BIRD question_id = 22).

In the underlying schema, a *"school"* can be represented by any of the columns `CDSCode`, `sname`, `School Name`, or `School`. Each of these columns can be reasonably regarded as a valid answer, highlighting the inherent ambiguity in the question formulation. Because each of these columns is a plausible surrogate for the same concept, multiple SQL queries—returning different but semantically equivalent result sets—should be deemed correct. The current one-to-one evaluation paradigm of BIRD cannot capture this variability and thus may unfairly penalize systems that choose an alternative legitimate representation.

Practical Guide. we provide the following recommendations for practitioners and benchmark developers.

First, to mitigate the impact of inaccurate gold annotations, we recommend implementing multi-reference evaluation where generated queries are validated against multiple acceptable SQL variants rather than a single gold query—this can be achieved through result-equivalence checking (comparing execution results across multiple

semantically equivalent queries) or through LLM-based semantic equivalence verification. For production systems, establish a human-in-the-loop validation process for queries that execute successfully but are marked incorrect by automated evaluation, as these cases often indicate annotation errors rather than system failures.

Second, to address strict evaluation rules that fail to recognize semantically equivalent results, adopt flexible evaluation metrics such as partial credit scoring (rewarding queries that return a superset or subset of required columns) or schema-normalized comparison (treating different valid representations of the same entity as equivalent). When deploying NL2SQL systems, provide users with query explanation mechanisms that clarify what information the generated SQL retrieves, enabling users to verify semantic correctness even when result formatting differs from expectations.

Third, to handle semantic ambiguity in natural language questions, implement ambiguity detection and clarification mechanisms: use LLM-based analysis to identify potentially ambiguous terms (e.g., "school" could map to multiple columns), generate clarification questions for users, or provide multiple query interpretations with explanations. For benchmark developers, we strongly advocate for systematic quality assurance protocols including: (1) multi-annotator consensus requirements for gold SQL creation, (2) automated consistency checking to detect annotation errors (e.g., mismatches between questions asking for "names" and gold queries returning IDs), and (3) explicit ambiguity annotations marking questions with multiple valid interpretations.

Furthermore, we recommend developing semantically-aware evaluation frameworks that can recognize and appropriately credit semantically equivalent but syntactically different SQL queries, moving beyond simple string matching or single-reference execution accuracy. These quality improvements are essential not only for fair benchmark evaluation but also for training more robust NL2SQL systems, as models trained or evaluated on noisy benchmarks may learn to replicate annotation errors rather than developing genuine semantic understanding.

A.7 Case Study on Leveraging NL2SQLBench for NL2SQL System Development

This section serves as the case study of the usability and recommendations on leveraging **NL2SQLBench** for NL2SQL system development. We follow the systematic workflow proposed in Sec 4.7.

First, establish a performance profile by running NL2SQLBench with different baselines on your target dataset to generate a comprehensive performance profile across all three core modules. This profile identifies the primary bottleneck. For Example, if Schema Selection F1-score < 75%, schema linking represents the critical constraint; if Candidate Generation Error Rate > 10%, focus on reducing execution errors through better prompting or intermediate representations; if Query Revision Correctness Improvement < 5% despite high Error-to-Correct rates, the revision module may be ineffective at addressing the dominant Incorrect query problem.

Second, apply targeted optimizations to the bottleneck module while monitoring cross-module impacts. For instance, improving schema precision from 32% to 85% not only enhances Schema Selection F1-score but also simplifies downstream candidate generation by reducing irrelevant context, potentially enabling faster generation strategies. Use NL2SQLBench to measure these cascading effects and validate that optimizations to one module do not inadvertently degrade others.

