IESR: Efficient MCTS-Based Modular Reasoning for Text-to-SQL with Large Language Models

Anonymous ACL submission

Abstract

Text-to-SQL is a key natural language processing task that maps natural language questions to SQL queries, enabling intuitive interaction with web-based databases. Although current methods perform well on benchmarks like BIRD and Spider, they struggle with complex reasoning, domain knowledge, and hypothetical queries, and remain costly in enterprise deployment. To address these issues, we propose a framework named **IESR**(**I**nformation **E**nhanced **S**tructured **R**easoning) for moderate-scale Large Language Models: (i) leverages Large Language Models (LLMs) for key information understanding and schema linking, and decoupling mathematical computation and SQL generation, (ii) integrates a multi-path reasoning mechanism based on Monte Carlo Tree Search (MCTS) with majority voting, and (iii) introduces a trajectory consistency verification module with a discriminator model to improve accuracy and consistency. Experimental results demonstrate that **IESR** achieves SOTA under the XiYanSQL setting on the complex reasoning benchmark LogicCat (24.28 EX) and the Archer dataset (37.28 EX), using only compact moderate-scale models without fine-tuning. Furthermore, our analysis reveals that current coder models exhibit notable biases and deficiencies in physical knowledge, mathematical computation, and common-sense reasoning, highlighting important directions for future research. We release the code in an anonymous repository for reproducibility.

1 Introduction

Text-to-SQL aims to translate natural language queries into executable SQL, enabling intuitive interaction with databases and reducing manual effort (Zelle and Mooney, 1996; Lei et al., 2025). Recent advances have reported strong performance on widely used benchmarks such as Spider (Yu et al., 2019) and BIRD (Wretblad et al., 2024), achieving execution accuracies above 90% and 70% respectively (Pourreza et al., 2025; Talaei et al., 2024; Li et al., 2025c). However, these benchmarks largely reflect simplified settings, where many queries can be solved through shallow pattern matching and limited reasoning. In contrast, recent benchmarks including LogicCat (Liu et al., 2025a) and Archer (Zheng et al., 2024) reveal fundamental limitations of existing Text-to-SQL systems by requiring cross-domain reasoning that integrates mathematical computation, physical units, common-sense constraints, and hypothetical conditions. Under such settings, current methods either degrade sharply or depend on substantially larger models and inference budgets, exposing a flawed assumption that schema grounding, logical reasoning, and numerical computation can be reliably resolved within a single or locally consistent generation pass. Some work (Xu et al., 2025; Yao et al., 2026; Weng et al., 2025) explores reinforcement learning and test-time optimization for Text-to-SQL using execution feedback, heuristic rewards, or consistency signals. While improving correctness, these methods often rely on heavy rollouts or learned critics and treat numerical computation as implicit, mak-

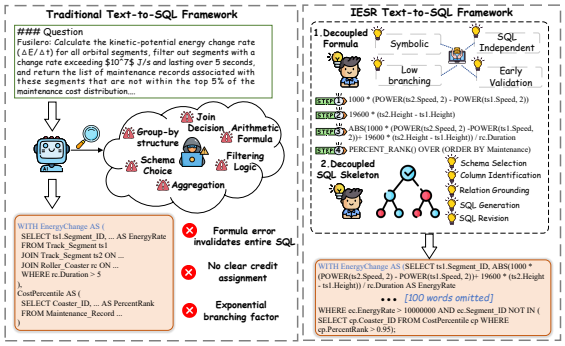

Figure 1: Motivation for Decoupling Mathematical Computation and SQL Generation in Text-to-SQL.

Code: https://anonymous.4open.science/r/IESR-B828.

ing rewards coarse in math-intensive queries and unstable under tight budgets.

Multi-agent and iterative reasoning have been explored to address this issue (Talaei et al., 2024; Pourreza et al., 2025; Deng et al., 2025), but general-purpose agent and planning paradigms (Wei et al.; Wang et al., 2023; Shinn et al., 2023; Yao et al., 2023; Song et al., 2025) remain ill-suited to the structured and high-stakes nature of SQL execution. In particular, heterogeneous decision factors—such as schema selection, join construction, aggregation, filtering logic, grouping structure, and arithmetic formulas—are often tightly entangled during generation, so that minor deviations in numerical or semantic reasoning can invalidate an otherwise correct SQL structure. This tight coupling leads to unclear credit assignment, error propagation, and instability under multi-step reasoning (Xia et al., 2024).

To address these limitations, we propose **IESR** (**I**nformation **E**nhanced **S**tructured **R**easoning) for moderate-scale Large Language Models, a modular reasoning framework for complex Text-to-SQL generation and hereafter referred to as IESR. Unlike previous search-based or agent-based works (Li et al., 2025b; Cen et al., 2025; Wang et al., 2025b) that still treat mathematical reasoning as part of SQL generation, our formulation explicitly separates symbolic computation as an independent reasoning dimension. As illustrated in Figure 1, IESR is motivated by the observation that mathematical computation is structurally orthogonal to SQL construction once relevant numerical attributes are identified. Accordingly, IESR decomposes the task into three tightly coupled stages that explicitly compensate for the failure modes of moderate-scale large language models: (i) an *information understanding* stage that extracts semantic hypotheses and performs schema-guided compression, (ii) an *MCTS-based reasoning* stage that explores multiple SQL generation trajectories with decoupled reasoning dimensions, and (iii) a *trajectory selection* stage that verifies and aggregates candidate SQL paths.

Without instruction fine-tuning, IESR operates entirely with moderate-scale open-source large language models and achieves strong performance on complex mathematical, physical, and hypothetical queries. Experiments demonstrate that IESR improves reasoning-oriented Text-to-SQL performance under fixed inference-time search budgets, with the accuracy-cost trade-off controlled by the rollout number. Ablation studies further confirm the importance of information understanding, structured search, and consistency-based verification in improving robustness and execution accuracy.

The main contributions of this work are summarized as follows:

- We propose **IESR**, a structured MCTS-based reasoning framework that integrates information understanding, schema linking, and trajectory-level verification for complex Text-to-SQL tasks.
- We demonstrates strong performance under moderate-scale LLMs, no-finetuning settings on reasoning-oriented benchmarks.
- We show that **IESR** is compatible with moderate-scale open-source large language models and requires no instruction fine-tuning, enabling robust Text-to-SQL reasoning under moderate-scale backbone settings.

2 Related Work

2.1 Text-to-SQL with Decomposition and Search-based Reasoning

Recent advances in large language models (LLMs) have substantially improved Text-to-SQL performance on benchmarks such as Spider and BIRD. Many methods decompose the task into candidate generation, refinement, and selection stages to better handle schema alignment and complex query structures (Li et al., 2024; Cao et al., 2024; Pourreza et al., 2025). Beyond static decomposition, recent work introduces explicit search to improve exploration and robustness. Alpha-SQL (Li et al., 2025b) formulates Text-to-SQL as a value-guided tree search problem, while SQL-o1 (Lyu et al., 2025) and MCTS-SQL (Yuan et al., 2025) incorporate self-reward heuristics and Monte Carlo Tree Search to guide structured exploration. Related approaches further exploit multi-trajectory reasoning or mutual verification to enhance robustness (Qi et al., 2025). IESR is related to Alpha-SQL (Li et al., 2025b) in using MCTS for Text-to-SQL, but differs in the search unit. Alpha-SQL mainly explores SQL trajectories with execution-derived rewards, whereas IESR searches over heterogeneous reasoning actions that separately model schema grounding, entity linking, numerical reasoning, SQL generation, and revision. This explicit action decomposition is designed for queries where numerical dependencies and domain constraints