Third, conduct cost-benefit analysis at the module level using our effectiveness and efficiency metrics. Compare approaches like $\text{CHESS}_{(\text{IR,SS,CG})}$ (63.43% accuracy, 310K tokens, 80 LLM calls) against MAC-SQL (60.82% accuracy, 7.5K tokens, 4 LLM calls) to determine whether marginal accuracy gains justify 40× cost increases for your specific application requirements. Define clear operating constraints—such as maximum latency budget (e.g., < 500ms end-to-end), cost budget (e.g., < $0.01 per query), or minimum acceptable accuracy (e.g., > 60% execution accuracy)—and use NL2SQLBench's fine-grained metrics to navigate the accuracy-efficiency trade-off space systematically. For example, latency-critical applications should favor approaches with minimal LLM calls (≤ 5), while accuracy-critical applications may allocate larger computational budgets to high-precision schema selection and multi-candidate generation

Finally, leverage NL2SQLBench for A/B testing and continuous improvement. When evaluating alternative strategies for a specific module (e.g., Few-Shot CoT vs. Preliminary-SQL for schema selection), use the framework's standardized metrics to make evidence-based decisions, and continuously benchmark production systems to detect performance degradation over time.

By treating NL2SQL development as a modular optimization problem where each component can be independently analyzed, improved, and composed, practitioners can systematically construct systems that optimally balance accuracy, efficiency, and deployment constraints, thus transforming NL2SQLBench from a benchmarking tool into a practical framework for iterative system development.

A.8 Technical Details of the Strategies for the Key Modules.

A.8.1 Schema Selection Module.

Few-Shot Chain-of-Thought (CoT). This strategy utilizes the few-shot in-context learning [10, 30] capabilities of LLMs supplemented by CoT prompting [7, 65] to identify relevant tables and columns for NL queries, enabling LLMs to generalize without extensive task-specific training. By guiding the LLM through a step-by-step process, the CoT method helps the model better interpret the NL query and effectively select relevant tables and columns. However, the effectiveness of this approach heavily relies on the quality of the examples and CoT reasoning steps provided in the prompt. Poorly chosen examples can lead to suboptimal schema selection. Designing effective reasoning steps for diverse scenarios requires careful prompt engineering.

MAC-SQL [61] selected few-shot demonstrations that illustrate how to map NL queries to expected schema selection results. These examples are incorporated into the prompt to guide the model's decision-making process. Building on the few-shot framework, DIN-SQL [48] adopts the CoT method, which involves structuring the prompt to include intermediate reasoning steps in the few-shot examples to explicitly describe how the relevance of schema elements is determined. By guiding the LLM through a step-by-step

process, the CoT method helps the model better interpret the NL query and systematically select relevant tables and columns. An example prompt (from MAC-SQL) is shown as below:

As an experienced and professional database administrator, your task is to analyze a user question and a database schema to provide relevant information. The database schema consists of table descriptions, each containing multiple column descriptions. Your goal is to identify the relevant tables and columns based on the user question and evidence provided.

[Instruction]
1. Discard any table schema that is not related to the user question and evidence.
2. Sort the columns in each relevant table in descending order of relevance and keep the top 6 columns.
3. Ensure that at least 3 tables are included in the final output JSON.
4. The output should be in JSON format.

[Requirements]
1. If a table has less than or equal to 10 columns, mark it as "keep_all".
2. If a table is completely irrelevant to the user question and evidence, mark it as "drop_all".
3. Prioritize the columns in each relevant table based on their relevance.

Here is a typical example:

==========
[DB_ID] banking_system
[Schema]
Table: account
[
(account_id, the id of the account. Value examples: [11382, 11362, 2, 1, 2367].),
(district_id, location of branch. Value examples: [77, 76, 2, 1, 39].),
(frequency, frequency of the acount. Value examples: ['POPLATEK MESICNE', 'POPLATEK TYDNE', 'POPLATEK PO OBRATU'].),
(date, the creation date of the account. Value examples: ['1997-12-29', '1997-12-28'].)
]
Table: client
[
(client_id, the unique number. Value examples: [13998, 13971, 2, 1, 2839].),
(district_id, location of branch. Value examples: [77, 76, 2, 1, 39].)
...
]
Table: loan
[
(loan_id, the id number identifying the loan data. Value examples: [4959, 4960, 4961].),
(account_id, the id number identifying the account. Value examples: [10, 80, 55, 43].),
...
]
Table: district
[
(district_id, location of branch. Value examples: [77, 76].),
(A2, area in square kilometers. Value examples: [50.5, 48.9].),
(A4, number of inhabitants. Value examples: [95907, 95616].),
(A5, number of households. Value examples: [35678, 34892].),
(A6, literacy rate. Value examples: [95.6, 92.3, 89.7].),
(A7, number of entrepreneurs. Value examples: [1234, 1456].),
...
]
[Foreign keys]
client.‘district_id‘ = district.‘district_id‘
[Question]
What is the gender of the youngest client who opened account in the lowest average salary branch? **[Evidence]**
Later birthdate refers to younger age; A11 refers to average salary
[Answer]
"'json
{
"account": "keep_all",
"client": "keep_all",
"loan": "drop_all",
"district": ["district_id", "A11", "A2", "A4", "A6", "A7"]
}
"'
Question Solved.
==========