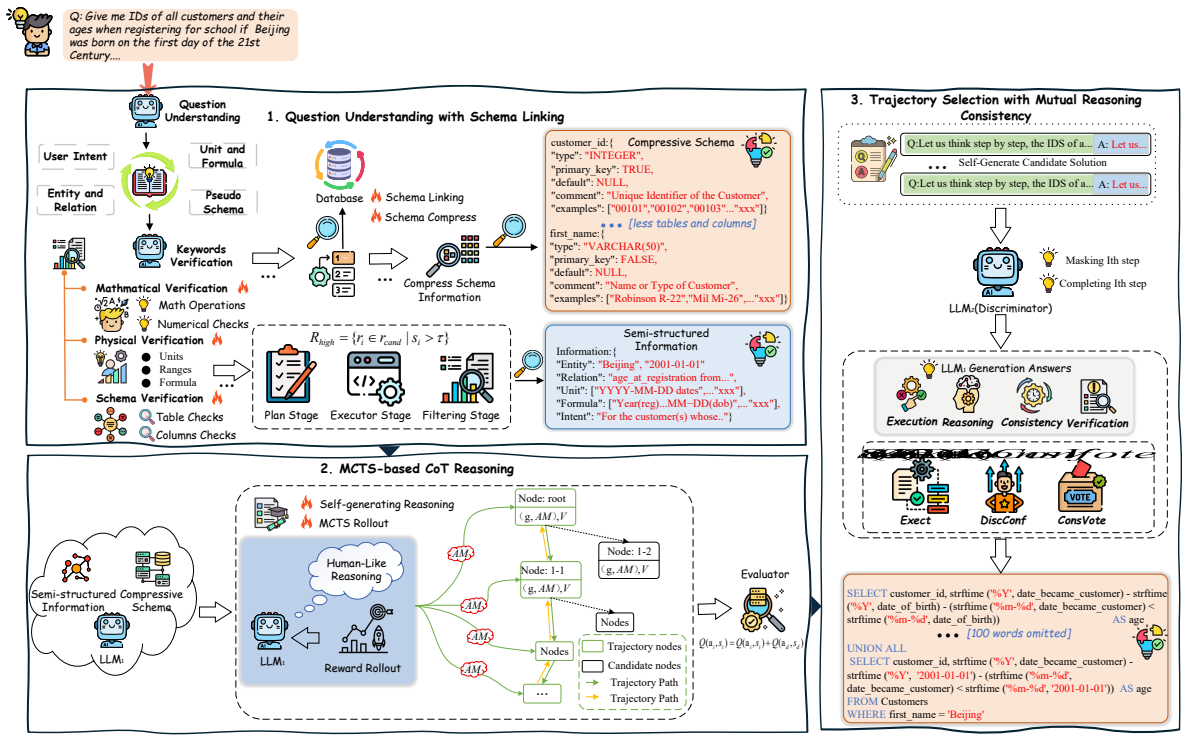

Figure 2: The comprehensive workflow of IESR including three stages: Question Understanding with Schema Linking, Monte Carlo Tree Search(MCTS)-based Reasoning and Trajectory Selection with Mutual Reasoning Consistency.

cannot be reliably handled by locally scored SQL trajectories.

2.2 Structured Reasoning and Optimization for Text-to-SQL

Recent work further improves Text-to-SQL via structured reasoning and optimization at training or inference time, targeting semantic faithfulness under limited computational budgets. Multi-turn and tool-integrated frameworks, such as MTIR-SQL (Xu et al., 2025), optimize generation trajectories through reinforcement signals, while Arctic-Text2SQL-R1 (Yao et al., 2025) shows that simple reward designs can already induce strong reasoning behaviors (Yao et al., 2026). To reduce execution overhead, Graph-Reward-SQL introduces execution-free reward modeling via graph matching and stepwise supervision (Weng et al., 2025). In parallel, retrieval-aware optimization mitigates schema context cost: in-context reinforcement learning with retrieval-augmented generation improves database and table selection under constrained context windows (Toteja et al., 2025). Test-time scaling strategies further allocate inference budgets to improve robustness, as explored in Agentar-Scale-SQL (Wang et al., 2025a). IESR also differs from CHASE-SQL (Pourreza et al., 2025) and DeepEye-SQL (Li et al., 2025a). CHASE-SQL focuses on multi-path candidate generation and preference-based final selection, while DeepEye-SQL emphasizes schema analysis and SQL refinement. In contrast, IESR verifies intermediate reasoning trajectories through masking-and-completion consistency and explicitly separates symbolic computation from SQL construction, making it more targeted toward math-intensive and domain-constrained Text-to-SQL.

3 Methodology

Motivated by the need to extract structured semantics and user intent from natural language questions and propagate them into schema linking and multi-step reasoning. Figure 2 presents IESR, a modular Text-to-SQL framework with three components: question understanding, MCTS-based reasoning, and trajectory selection. IESR first extracts structured semantics and aligns them with relevant schema, then explores multiple reasoning trajectories via MCTS, and finally select the most reliable SQL.

3.1 Question Information Understanding

Intent and Information Understanding. To support schema selection and downstream reasoning, we first map each multi-domain query into an intermediate *latent semantic state*, which encodes query-implied entities, relations, numerical cues, units, and field patterns before schema grounding. Detailed algorithm listed at Appendix C.2.

Given a query q and schema $S = (T, C)$, we use a moderate-scale LLM to produce a latent semantic state $\mathcal{S}_q = (i, r, E, \mathcal{R}, N, U, P)$, where E denotes extracted entities, $\mathcal{R}$ denotes candidate relations and schema references, N denotes numeric expressions, U denotes associated units, and P denotes candidate field patterns. We treat $\mathcal{S}_q$ as a high-recall set of noisy semantic hypotheses, which is later refined through consistency verification to guide schema compression and reasoning.

Constraint-aware Relation Filtering. To mitigate hallucination and semantic drift, we introduce a set of domain-agnostic *consistency constraints* that evaluate the internal coherence of semantic hypotheses. From $\mathcal{S}_q$, we obtain an initial relation set $\mathcal{R}_{\text{init}}$. Each constraint encodes compatibility among entities, units, and transformation patterns as $C_j = (\text{Enti}_j, \text{Uni}_j, \text{Equt}_j)$.

Candidate relations are filtered via semantic compatibility:

$$\mathcal{R}_{\text{cand}} = \{\, r_i \in \mathcal{R}_{\text{init}} \mid \textsc{Match}(r_i, \mathcal{C}) \,\}. \tag{1}$$

Here, $\textsc{Match}(r_i, \mathcal{C})$ holds if there exists a constraint $C_j \in \mathcal{C}$ such that $\text{sim}(r_i, \text{Uni}_j, \text{Equt}_j) > \delta_{\text{match}}$ and $\text{sim}(\cdot)$ is implemented by a matcher $\mathcal{M}_{\text{match}}$.

Soft Consistency Scoring. The filtered relations $\mathcal{R}_{\text{cand}}$ are evaluated by a *Plan & Executor* module, which assigns a soft consistency score:

$$P_i = \text{Plan}(r_i, \mathcal{C}), \quad s_i = \text{Executor}(r_i, P_i), \tag{2}$$

where $s_i \in [0, 1]$ reflects the degree of consistency between a semantic hypothesis and the constraint set. We intentionally adopt this non-learned, constraint-based scorer to ensure robustness in low-resource settings and avoid additional supervision or training overhead.

Relations with scores above a threshold τ are retained:

$$\mathcal{R}_{\text{high}} = \{\, r_i \in \mathcal{R}_{\text{cand}} \mid s_i > \tau \,\}. \tag{3}$$

The resulting validated semantic state conditions subsequent schema linking and compression, enabling relevance estimation beyond the raw query.

Schema Linking and Compression. Conditioned on the validated semantic state from the previous stage, we perform schema linking and compression to surface task-relevant structures and constrain the reasoning search space. Following the M-Schema technique (Liu et al., 2025b), we encode schema elements in a semi-structured format with explicit annotations, and apply filtering to retain only high-salience fields. We employ two complementary strategies:

- **Locality-Sensitive Hashing:** filters tables and columns by lexical similarity, efficiently narrowing candidate schema elements.
- **Semantic Similarity Matching:** selects schema elements whose semantic representations are most relevant to the conditioned query.

Given the large size of full schema where only a small subset is relevant per query, the refined schema with key fields and annotations is provided to the reasoning agent for subsequent processing.

3.2 MCTS-based CoT Reasoning

Following prior work (Qi et al., 2025; Li et al., 2025b), we cast Text-to-SQL generation as structured search over multi-step reasoning trajectories. Complex SQL requires interdependent decisions over schema grounding, joins, aggregation, filters, and numerical computation, which moderate-scale LLMs often cannot resolve reliably in a single pass. MCTS addresses this by representing each node as a reasoning state with a partial SQL hypothesis and semantic context, each edge as a reasoning action, and each root-to-terminal path as a candidate trajectory. Its exploration-exploitation tradeoff allows promising partial solutions to be expanded while preserving alternative paths, making it more suitable than beam search or self-consistency for heterogeneous reasoning actions. After search, terminal trajectories are collected as $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$, where $n \geq 1$, and passed to the trajectory selection module. The detailed algorithm is provided in Appendix C.2.

Human-inspired Reasoning Actions. Most MCTS-based reasoning methods use a single action type, usually next-step generation (Chen et al., 2024). For complex SQL, this coarse transition is inefficient because schema grounding, numerical reasoning, and structural refinement require different operations. Each reasoning action is defined as

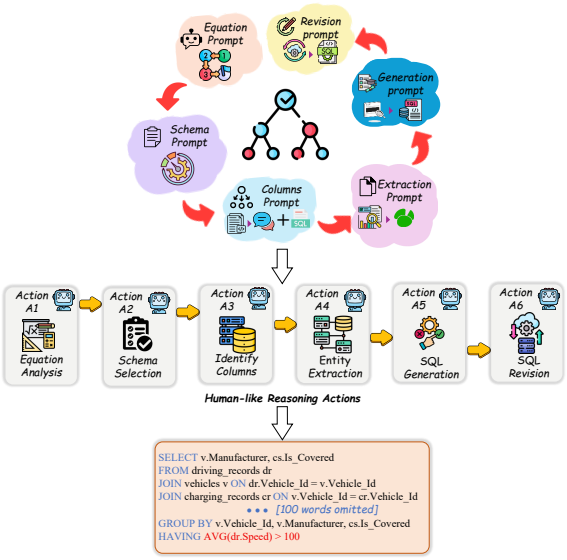

Figure 3: A visual illustration of heterogeneous MCTS actions (A1–A6) for SQL generation and reasoning.

a state transition over the partial SQL structure and semantic context. Guided by the problem-solving patterns in Figure 3, we design a heterogeneous action space for dimension-specific exploration. Each action updates the reasoning state along a specific semantic dimension, enabling more targeted exploration than generic next-step expansion. The formal node representation is provided in Appendix C.

- **Equation Analysis**: explicit modeling of formulas and numerical relations (Mao et al., 2025).
- **Schema Selection**: identifying task-relevant tables and columns from large schemas (Li et al., 2025b; Talaei et al., 2024).
- **Identify Columns**: resolving ambiguous schema fields using semi-structured semantic cues (Li et al., 2025a).
- **Entity Extraction**: grounding entities and relations for accurate filtering and joins.
- **SQL Generation**: constructing executable SQL queries, particularly nested structures (Pourreza et al., 2025).
- **SQL Revision**: correcting erroneous queries via reasoning over formulas, units, and domain knowledge.

At each step, MCTS selects an action from this space and generates the next reasoning state accordingly.

Reward-based Node Evaluation. The reward function guides trajectory quality in MCTS. In our low-resource setting, we adopt an execution-based, terminal-only self-consistency reward without learned critics, following Alpha-SQL (Li et al., 2025b). When MCTS reaches a terminal node and produces a candidate SQL y, we sample N additional SQL queries $\{y_i\}_{i=1}^{N}$ under the same terminal context and execute them on database D. The reward is defined as the agreement rate of execution results:

$$r_t = R(y, q, D) = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}\Big[\textsc{Execute}(y, D) = \textsc{Execute}(y_i, D)\Big]. \quad (4)$$

MCTS Search and Backpropagation. Given a rollout path consisting of nodes $v_0, \ldots, v_d$ and actions $a_1, \ldots, a_d$, we backpropagate the terminal reward r_t to update visit counts and cumulative values:

$$\begin{aligned} N(v_i, a_{i+1}) &\leftarrow N(v_i, a_{i+1}) + 1, \\ Q(v_i, a_{i+1}) &\leftarrow Q(v_i, a_{i+1}) + R(y, q, D), \end{aligned} \quad (5)$$

for $i = 0, \ldots, d-1$. Action selection follows the standard UCT criterion:

$$UCT(v, a) = \frac{Q(v, a)}{N(v, a)} + c\sqrt{\frac{\ln N(v)}{N(v, a)}}. \quad (6)$$

The search iterates through selection, expansion, simulation, and backpropagation. After N_{rollout} rollouts, all terminal trajectories are collected as candidate SQL queries for subsequent selection and verification.

3.3 Trajectory Selection with Mutual Reasoning Consistency

To reliably select executable and semantically consistent SQL from noisy candidates, we introduce a collaborative selection mechanism between a primary model LLM_1 and a secondary verifier LLM_2. Detailed algorithm listed at Appendix C.2.

Discriminator Consistency Verification. For a candidate reasoning trajectory $t = x \oplus s_1 \oplus s_2 \oplus \cdots \oplus s_d$, we assess its logical self-consistency via a masking-and-completion procedure. Specifically, we mask the reasoning process at an intermediate step i ($i < d$), retaining only the first $i-1$ intermediate steps such as selected tables, partial columns, or partial `WHERE` conditions. The incomplete sequence is then provided to the verifier LLM_2 for completion.

If the completion produced by LLM_2 is semantically consistent with the original trajectory t, yielding compatible schema grounding and SQL structure, we regard the trajectory as stable and retain it. This masking-and-completion check tests whether the same semantic constraints can be recovered from partial reasoning states. LLM_2 is a frozen verifier with scale comparable to LLM_1 and requires no additional supervision.

Scoring Mechanism. To select the most reliable SQL from filtered candidates, we assign a composite score to each trajectory:

$$\begin{aligned}\text{Score}(t) = \alpha \cdot \text{Exec}(t) \\ + \beta \cdot \text{DiscConf}(t) + \gamma \cdot \text{ConsVote}(t).\end{aligned} \tag{7}$$

Here, $\text{Exec}(t)$ evaluates execution correctness or equivalence when available, while $\text{DiscConf}(t)$ and $\text{ConsVote}(t)$ provide consistency signals derived from discriminator verification and peer agreement, respectively. The coefficients α, β, γ are fixed to balance executability and consistency, with consistency serving as auxiliary evidence. Detailed representation is illustrated at Appendix E. The final SQL answer is selected by maximizing $\text{Score}(t)$ over all candidate trajectories.

4 Experiments

We conduct comprehensive experiments to evaluate effectiveness, robustness, and inference-budget trade-offs on complex Text-to-SQL benchmarks, with a particular focus on mathematical, physical, and hypothetical reasoning under model settings.

4.1 Experimental Setup

Datasets. We evaluate IESR on two challenging reasoning-oriented benchmarks, LogicCat (Liu et al., 2025a) and Archer (Zheng et al., 2024), which require multi-domain reasoning including mathematics, physics, commonsense, and hypothetical analysis. Detailed datasets listed at Appendix F.

Baseline. We compare IESR with representative SOTA Text-to-SQL systems, including DIN-SQL (Pourreza and Rafiei, 2023), DAIL-SQL (Gao et al., 2023), DTS-SQL (Pourreza and Rafiei, 2024), CHESS (Talaei et al., 2024), LinkAlign (Wang and Liu, 2025), Alpha-SQL (Li et al., 2025b), Chase-SQL (Pourreza et al., 2025), Deepeye-SQL (Li et al., 2025a) and SQL-O1 (Lyu et al., 2025). These methods cover a broad range of paradigms such as decomposed prompting, schema linking, hierarchical reasoning, and MCTS-based search. Detailed public methods listed at Appendix F.

Backbone Models. For the information understanding module, we evaluate several general-purpose LLMs and select Qwen3-8B (Yang et al., 2025) for its favorable balance between reasoning capability and efficiency. For MCTS-based reasoning, we adopt four code-oriented models: Qwen2.5-Coder-7B (Hui et al., 2024), XiYanSQL-QwenCoder-7B-2504 (Gao et al., 2025), Seed-Coder-8B (Seed et al., 2025), and OmniSQL-7B (Li et al., 2025c). For trajectory consistency verification, we employ Qwen3-4B as a discriminator, which offers strong consistency assessment with minimal overhead.

Evaluation Metric. We adopt the widely used metric Execution Accuracy (EX) (Pourreza et al., 2025). This evaluates where a query is correct only if its execution result matches that of the ground-truth query.

Experiment Details. Experiments were conducted on a cloud server with four RTX 4090 GPUs. Local LLM inference was accelerated with vLLM. Baseline hyperparameters follow their original implementations, while the detailed MCTS and decoding configurations are reported in Appendix D, and all prompts are provided in Appendix H.

4.2 Main Results

Overall Performance. Table 1 reports the main results on LogicCat and Archer, where SQL generation requires joint reasoning over schema grounding, numerical computation, physical constraints, and hypothetical conditions. IESR shows stronger performance on reasoning-intensive settings under the evaluated small-backbone configurations on these reasoning-intensive settings, indicating that explicit task decomposition, MCTS-based trajectory exploration, and consistency verification are most useful when local SQL decisions are tightly coupled with external semantic constraints. We report BIRD and Spider results in Appendix A; the weaker transfer on these conventional benchmarks suggests that IESR primarily benefits complex reasoning scenarios, while standard schema-grounding tasks may favor more specialized ranking or linking pipelines.