Here is a new example, please start answering:

[DB_ID] {db_id}
[Schema]
{desc_str}
[Foreign keys]
{fk_str}
[Question]
{query}
[Evidence]
{evidence}
[Answer]

Multi-Stage Schema Pruning. By progressively narrowing the database schema in a multi-step manner, this strategy aims to ensure that only the necessary schema are considered. Although effective, this strategy has some drawbacks. The multi-stage pruning approach introduces additional computational steps at each stage of schema pruning, which can cumulatively extend the overall time and increase the token cost. This may pose challenges in applications where low latency and low token cost are critical. For instance, C3-SQL [11] first selects relevant tables based on a ranking mechanism and self-consistency method, followed by a column recall step that retrieves relevant columns within the selected candidate tables. CHESS$_{(IR,SS,CG)}$ [60] leverages locality-sensitive hashing and vector retrieval to efficiently identify relevant database values and catalogs, then applies a three-stage schema pruning process guided by the retrieved information: first, broadly filtering out irrelevant columns; next, selecting the most pertinent table; and

Figure 17: The Schema Selection workflow employed by CHESS.

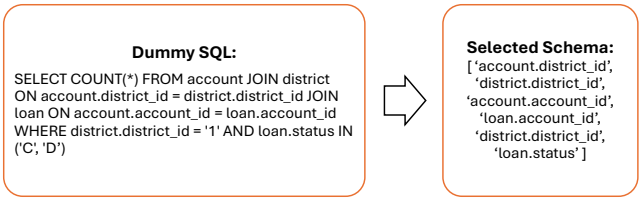


Figure 18: An example of Schema Selection strategy employed by TA-SQL.

finally, precisely identifying relevant columns within the chosen table. The workflow employed by $\text{CHESS}_{(IR,SS,CG)}$ is shown in Figure 17. OpenSearch-SQL [68] employs a similar way: information retrieval and column filtering - but chooses to recall relevant schema by leveraging both LLMs and vector retrieval.

Preliminary-SQL Enhanced. This strategy offers a novel approach to schema selection in NL2SQL pipelines by leveraging the inherent strengths of LLMs in SQL generation. Instead of explicitly instructing LLMs on schema selection, this method begins by prompting the LLMs to generate a preliminary SQL query based on the natural language query. Once such a query is generated, relevant schema elements, such as tables, columns, and relationships, are extracted from it. The primary purpose of this method is not to produce an executable SQL query but to exploit the model's familiarity with SQL generation, producing an unrefined SQL query that serves as a rough interpretation of the user's intent. TA-SQL [53] is a notable example of employing this strategy. It first generates a dummy SQL query and then extracts schema entities. An example of the strategy employed by TA-SQL is shown in the Figure 18. PET-SQL [31] and GSR [2] also adopt similar methods.

While novel, the effectiveness of this strategy heavily relies on the accuracy and completeness of the preliminary SQL query. Additionally, the intermediate step of generating this query introduces additional processing overhead, potentially increasing the latency of the overall NL2SQL pipeline.

A.8.2 Candidate Generation Module.