LogicCAT Analysis. As illustrated in Figure 4, IESR achieves substantial improvements across all difficulty levels, with the largest relative gains are often observed on Hard. Further analysis by

Method	Qwen2.5-Coder	XiYanSQL	OmniSQL	Seed-Coder	Qwen2.5-Coder	XiYanSQL	OmniSQL	Seed-Coder
Size	7B			8B	7B			8B
	LogicCat EX(%)				**Archer-Dev EX(%)**			
Cot-SQL	5.07	7.89	6.93	6.58	17.16	18.46	18.14	19.89
DIN-SQL	10.31	11.56	9.56	8.98	31.45	34.41	26.04	29.13
DAIL-SQL	10.73	10.98	9.43	8.18	29.67	32.28	27.56	24.12
DTS-SQL	14.12	14.88	12.67	10.31	31.88	33.17	32.14	30.33
CHESS-SQL	16.16	19.21	10.89	14.52	32.45	35.54	27.27	**31.56**
Chase-SQL	21.92	22.13	19.76	21.56	**36.12**	35.78	27.12	28.98
LinkAlign	19.56	22.56	14.34	18.89	29.21	31.28	30.21	30.10
Alpha-SQL	18.32	19.78	16.56	19.34	28.32	30.12	29.88	31.22
Deepeye-SQL	**22.81**	21.22	22.30	21.75	19.23	21.15	17.12	13.46
SQL-O1	16.42	17.91	13.24	17.33	26.56	28.22	27.41	28.78
Standard CoT	10.07	16.83	11.56	14.53	-	-	-	-
IESR	21.66	**24.28**	**22.41**	**22.11**	35.21	**37.28**	**33.11**	30.66

Table 1: Main results on reasoning-intensive Text-to-SQL benchmarks. EX denotes execution accuracy. Bold values indicate the best result in each backbone-dataset column. SQL-O1 requires task-specific tuning.

	Qwen2.5-Coder-7B		XiYanSQL-QwenCoder-7B		Seed-coder-8B		OmniSQL-7B	
Configuration	**EX**	**Drop**	**EX**	**Drop**	**EX**	**Drop**	**EX**	**Drop**
Full Model (IESR)	**21.66**	-	**24.28**	-	**22.11**	-	**22.41**	-
Core Stage Isolation								
- Stage 1.1+2+3 Only (w/o)	19.96	1.70	21.68	2.60	20.41	1.70	21.42	0.99
- Stage 1.2+2+3 Only (w/o)	19.28	2.38	20.21	4.07	19.22	2.89	20.42	1.99
- Stage 1+2+3.1 Only (w/o)	19.24	2.42	21.83	2.45	18.29	3.82	19.14	3.27
- Stage 1+2+3.2 Only (w/o)	18.11	3.55	20.63	3.65	17.21	4.90	18.94	3.47
- Stage 1 (w/o)	11.12	10.54	14.36	9.92	9.42	12.69	12.67	9.74

Table 2: Comprehensive ablation study of IESR components across four backbone models on the LogicCat dataset. We report **EX** and the absolute performance drop in percentages (%). Understanding for Intent and Information Understanding. Linking for Schema Linking and Compression. Consistency Verification for Discriminator Consistency Verification. Reasoning and Discriminator Agent for MCTS-based CoT Reasoning and Trajectory Selection with Mutual Reasoning Consistency.

reasoning type reveals that current code-oriented LLMs perform significantly worse on physical and mathematical reasoning compared to commonsense queries, highlighting a critical limitation. IESR effectively mitigates this gap through structured reasoning and trajectory verification.

4.3 Ablation Analysis

Component Contribution Analysis. Table 2 presents ablation results for IESR. Removing any module degrades performance, with the largest drops observed when disabling MCTS-based reasoning or schema linking, indicating their central role. Consistency verification and scoring yield smaller but consistent improvements, confirming their effectiveness as reliability enhancements. More error analysis are listed at Appendix A.1.

Action Space Analysis. This experiment evaluates the contribution of individual reasoning actions in the proposed action space. Following rStar (Qi et al., 2025), we conduct action-level ablation on the LogicCat dataset by removing one reasoning action at a time while keeping all other settings unchanged. The results in Table 3 show that removing any remaining action consistently degrades performance across all backbone models, indicating that each reasoning action contributes to effective exploration. Notably, removing the SQL revision action results in the largest performance drop, highlighting its critical role in correcting intermediate reasoning errors and refining executable

Action	Qwen2.5-Coder	XiYanSQL	OmniSQL	Seed-Coder
All actions	21.66	24.28	22.11	22.41
w/o A1	21.45 (↓ 0.21)	24.02 (↓ 0.26)	21.92 (↓ 0.19)	22.23 (↓ 0.18)
w/o A2	20.83 (↓ 0.83)	23.88 (↓ 0.40)	20.68 (↓ 1.43)	22.01 (↓ 0.40)
w/o A3	20.32 (↓ 1.34)	23.25 (↓ 1.03)	19.78 (↓ 0.33)	22.12 (↓ 0.29)
w/o A4	20.86 (↓ 0.80)	23.87 (↓ 1.31)	20.89 (↓ 0.80)	21.98 (↓ 0.43)
w/o A6	20.12 (↓ **1.54**)	22.12 (↓ **2.16**)	19.21 (↓ **2.90**)	21.15 (↓ **1.26**)

Table 3: Ablation study on the reasoning action space (A1–A4, A6), corresponding to the reasoning actions defined in Section 3.2. We do not ablate A5 (SQL Generation) since removing it yields degenerate trajectories. The evaluation metric is **EX** (execution accuracy).

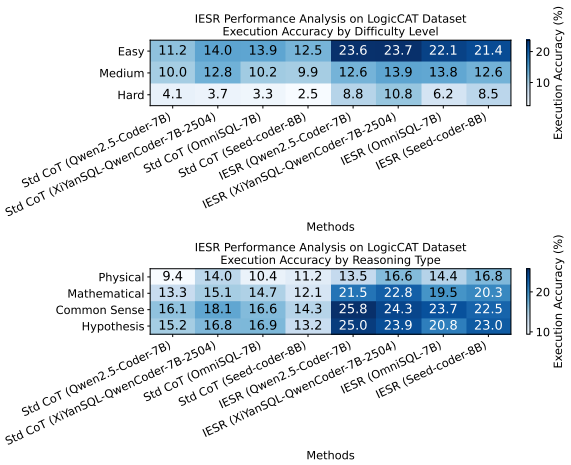

Figure 4: Performance heatmap of different methods on the LogicCat dataset across three difficulty levels (Easy, Medium, Hard) and different reasoning types.

	table		column		unit_num		operator	
	P	R	P	R	P	R	P	R
Qwen2.5-7B-Instruct	64.60	63.80	57.28	57.77	57.86	57.90	77.85	73.96
Qwen3-8B	75.25	73.54	63.32	62.30	60.80	61.06	77.02	75.36
Deepseek-V3	76.72	74.81	67.34	66.67	62.88	62.18	78.18	74.03
Deepseek-R1	78.09	75.35	62.91	62.02	60.52	60.32	76.00	72.20
Gemini2.5-Flash	87.08	92.79	77.70	82.78	78.94	80.20	92.75	**92.16**
GPT-4o	88.73	**93.02**	**79.80**	84.11	79.77	80.31	92.75	92.10
Qwen2.5-7B-Instruct+Ve	87.46	89.87	79.41	82.72	79.88	80.30	91.53	89.50
Qwen3-8B+Ve	**90.09**	91.24	78.20	**83.24**	**81.02**	**81.11**	**92.91**	91.31

Table 4: Precisions and Recalls of schema items, units, key numbers, and compute operators used in verification. **Ve** for Keywords Verification.

SQL queries.

4.4 N_{rollout} Analysis

Shown in Figure 5 disabling schema compression leads to a clear performance decrease, confirming the role of schema understanding in Text-to-SQL tasks. We observe that increasing the number of N_{rollout} generally improves the reliability and performance of the output, as greater diversity and consensus help filter out spurious results. In particular, our best results are achieved with 32 N_{rollout}, indicating that a larger ensemble improves robustness. However, as the number of N_{rollout} increases beyond 32, performance gains tend to plateau and may even decline. This is likely because excessive rollouts introduce more noisy or low-quality reasoning paths, which can dilute the effectiveness of the majority-voting strategy. Detailed Experiments are shown at Appendix A.3.

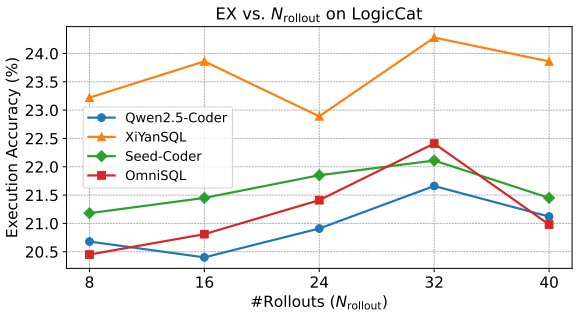

Figure 5: Ablation study of N_{rollout} across four backbone models on the LogicCat dataset.

4.5 Different Models on Information Understanding Analysis

With the Verification module in place, shown in Table 4, the extractor shifts from a pattern matching heuristic to a recall amplifying pipeline that systematically recovers missed schema cues and normalizes numbers and units without sacrificing precision. This manifests itself most strongly in the long-tail fields (*column*, *unit_num*) where add Verification delivers large double-digit recall gains, while on *table* and *operator* the verified Qwen3-8B climbs into the same performance regime as frontier models such as GPT-4o and Gemini-2.5-Flash. The improvements are not cosmetic: precision rises alongside recall, indicating that verification corrects upstream omissions and inconsistencies rather than merely relaxing decision thresholds.

5 Conclusion

We propose IESR, a MCTS based framework that integrates information understanding and schema linking for complex Text-to-SQL reasoning. Our method achieves strong reasoning-oriented performance with 7B/8B backbones and exposes a controllable accuracy-cost trade-off through the rollout budget, as validated by comprehensive ablation studies.

Limitations

While IESR demonstrates strong performance on complex reasoning–oriented Text-to-SQL benchmarks, it also has limitations. First, the framework relies on the quality of early semantic hypothesis extraction; errors in entity, unit, or formula identification may propagate into subsequent schema compression and search. Second, although MCTS enables structured exploration, it introduces additional inference cost compared to single-pass gen-

eration, and the search budget must be carefully tuned to avoid diminishing returns from noisy trajectories. Third, our reward design is execution- and consistency-based, which provides robust supervision without annotations but offers limited insight into intermediate reasoning errors. Moreover, our evaluation is conducted under controlled benchmark settings, and extending the framework to real-world databases with evolving schema or noisy execution environments remains an open challenge. Finally, we briefly discuss potential risks such as erroneous SQL execution and misinterpretation of complex queries, which could impact downstream decision-making if used without human oversight. We stress that automated Text-to-SQL systems should be deployed with appropriate validation in real-world scenarios.

References

Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. 2024. Rsl-sql: Robust schema linking in text-to-sql generation. *Preprint*, arXiv:2411.00073.

J. Cen and 1 others. 2025. Sqlfixagent: Towards semantic-accurate text-to-sql. In *AAAI 2025*.

Guoxin Chen, Minpeng Liao, Chengxi Li, and Kai Fan. 2024. Alphamath almost zero: Process supervision without process. *Preprint*, arXiv:2405.03553.

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

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

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. 2025. A preview of xiyan-sql: A multi-generator ensemble framework for text-to-sql. *Preprint*, arXiv:2411.08599.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

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*.

Boyan Li, Chong Chen, Zhujun Xue, Yinan Mei, and Yuyu Luo. 2025a. Deepeye-sql: A software-engineering-inspired text-to-sql framework. *Preprint*, arXiv:2510.17586.

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The dawn of natural language to sql: Are we fully ready? *Proceedings of the VLDB Endowment*, 17(11):3318–3331.

Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025b. Alpha-SQL: Zero-shot text-to-SQL using monte carlo tree search. In *Forty-second International Conference on Machine Learning*.

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. 2025c. Omnisql: Synthesizing high-quality text-to-sql data at scale. *Preprint*, arXiv:2503.02240.

Tao Liu, Hongying Zan, Yifan Li, Dixuan Zhang, Lulu Kong, Haixin Liu, Jiaming Hou, Aoze Zheng, Rui Li, Yiming Qiao, Zewei Luo, Qi Wang, Zhiqiang Zhang, Jiaxi Li, Supeng Liu, Kunli Zhang, and Min Peng. 2025a. Logiccat: A chain-of-thought text-to-sql benchmark for multi-domain reasoning challenges. *Preprint*, arXiv:2505.18744.

Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2025b. Xiyan-sql: A novel multi-generator framework for text-to-sql.

Shuai Lyu, Haoran Luo, Ripeng Li, Zhonghong Ou, Jiangfeng Sun, Yang Qin, Xiaoran Shang, Meina Song, and Yifan Zhu. 2025. Sql-o1: A self-reward heuristic dynamic search method for text-to-sql. *Preprint*, arXiv:2502.11741.

Xutao Mao, Tao Liu, and Hongying Zan. 2025. Steiner-sql: Graph-guided mathematical reasoning for text-to-sql generation. *Preprint*, arXiv:2509.19623.

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*.

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*.

Mohammadreza Pourreza and Davood Rafiei. 2024. DTS-SQL: Decomposed text-to-SQL with small large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 8212–8220, Miami, Florida, USA. Association for Computational Linguistics.

Zhenting Qi, Mingyuan MA, Jiahang Xu, Li Lyna Zhang, Fan Yang, and Mao Yang. 2025. Mutual reasoning makes smaller LLMs stronger problem-solver. In *The Thirteenth International Conference on Learning Representations*.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, and 8 others. 2025. Seed-coder: Let the code model curate data for itself. *Preprint*, arXiv:2506.03524.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Jinwang Song, Hongying Zan, Kunli Zhang, Lingling Mu, Yingjie Han, Haobo Hua, and Min Peng. 2025. Jolt-sql: Joint loss tuning of text-to-sql with confusion-aware noisy schema sampling. *Preprint*, arXiv:2505.14305.

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

Rishit Toteja, Arindam Sarkar, and Prakash Mandayam Comar. 2025. In-context reinforcement learning with retrieval-augmented generation for text-to-SQL. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 10390–10397, Abu Dhabi, UAE. Association for Computational Linguistics.

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

Pengfei Wang, Baolin Sun, Xuemei Dong, Yaxun Dai, Hongwei Yuan, Mengdie Chu, Yingqi Gao, Xiang Qi, Peng Zhang, and Ying Yan. 2025a. Agentar-scale-sql: Advancing text-to-sql through orchestrated test-time scaling. *Preprint*, arXiv:2509.24403.

Yihan Wang and Peiyu Liu. 2025. Linkalign: Scalable schema linking for real-world large-scale multi-database text-to-sql. *Preprint*, arXiv:2503.18596.

Ziyang Wang, Yuanlei Zheng, Zhenbiao Cao, Xiaojin Zhang, Zhongyu Wei, Pei Fu, Zhenbo Luo, Wei Chen, and Xiang Bai. 2025b. Autolink: Autonomous schema exploration and expansion for scalable schema linking in text-to-sql at scale. *Preprint*, arXiv:2511.17190.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models.

Han Weng, Puzhen Wu, Cui Longjie, Yi Zhan, Boyi Liu, Yuanfeng Song, Dun Zeng, Yingxiang Yang, Qianru Zhang, Dong Huang, Xiaoming Yin, Yang Sun, and Xing Chen. 2025. Graph-reward-SQL: Execution-free reinforcement learning for text-to-SQL via graph matching and stepwise reward. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 12917–12943, Suzhou, China. Association for Computational Linguistics.

Niklas Wretblad, Fredrik Gordh Riseby, Rahul Biswas, Amin Ahmadi, and Oskar Holmström. 2024. Understanding the effects of noise in text-to-sql: An examination of the bird-bench benchmark. *Preprint*, arXiv:2402.12243.

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

Zekun Xu, Siyu Xia, Chuhuai Yue, Jiajun Chai, Mingxue Tian, Xiaohan Wang, Wei Lin, Haoxuan Li, and Guojun Yin. 2025. Mtir-sql: Multi-turn tool-integrated reasoning reinforcement learning for text-to-sql. *Preprint*, arXiv:2510.25510.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

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

Zhewei Yao, Guoheng Sun, Lukasz Borchmann, Gaurav Nuti, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. 2026. Arctic-text2sql-r1: Simple rewards, strong reasoning in text-to-sql. *Preprint*, arXiv:2505.20315.

Zhewei Yao and 1 others. 2025. Arctic-text2sql-r1: Simple rewards, strong reasoning in text-to-sql. *arXiv preprint*.

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. 2019. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Preprint*, arXiv:1809.08887.

Shuozhi Yuan, Limin Chen, Miaomiao Yuan, and Zhao Jin. 2025. Mcts-sql: Light-weight llms can master the text-to-sql through monte carlo tree search. *Preprint*, arXiv:2501.16607.

JohnM. Zelle and RaymondJ. Mooney. 1996. Learning to parse database queries using inductive logic programming. *National Conference on Artificial Intelligence,National Conference on Artificial Intelligence*.