Few-Shot CoT. By explicitly incorporating intermediate reasoning steps in few-shot NL2SQL generation examples, few-shot CoT methods help LLMs process and solve complex logical processes in NL2SQL. For example, DIN-SQL [48] employs human-designed CoT frameworks for different types of NL questions, as shown below:

```
# Create SQL queries for the given questions.

Table advisor, columns = [*,s_ID,i_ID]
Table classroom, columns = [*,building,room_number,capacity]
Table course, columns = [*,course_id,title,dept_name,credits]
Table department, columns = [*,dept_name,building,budget]
Table instructor, columns = [*,ID,name,dept_name,salary]
Table prereq, columns = [*,course_id,prereq_id]
Table section, columns = [*,course_id,sec_id,semester,year,building,
room_number]
Table student, columns = [*,ID,name,dept_name,tot_cred]
Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]
Table teaches, columns = [*,ID,course_id,sec_id,semester,year]
Table time_slot, columns = [*,time_slot_id,day,start_hr,start_min,end_hr,end_min]

Q: "Find the buildings which have rooms with capacity more than 50."
SQL: SELECT DISTINCT building FROM classroom WHERE capacity > 50

Q: "Find the room number of the rooms which can sit 50 to 100 students and their buildings."
SQL: SELECT building , room_number FROM classroom WHERE capacity BETWEEN 50 AND 100

Q: "Give the name of the student in the History department with the most credits."
SQL: SELECT name FROM student WHERE dept_name = 'History' ORDER BY tot_cred DESC LIMIT 1
...

Use the schema links to generate the correct sqlite SQL query for the given question. Hint helps you to write the correct sqlite SQL query.
### Schema of the database with sample rows and column descriptions: #
schema
columns_descriptions # Q: question
Hint: hint
Schema_links: schema_links
```

Query Classification. This strategy classifies NL queries into different classes according to their complexity and uses different prompts for each class. DIN-SQL [48] is a representative example that employs a classification strategy to generate SQL candidates, of which the prompt employed is shown as below:

```
# For the given question, classify it as EASY, NON-NESTED, or NESTED based on nested queries and JOIN.

if need nested queries: predict NESTED
elif need JOIN and don't need nested queries: predict NON-NESTED
elif don't need JOIN and don't need nested queries: predict EASY

Table advisor, columns = [*,s_ID,i_ID]
Table classroom, columns = [*,building,room_number,capacity]
Table course, columns = [*,course_id,title,dept_name,credits]
Table department, columns = [*,dept_name,building,budget]
Table instructor, columns = [*,ID,name,dept_name,salary]
Table prereq, columns = [*,course_id,prereq_id]
```

Table section, columns = [*,course_id,sec_id,semester,year,building, room_number]
Table student, columns = [*,ID,name,dept_name,tot_cred]
Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]
Table teaches, columns = [*,ID,course_id,sec_id,semester,year]
Table time_slot, columns = [*,time_slot_id,day,start_hr,start_min,end_hr,end_min]
Foreign_keys = [course.dept_name = department.dept_name,instructor.dept_name = department.dept_name,section.building = classroom.building,section.room_number = classroom.room_number, section.course_id = course.course_id,teaches.ID = instructor.ID,teaches.course_id = section.course_id,teaches.sec_id = section.sec_id,teaches.semester = section.semester, teaches.year = section.year,student.dept_name = department.dept_name,takes.ID = student.ID,takes.course_id = section.course_id,takes.sec_id = section.sec_id,takes.semester = section.semester,takes.year = section.year,advisor.s_ID = student.ID, advisor.i_ID = instructor.ID,prereq.prereq_id = course.course_id,prereq.course_id = course.course_id]

Q: "Find the buildings which have rooms with capacity more than 50."
schema_links: [classroom.building,classroom.capacity,50]
A: Let's think step by step. The SQL query for the question "Find the buildings which have rooms with capacity more than 50." needs these tables = [classroom], so we don't need JOIN.
Plus, it doesn't require nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we don't need JOIN and don't need nested queries, then the the SQL query can be classified as "EASY".
Label: "EASY"

Q: "What are the names of all instructors who advise students in the math depart sorted by total credits of the student."
schema_links: [advisor.i_id = instructor.id,advisor.s_id = student.id,instructor.name, student.dept_name,student.tot_cred,math]
A: Let's think step by step. The SQL query for the question "What are the names of all instructors who advise students in the math depart sorted by total credits of the student." needs these tables = [advisor,instructor,student], so we need JOIN.
Plus, it doesn't need nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we need JOIN and don't need nested queries, then the the SQL query can be classified as "NON-NESTED".
Label: "NON-NESTED"