Danna Zheng, Mirella Lapata, and Jeff Pan. 2024. Archer: A human-labeled text-to-SQL dataset with arithmetic, commonsense and hypothetical reasoning. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 94–111, St. Julian's, Malta. Association for Computational Linguistics.

A Experimental Analysis

A.1 Comparison on General datasets

Table 5 reports results on BIRD-Dev and Spider. These benchmarks are included to characterize the transfer behavior of IESR beyond reasoning-intensive queries. The results show that explicit multi-path reasoning does not consistently improve standard Text-to-SQL benchmarks, where specialized schema-linking and candidate-ranking methods remain highly competitive.

A.2 Comparison with Baseline LLMs

In this section, we evaluate the baseline performance of LLMs on the LogicCat dataset, categorizing them into general LLMs (such as GPT-4o) and reasoning-oriented LLMs (such as DeepSeek-R1), all under the same Text-to-SQL prompt. As shown in Table 6, among the baseline models, o4-mini achieves the highest standalone accuracy (14.96%), outperforming both general-purpose models such as GPT-4.1 and reasoning-focused models such as Gemini-2.5-Flash. However, IESR, even when paired with a relatively moderate-scale large 7B parameter model, exceeds the compared zero-shot LLM prompting baselines on LogicCat under the same prompt, including o4-mini, highlighting the strength of the framework in targeting reasoning optimization. Furthermore, to demonstrate the plug-and-play advantage of IESR, we extended the experiments to Qwen2.5-7B-Instruct and Qwen3-8B. Compared to direct prompting, IESR substantially increases their execution accuracy by 17.0% and 16.5%, respectively, validating both the generalizability and effectiveness of the framework across various inference models.

A.3 Detailed N_{rollout} Analysis

As shown in Figure 5, disabling schema compression leads to a consistent performance drop across all backbones, highlighting the importance of schema understanding in constraining the search space for Text-to-SQL reasoning. Without compression, the reasoning agent must explore a larger set of tables and columns, which increases the likelihood of semantically mis-grounded yet executable SQL trajectories and reduces the stability of downstream aggregation.

We further observe that increasing N_{rollout} generally improves performance up to $N_{\text{rollout}} = 32$, where all models achieve their best results. This improvement can be attributed to enhanced exploration coverage and more reliable consensus estimation: a larger rollout budget increases the probability of sampling high-quality reasoning trajectories while reducing variance through majority voting.

However, performance gains saturate and may slightly decline beyond this point. We attribute this behavior to noise amplification effects: additional rollouts increasingly sample low-probability or semantically inconsistent trajectories, which can dilute the majority signal and introduce competition between incorrect but internally consistent modes. As a result, excessive rollouts do not provide independent evidence but may instead reinforce systematic biases of the backbone model.

Overall, these results indicate that a moderate rollout budget strikes the best balance between exploration and aggregation reliability. In practice, $N_{\text{rollout}} = 32$ offers a strong trade-off between robustness and noise, while larger budgets yield diminishing or even negative returns.

A.4 Perturbation Experiment Analysis

We evaluated the robustness of IESR with a systematic perturbation study on 500 LogicCat questions with explicit JOIN intent. We construct three decay levels of increasing distractor complexity and compare IESR to standard prompting across four backbones, controlling for model capacity so that performance gaps reflect the framework itself.

- **L1-Semantic:** Semantically related tables with misleading column names.
- **L2-Structural:** Similar table names with irrelevant columns or redundant JOIN paths.
- **L3-Hybrid Advanced:** Semantically relevant

Method Method	Qwen2.5-Coder	XiYanSQL	OmniSQL	Seed-Coder	Qwen2.5-Coder	XiYanSQL	OmniSQL	Seed-Coder
Size	7B			8B	7B			8B
	BIRD-Dev EX(%)				**Spider EX(%)**			
Cot-SQL	31.22	33.58	37.58	30.16	65.32	69.12	68.32	66.23
DIN-SQL	51.78	52.45	51.23	49.88	77.19	79.30	76.80	78.01
DAIL-SQL	52.12	51.16	52.23	50.66	80.31	82.56	81.56	82.98
DTS-SQL	58.32	60.78	61.56	58.18	84.12	85.09	84.99	83.21
CHESS-SQL	56.80	60.13	57.96	57.22	80.31	83.30	77.02	81.31
Chase-SQL	58.99	61.68	56.23	62.17	**85.11**	85.12	81.42	80.15
Deepeye-SQL	63.13	67.27	**66.56**	**65.21**	83.38	**86.89**	**85.14**	83.78
LinkAlign	58.90	61.32	60.56	54.78	80.11	85.89	82.32	82.12
SQL-O1	66.70	67.21	61.39	64.67	85.10	86.54	81.12	**85.25**
Alpha-SQL	**66.80**	**67.92**	62.32	65.13	84.00	84.93	79.34	84.37
IESR	54.80	62.22	56.22	58.96	83.45	83.36	80.02	79.22

Table 5: Results on conventional Text-to-SQL benchmarks. These results are reported for completeness and clarify the scope of IESR. IESR is primarily designed for reasoning-intensive queries, while specialized Text-to-SQL systems remain stronger on standard schema-grounding benchmarks.

Model	Accuracy (EX%)
Deepseek-V3	8.70
GPT-4o	10.17
GPT-4.1	13.57
Gemini-2.5-Pro	10.17
Claude-3.7-Sonnet	13.99
Claude-4.0-Sonnet	11.51
Seed-Coder-8B-Reasoning	4.16
DeepSeek-R1	9.98
o4-mini	**14.96**
o3-mini	9.11
Qwen3-235B-A22B	10.52
Qwen3-Coder-480B-A35B-Instruct	9.42
+ IESR (Qwen2.5-Coder-7B)	**21.66 (↑ 6.70)**
+ IESR (XiYanSQL-QwenCoder-7B-2504)	**24.28 (↑ 9.32)**
+ IESR (OmniSQL-7B)	**22.41 (↑ 7.45)**
+ IESR (Seed-Coder-8B)	**22.11 (↑ 7.15)**
+ IESR (Qwen2.5-7B-Instruct)	**16.69 (↑ 1.73)**
+ IESR (Qwen3-8B)	**18.23 (↑ 3.27)**

Table 6: Comparison with Baseline LLMs on the LogicCat dataset. Comparing with o4-mini, **EX** shows Execution Accuracy.

	Qwen2.5-Coder		OmniSQL		Seed-Coder		XiYanSQL	
Configuration	**EX**	**Drop**	**EX**	**Drop**	**EX**	**Drop**	**EX**	**Drop**
SC@maj32	**21.66**	–	**24.28**	–	**22.11**	–	**22.41**	–
Ablation $N_{rollout}$								
SC@maj8	20.68	0.98	23.22	1.06	21.18	0.93	20.45	1.96
SC@maj16	20.40	1.26	23.86	0.42	21.45	0.66	20.81	1.60
SC@maj24	20.91	0.75	22.89	1.39	21.85	0.26	21.41	1.00
SC@maj40	21.12	0.54	23.86	0.42	21.45	0.66	20.98	1.43

Table 7: Ablation study of N_{rollout} across four backbone models on the LogicCat dataset. We report **EX** and the absolute performance drop (%). **XiYanSQL** refers to XiYanSQL-QwenCoder-7B-2504; **Qwen2.5-Coder** to Qwen2.5-Coder-7B; **OmniSQL** to OmniSQL-7B; and **Seed-Coder** to Seed-Coder-8B.

and structurally valid decays that violate specific constraints.

For each question, we create four database variants: *DB_Original*, *DB_Decay_L1*, *DB_Decay_L2*, and *DB_Decay_L3*, testing both methods across all environments.

Figure 7 presents the experimental results in all combinations of model-environment combinations, revealing the superior robustness of IESR against adversarial database perturbations. The results demonstrate IESR's consistent outperformance of standard prompting across all environments, with performance gaps that widen as adversarial complexity increases. In clean databases, IESR establishes a clear baseline advantage that expands progressively through L1 semantic decays and L2 structural decays and culminates in the largest gap under advanced hybrid decays L3.

IESR exhibits better resilience against adversarial perturbations, showing more graceful degradation compared to the steep decline in standard prompting. This robustness advantage stems from the framework's dual-component architecture: MCTS-based reasoning Configuration systematically breaks down complex queries while Trajectory Selection bridging ensures consistent relationship traversal, effectively avoiding the semantically plausible but logically incorrect decay paths that mislead standard approaches.

A.5 Stability Analysis

Figure 6 evaluates the robustness of IESR under different random seeds. We compare IESR with the strongest baseline on each dataset using the same backbone, decoding configuration, rollout budget,

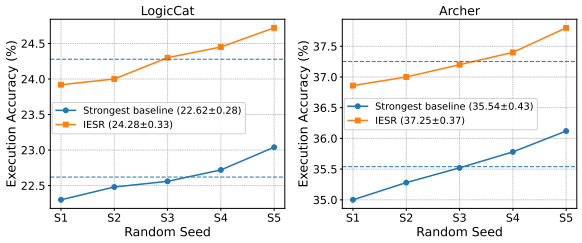

Figure 6: Execution accuracy across five random seeds on LogicCat and Archer. Solid lines show per-seed EX for the strongest baseline and IESR. Dashed horizontal lines indicate the mean EX, and shaded bands denote the 95% confidence interval across seeds.

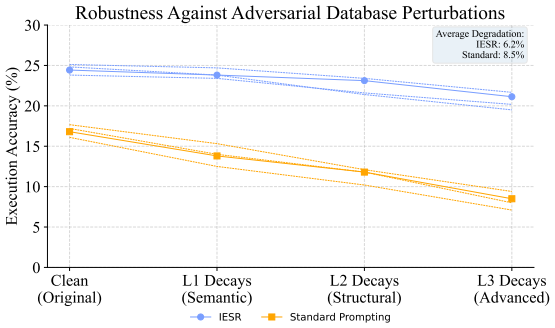

Figure 7: Robustness comparison between IESR and standard prompting across adversarial database environments. IESR maintains superior performance and exhibits more graceful degradation as decay sophistication increases from clean databases to advanced hybrid decays.

and execution backend. Across five independent runs, IESR consistently outperforms the strongest baseline on both LogicCat and Archer. On LogicCat, IESR achieves an average EX of 24.28% with a standard deviation of 0.33, compared with 22.56% and 0.35 for the strongest baseline. On Archer, IESR reaches 37.28% with a standard deviation of 0.37, compared with 35.54% and 0.43 for the strongest baseline. Although IESR shows comparable variance due to stochastic MCTS exploration and trajectory selection, its lower confidence bound remains above the upper confidence bound of the strongest baseline on both datasets. This indicates that the improvement is stable across random seeds and is unlikely to result from a favorable sampling instance.

A.6 Efficiency Analysis

IESR introduces additional inference cost because it treats Text-to-SQL generation as a search problem rather than a single-pass decoding problem. Its overhead comes from two sources: semantic preprocessing and MCTS-based trajectory exploration. The preprocessing stage is relatively moderate-scale, as shown in Table 10, requiring only 5.1 calls and 2,803.2 tokens per query on LogicCat, and 6.2 calls and 3,424.7 tokens on Archer. Therefore, the dominant cost does not come from schema understanding or semantic extraction, but from repeatedly expanding, evaluating, and selecting candidate reasoning trajectories during search.

Table 8 compares IESR with representative Text-to-SQL systems on LogicCat. With $N_{\text{rollout}} = 32$, IESR achieves 21.66 EX using 43.8 average calls and 80.7K tokens per query. The token cost is higher than most baselines because each rollout is not a minimal SQL-generation call. Instead, a rollout contains richer intermediate context, including decomposed task information, schema-grounded reasoning states, candidate SQL construction, execution-oriented feedback, and trajectory-level verification signals. This makes the average token cost per call substantially larger than methods that mainly rely on direct generation, retrieval-augmented prompting, or shallow self-consistency. Compared with Chase-SQL, IESR uses fewer calls but a comparable token budget, which suggests that IESR shifts the cost from call frequency to context-rich trajectory construction. Compared with Alpha-SQL, IESR obtains higher EX under a similar search-oriented setting, but consumes more tokens because its reasoning traces and verifier prompts preserve more state information across candidate trajectories.

Table 9 further shows that increasing N_{rollout} does not translate into monotonic accuracy improvement. Accuracy peaks at $N_{\text{rollout}} = 32$ with 21.66 EX, while larger rollout budgets from $N = 40$ to $N = 64$ consume more calls and tokens but do not yield consistent gains. This pattern indicates a saturation effect in the search space: after a moderate number of rollouts, additional trajectories are more likely to revisit semantically similar query plans, explore low-quality branches, or amplify noisy verifier preferences. The cost increase is also sublinear with respect to the nominal rollout budget. For example, increasing the budget from $N = 32$ to $N = 64$ doubles the rollout limit, but

the average token usage rises only from 80.7K to 95.8K. This is because the rollout number is an upper bound: pruning, early termination, invalid-branch filtering, and shared preprocessing reduce the effective amount of newly generated context. However, the reduced marginal cost does not imply improved efficiency, since the additional tokens bring little or no accuracy gain.

We therefore select $N_{\text{rollout}} = 32$ as the default setting. It provides the best execution accuracy in the rollout sweep and represents the best trade-off between search depth and inference budget. In practice, the wall-clock cost can be further reduced through batched rollout decoding, prompt caching, trajectory reuse, and parallel execution across GPUs, while preserving the same logical search procedure.

Method	EX(%)	Avg. calls	Avg. total tokens
Din-SQL	10.31	3.2	2.5K
DAIL-SQL	10.73	3.4	2.3K
CHESS	16.16	20.2	7.5K
Alpha-SQL	18.32	61.1	45.2K
Chase-SQL	21.92	125.2	83.1K
Deepeye-SQL	22.81	21.0	43.1K
LinkAlign	19.56	42.2	19.3K
DTS-SQL	14.12	9.2	11.4K
IESR(N=32)	21.66	43.8	80.7K

Table 8: Inference costs of different methods on LogicCat. We report execution accuracy, average model calls, and average total tokens per question.

Method	EX(%)	Avg. calls	Avg. total tokens
IESR, N=4	20.68	11.7	22.1K
IESR, N=8	20.68	23.9	25.9K
IESR, N=16	20.40	33.2	45.7K
IESR, N=32	21.66	43.8	80.7K
IESR, N=40	21.12	48.9	83.4K
IESR, N=48	20.21	54.4	89.6K
IESR, N=56	21.39	60.7	91.9K
IESR, N=64	21.15	63.9	95.8K

Table 9: Effect of the rollout budget on IESR inference cost and execution accuracy on LogicCat. Larger rollout budgets increase the search budget, but accuracy saturates after $N_{\text{rollout}} = 32$.

B Fine-grained Error Analysis

We conduct a fine-grained failure analysis of the IESR framework on 500 randomly sampled LogicCat examples. Figure 8 reports the error distribution. The analysis reveals that the remaining failures are not dominated by a single source; instead, execution accuracy is constrained by a coupled decision chain involving schema grounding, value normalization, query-plan construction, domain-formula application, and answer-schema preservation.

	LogicCat	Archer
Avg. calls	5.1	6.2
Avg. total tokens	2,803.2	3,424.7

Table 10: Information understanding costs of IESR on LogicCat and Archer. We report the average number of moderate-scale model calls and generated tokens required for semantic extraction and schema-aware preprocessing per query.

- **Knowledge-Grounded Reasoning.** *Incorrect Knowledge* accounts for the largest fraction of errors (30.4%). These errors arise when the model fails to apply the domain operation required by the question, including unit conversion, formula selection, physical-quantity interpretation, entity-relation reasoning, and vague intent resolution. Cases C and D illustrate this issue: the generated SQL is executable, but the computed denotation is wrong because the model uses an incorrect statistical definition or an invalid physical constant. This suggests that SQL executability alone is insufficient for these tasks; the verifier must also judge whether the generated program implements the intended domain computation.

- **Compositional Query Planning.** *Wrong Query* errors constitute 20.8% of the sampled failures. In these cases, the model often selects a globally wrong computational strategy while still producing syntactically valid SQL. Typical patterns include over-restricting the query scope, using an inappropriate aggregation, or replacing a required derived computation with a counting objective. Cases B and F demonstrate that the model can partially ground the relevant table or value but still fail to preserve the target answer semantics. This indicates that the main difficulty is not always locating the right schema element; it is often maintaining the correct transformation sequence from question intent to executable query.

- **Tuple-Schema Preservation.** *Other Detail* errors account for 20.0% of the failures. These

errors are local in form but decisive under strict execution matching. Case H shows a representative projection failure: the join path, arithmetic expression, and row cardinality are largely correct, yet the prediction omits the required key column, causing a tuple-schema mismatch. This type of error suggests that evaluation-sensitive details, such as projected columns, output arity, numeric safeguards, and result ordering assumptions, should be explicitly represented in search and verification.