Q: "How many courses that do not have prerequisite?"
schema_links: [course.*,course.course_id = prereq.course_id]
A: Let's think step by step. The SQL query for the question "How many courses that do not have prerequisite?" needs these tables = [course,prereq], so we need JOIN.
Plus, it requires nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = ["Which courses have prerequisite?"].
So, we need JOIN and need nested queries, then the the SQL query can be classified as "NESTED".
Label: "NESTED"

For the given question, classify it as EASY, NON-NESTED, or NESTED based on nested queries and JOIN. if need nested queries: predict NESTED elif need JOIN and don't need nested queries: predict NON-NESTED elif don't need JOIN and don't need nested queries: predict EASY
###
Schema of the database with sample rows and column descriptions:
schema
columns_descriptions # Q: question
Hint: hint
Schema links: schema_links

Query Decomposition. The decomposition-based strategy extends the CoT approach by decomposing a problem into smaller, manageable components [69]. The decomposer agent introduced by MAC-SQL [61] breaks the query into a series of intermediate steps, such as sub-questions, and generates corresponding sub-queries for each step before generating the final SQL. An example prompt (from MAC-SQL) is shown as below:

Given a **[Database schema]** description, a knowledge **[Evidence]** and the **[Question]**, you need to use valid SQLite and understand the database and knowledge, and then decompose the question into subquestions for text-to-SQL generation.

When generating SQL, we should always consider constraints:
[Constraints]
- In 'SELECT <column>', just select needed columns in the [Question] without any unnecessary column or value
- In 'FROM <table>' or 'JOIN <table>', do not include unnecessary table
- If use max or min func, 'JOIN <table>' FIRST, THEN use 'SELECT MAX(<column>)' or 'SELECT MIN(<column>)'
- If [Value examples] of <column> has 'None' or None, use 'JOIN <table>' or 'WHERE <column> is NOT NULL' is better
- If use 'ORDER BY <column> ASC|DESC', add 'GROUP BY <column>' before to select distinct values

========== Example 1:

[Database schema]
...
[Foreign keys]
frpm.'CDSCode' = satscores.'cds'
[Question]
List school names of charter schools with an SAT excellence rate over the average.
[Evidence]
...

Decompose the question into sub questions, considering **[Constraints]**, and generate the SQL after thinking step by step:

Sub question 1: Get the average value of SAT excellence rate of charter schools.
SQL
”’ sql
SELECT AVG(CAST(T2.‘NumGE1500‘ AS REAL) / T2.‘NumTstTakr‘)
FROM frpm AS T1
INNER JOIN satscores AS T2
ON T1.‘CDSCode‘ = T2.‘cds‘
WHERE T1.‘Charter School (Y/N)‘ = 1
”’

Sub question 2: List out school names of charter schools with an SAT excellence rate over the average.
SQL
”’ sql
SELECT T2.‘sname‘
FROM frpm AS T1
INNER JOIN satscores AS T2
ON T1.‘CDSCode‘ = T2.‘cds‘
WHERE T2.‘sname‘ IS NOT NULL
AND T1.‘Charter School (Y/N)‘ = 1
AND CAST(T2.‘NumGE1500‘ AS REAL) / T2.‘NumTstTakr‘ > (
SELECT AVG(CAST(T4.‘NumGE1500‘ AS REAL) / T4.‘NumTstTakr‘)
FROM frpm AS T3
INNER JOIN satscores AS T4
ON T3.‘CDSCode‘ = T4.‘cds‘
WHERE T3.‘Charter School (Y/N)‘ = 1
)
”’

Question Solved.

========== Example 2:
…

==========

[Database schema]
{desc_str}
[Foreign keys]
{fk_str}
[Question]
{query}
[Evidence]
{evidence}

Decompose the question into sub questions, considering [Constraints], and generate the SQL after thinking step by step:

Intermediate Representation. To bridge the gap between NL and SQL queries, intermediate representations (IRs) [14, 17, 26], such as Pandas-like or SQL-like codes, have been introduced to facilitate the generation of SQL queries. TA-SQL [53] employs Pandas-like code as an IR, DIN-SQL adopts the IR from NatSQL [14] while OpenSearch-SQL invents an SQL-like language to encourage LLMs

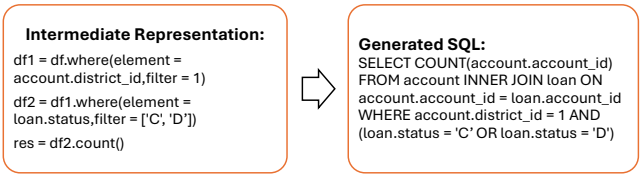


Figure 19: An example of Candidate Generation strategy employed by TA-SQL.

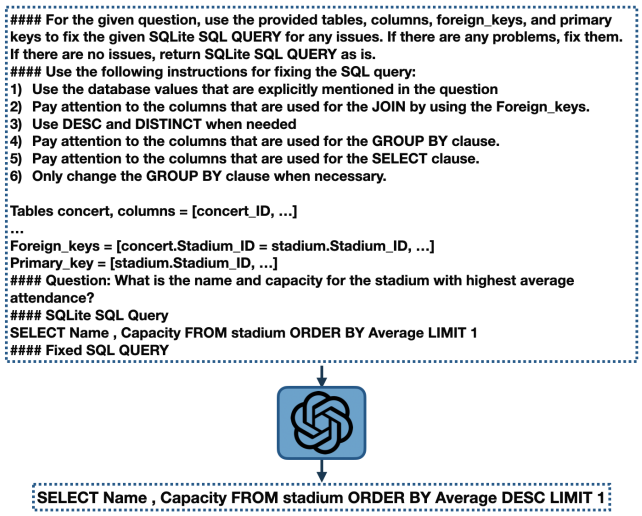


Figure 20: An example of Query Revision strategy employed by DIN-SQL.

to focus more on logic before generating final SQL queries. An example of the Intermediate Representation employed by TA-SQL is whoen in Figure 19.

Multiple Candidate Generation. Some recent works choose to increase LLM calls or sampling numbers and generate multiple candidates before selecting the final answers among them. This strategy has been implemented in studies such as C3-SQL [11], CHESS [60], MCS-SQL [21] and OpenSearch-SQL [68].

A.8.3 Query Revision Module.

LLM-Based. Similar to Self-Refine [40], generally, the process begins by presenting the candidate SQL query to LLMs alongside the original natural language query. In a zero-shot setting, the model is explicitly instructed to review the SQL query for potential issues, such as syntax errors, semantic misalignments, or missing components, and produce a corrected SQL query. DIN-SQL [48] implements this strategy to input the candidate SQL and the original NL query to LLMs and then instruct LLMs to identify and correct issues such as syntax errors, semantic mismatches, or missing elements. An example of the prompt of DIN-SQL is demonstrated in Figure 20. The module relies on clear, instruction-based prompts to guide LLMs through error detection and correction, enhancing the overall accuracy of the generated SQL queries.

Execution-Guided. This strategy leverages the execution results of candidate SQL queries as a critical feedback mechanism to improve the accuracy.. This process can be iterative, allowing

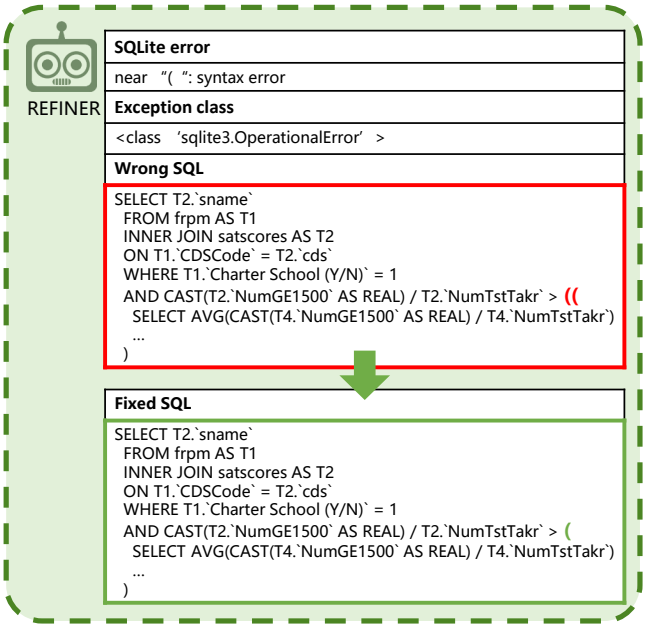


Figure 21: An example of Query Revision strategy employed by MAC-SQL.

multiple execution rounds, feedback analysis, and refinement until predefined termination conditions. MAC-SQL [61] employs a dedicated Refiner Agent that automates the error detection and correction process. Similarly, CHESS [60] starts with an initial draft query and executes it to gather feedback. The execution results, including error messages or outputs, are provided to the LLM, which adjusts and refines the SQL query accordingly. An example of the Execution-Guided employed by MAC-SQL is shown in Figure 21.

Consistency-Based. Following the principle of self-consistency [63], this strategy generates multiple SQL queries through diverse reasoning paths, evaluates their execution results, and selects the most consistent query as the final output. C3-SQL [11] incorporates a Consistency Output module, where multiple SQL queries are executed and filtered, and a voting mechanism is applied to the execution results to identify the most consistent query. CHESS [60] selects the most consistent SQL query from three samples. The demonstration of the Consistency-Based startegy is shown in Figure 22.

Unit-Test-Based. This strategy first generates multiple unit tests to highlight the differences between the candidate queries. Then, it selects the best query based on the evaluation results of the unit tests. To the best of our knowledge, $\text{CHESS}_{(\text{IR,CG,UT})}$ is the first approach that introduces this novel strategy.

Question Rewriting. This strategy aims to better guide LLMs in revising and constructing SQL queries by enriching or rewriting the natural language questions. For example, E-SQL [4] enriches the questions by incorporating relevant database items, candidate predicates, and SQL generation steps. DART-SQL [41] utilizes database content to clarify and disambiguate questions through rewriting.

Ranking and Selection. Given multiple candidate queries, this strategy ranks the candidate queries based on some criteria and then selects the top-1 query from the query pool. A representative

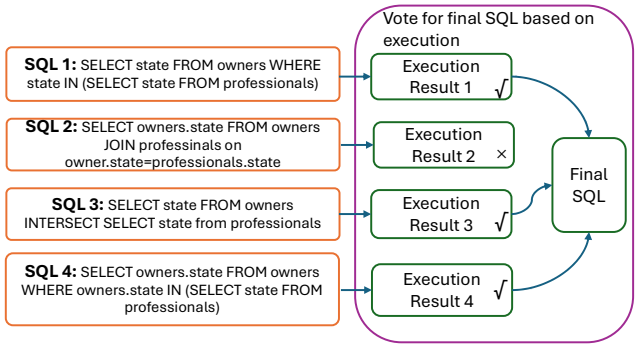


Figure 22: An example of Query Revision strategy employed by C3-SQL.

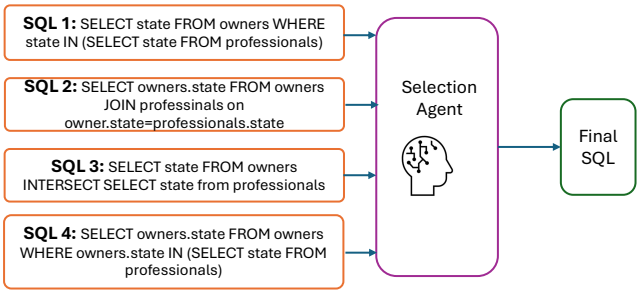


Figure 23: An example of Query Revision strategy employed by CHASE-SQL.

approach employing this method is CHASE-SQL [47], which fine-tunes a model specialized for ranking and selecting queries. The workflow employed by CHASE-SQL is demonstrated in Figure 23.

A.9 Additional Related Work

DIN-SQL [48] splits the task into schema linking, classification, SQL generation, and self-correction, enabling LLMs to produce more precise SQL queries. MAC-SQL [61] decomposes the question into a series of sub-questions and generates corresponding sub-queries before generating the final SQL. CHESS [60] proposes harnessing contextual information by efficient retrieval and schema pruning methods to enhance SQL generation. TA-SQL [53] introduces alignment-based strategies to mitigate hallucinations in SQL generation. CHASE-SQL [47] enhances the reasoning process of SQL generation by multiple CoT strategies. OpenSearch-SQL [68] uses dynamic few-shot and multiple alignment mechanisms to enhance the performance. ROUTE [52] proposes a multitask tuning paradigm and multitask collaboration strategy for NL2SQL.

A.10 Interactive Perspective Analysis

We provide a case study of the analysis of error propagation and interaction relationships between the Schema Selection module and Candidate Generation module using **NL2SQLBench**. Table 23 reports schema selection recall under four outcome conditions: when the Candidate Generation (CG) module produces a correct vs. wrong SQL, and when the Query Revision (QR) module produces a correct

vs. wrong SQL. We also list multi-candidate methods at different Pass@k values.

Higher schema recall strongly correlates with success downstream. For nearly all approaches, the recall measured on queries that end up correct is consistently higher than the recall measured on queries that end up wrong. This pattern indicates error propagation: if the schema selection stage omits a needed table or column, CG is forced to compose with incomplete parts, and QR rarely recovers because the missing elements were never surfaced. In short, low early recall makes later stages unrecoverable.

Two distinct failure regimes appear. A contrasting behavior is visible for breadth-heavy systems (e.g., OpenSearch-SQL): even when the final SQL is wrong, schema recall remains very high (0.95), only a few points below the "correct" condition (0.98). Here the schema stage over-selects (high recall, lower precision), so errors arise downstream from semantic composition—wrong join path, mis-scoped filters, or aggregations—not from missing schema items. By comparison, structured pipelines such as $\text{CHESS}_{(\text{IR,SS,CG})}$ show larger recall drops on wrong cases (e.g., 0.92→0.82), which is a recall-limited regime: missing tables/columns at the schema stage directly cause downstream failure. **Implication:** mitigation must match regime—improve recall (question-aware retrieval, bridge-table priors) for recall-limited systems; improve precision and reasoning checks (join verifiers, aggregation/constraint audits) for precision-limited systems.

Pass@k does not fix low-recall inputs. Increasing k changes candidate breadth, but the schema recall per outcome remains essentially flat across k (e.g., C3-SQL correct recall 0.985→0.982 from k=1 to 20; wrong recall 0.929→0.927). Thus, more samples cannot compensate for missing schema. If low recall is detected, the system should revisit the schema step (e.g., raise the table/column budget, rerun selection with different cues) before spending tokens on additional CG/QR attempts.

Table 23: Interactive analysis (between the Schema Selection module and Candidate Generation module.) on BIRD Dev + DeepSeek-V3. We report schema selection recall (higher is better) under four scenarios: Candidate Generation module generate correct SQL, wrong SQL, and Query Revision module generate correct SQL, wrong SQL. For methods producing multiple candidates, we list results at different Pass@k.

Approach	Candidate Generation		Query Revision	
	Correct Recall	Wrong Recall	Correct Recall	Wrong Recall
C3-SQL (Pass@1)	0.9848	0.9291	0.9837	0.9300
C3-SQL (Pass@5)	0.9834	0.9277	0.9837	0.9300
C3-SQL (Pass@10)	0.9837	0.9268	0.9837	0.9300
C3-SQL (Pass@15)	0.9829	0.9269	0.9837	0.9300
C3-SQL (Pass@20)	0.9817	0.9273	0.9837	0.9300
DIN-SQL	0.9332	0.8203	0.9316	0.8215
MAC-SQL	0.9946	0.9763	0.9930	0.9778
$\text{CHESS}_{(\text{IR,SS,CG})}$	0.9197	0.8209	0.9135	0.8218
TA-SQL	0.9466	0.8500	–	–
GSR	0.9208	0.8142	0.9142	0.8132
RSL-SQL	0.9494	0.9204	0.9495	0.8986
OpenSearch-SQL (Pass@1)	0.9831	0.9547	0.9811	0.9572
OpenSearch-SQL (Pass@5)	0.9821	0.9538	0.9811	0.9572
OpenSearch-SQL (Pass@10)	0.9815	0.9543	0.9811	0.9572
OpenSearch-SQL (Pass@15)	0.9814	0.9540	0.9811	0.9572
OpenSearch-SQL (Pass@20)	0.9812	0.9538	0.9811	0.9572

Insight: *Schema recall is a important indicator of end-to-end success. A shortfall at the schema selection stage propagates and is rarely corrected later; pipelines should monitor recall and re-enter schema selection when coverage is low. Additionally, More candidates are not a remedy for missing schema.*