- **Schema–Dialect Grounding.** *Schema Linking* errors (12.2%) include hallucinated tables, wrong schema elements, and incorrect keyword-to-column grounding, as shown in Case A. *SQL Syntax* errors (10.6%) are typically caused by dialect-level constraints, such as unquoted reserved identifiers, as shown in Case G. Although these categories are smaller than reasoning and planning errors, they remain important because they cause execution failure before denotation can be compared.

Overall, the error profile suggests that future improvements should not only strengthen schema retrieval or syntax repair, but also introduce verification signals that evaluate whether a candidate SQL query preserves the intended denotation. In particular, robust text-to-SQL agents for LogicCat-style tasks need to check four properties jointly: whether the schema elements are valid, whether categorical values are normalized to database literals, whether the query plan implements the intended domain computation, and whether the projected answer schema matches the benchmark target.

C Algorithm

C.1 Action Nodes

Inspired by human reasoning processes, shown in Table 3, we design a diverse action space consisting of five reasoning actions:

- **Equation Analysis:** Building on prior work (Mao et al., 2025) that emphasizes the explicit expression of formulas, the action space is introduced to enable their explicit representation.

- **Schema Selection:** Databases typically contain large and complex schema, while individual queries often involve only a small subset of them. This discrepancy poses significant challenges for SQL generation. Prior work (Li et al., 2025b; Talaei et al., 2024) has addressed this issue by designing schema selection mechanisms to identify relevant schema components, and we similarly leverage chain-of-thought reasoning to select the most relevant columns.

- **Identify Columns:** Database schema often contain ambiguous fields such as *ids, Keys, Columns* and special function like *POWER(x, 2)* rather than $x * x$ and so on, while specific queries refer to concrete numerical values, leading to ambiguous semantic references during SQL generation. Prior work (Li et al., 2025a) and existing datasets (Lei et al., 2025; Wretblad et al., 2024) have highlighted this

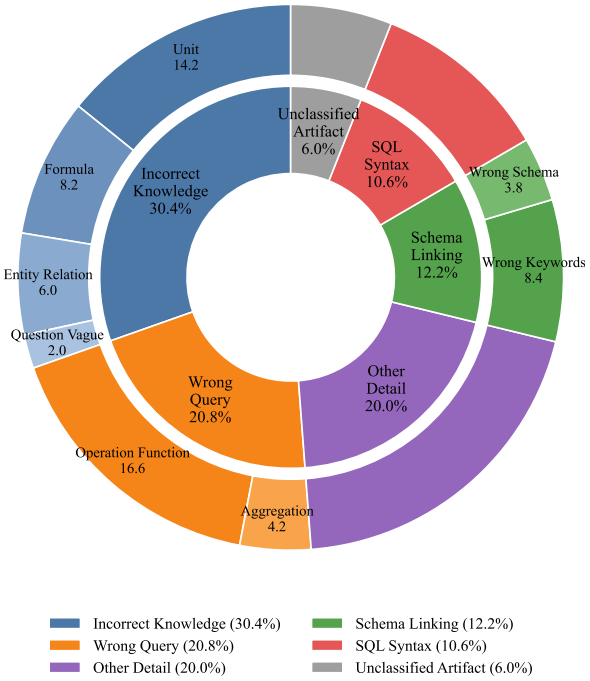

Figure 8: Distribution of Errors on Sampled Set.

Case A: Schema Hallucination (QID: 101)
Outcome: `status = pred_error`

Observed behavior:
- ✗ Hallucinated table: `emergency_response_plans`
- ✓ Existing table: `historicaldataanalysis`

Server error:
`(1146, "Table 'alarmsystem.emergency_response_plans' doesn't exist")`

Final SQL (abridged)
```
FROM historicaldataanalysis h
INNER JOIN emergency_response_plans r ON h.EventType = r.EventType
WHERE h.EventType = 'tornado'
  AND h.MaxSeverity = 8
  AND h.FalseAlarmRate = 2
```

Interpretation:
The model invents a plausible domain table that is absent from the alarmsystem schema, so execution fails before result comparison.

Figure 9: Case A: schema hallucination caused by a non-existent generated table.

Case B: Query-Scope / Cardinality Error (QID: 1)
Outcome: status = mismatch

Execution results:
Gold result: 761.999966, 1292.199953, 646.800011
Predicted result: 761.999966

Observed behavior:
✓ Correct quantity: Total_Disk_Area × Max_Disk_Loading
✗ Over-restricted scope: WHERE Aircraft_ID = 1

Final SQL
```
SELECT Total_Disk_Area * Max_Disk_Loading AS Lift
FROM aircraft
WHERE Aircraft_ID = 1;
```

Interpretation:
The model derives the right formula but restricts the query to one entity, returning a single row instead of the intended multi-row answer set.

Figure 10: Case B: query-scope error where a multi-row answer is restricted to one entity.

Case C: Statistical Definition / Filter Mismatch (QID: 100)
Outcome: status = mismatch

Execution results:
Gold: 33.33333 Predicted: 75.0

Observed behavior:
✓ Computes a percentage statistic
✗ Uses an incorrect filter or denominator

Final SQL
```
SELECT CAST(COUNT(CASE WHEN MaxSeverity > 8.0 THEN 1 ELSE NULL END) AS REAL) * 100
       / COUNT(MaxSeverity) AS percentage_exceeding_8_0
FROM historicaldataanalysis;
```

Interpretation:
The query expresses a percentage, but the numerator condition, denominator, or underlying population differs from the benchmark definition.

Figure 11: Case C: denotation mismatch caused by an incorrect statistical filter or denominator.

issue. The purpose of this dynamic space is to leverage semi-structured information extracted during the information extraction stage to identify such ambiguous fields and explicitly resolve their semantic references.

- **Entity Extraction:** This step extracts relevant schema information based on entities, relations, and formulas identified from the semi-structured information. A text-to-SQL system is required not only to accurately capture filtering conditions and user intent, but also to ensure the correctness of the extracted SQL data.

Case D: Physical Constant / Unit Reasoning Error (QID: 107)
Outcome: status = mismatch

Execution results:
Gold: 20.0 Predicted: 3.636364

Observed behavior:
✗ Hard-coded propagation speed: 5.5
✗ Incorrect unit or constant mapping

Final SQL
```
SELECT (radiuskm / 5.5) AS PropagationTime
FROM WarningEvent
WHERE radiuskm = 20;
```

Interpretation:
The SQL is executable, but the model injects an incorrect physical constant, producing a wrong propagation time under the benchmark formula.

Figure 12: Case D: unit-reasoning error caused by an incorrect physical constant.

Case F: Categorical Value Grounding / Case Mismatch (QID: 103)
Outcome: status = mismatch

Execution results:
Gold result: empty ([]) — gold SQL filters on EventType = 'flood' (lowercase)
Predicted result: [[0]] — a scalar count, not the gold propagation-time value

Observed behavior:
✗ Uses 'Flood' (capital F) instead of the benchmark literal 'flood' (lowercase)
✓ Returns COUNT(*) = 0 instead of the required propagation-time formula

Final SQL
```
SELECT COUNT(*) FROM historicaldataanalysis
    WHERE EventType = 'Flood' AND AverageSeverity = 5;
```

Interpretation:
The model uses 'Flood' (capital F) instead of the benchmark literal 'flood', so the filter matches no row under case-sensitive MySQL string comparison; it then returns COUNT(*) = 0 rather than the gold formula (50 * 1000) / 2. This is a categorical value grounding failure.

Figure 13: Case F: categorical value grounding error caused by case-sensitive value mismatch.

Case G: Pure SQL Syntax Error (QID: 9)
Outcome: status = pred_error

Server error:
(1064, "You have an error in your SQL syntax; check the manual ... near 'MATCH m ON p.Pilot_ID = m.Winning_Pilot' at line 19")

Observed behavior:
✗ Uses reserved identifier MATCH without quoting
✓ Expected quoting form: `match`

Final SQL (abridged)
```
FROM pilot p
LEFT JOIN MATCH m ON p.Pilot_ID = m.Winning_Pilot
LEFT JOIN MATCH m2 ON p.Pilot_ID = m2.Losing_Pilot
WHERE p.Name = 'Prof. Zackery Collina';
```

Interpretation:
The database contains a table named match, but MySQL parses unquoted MATCH as syntax rather than a table identifier, so execution fails before result comparison.

Figure 14: Case G: SQL dialect error caused by an unquoted reserved identifier.

- **SQL Generation:** SQL generation is the core component of Text-to-SQL systems (Pourreza et al., 2025). This method particularly excels at handling nested queries. We incorporate this strategy into our reasoning action space.

- **SQL Revision:** We observe that many errors from CHASE (Pourreza et al., 2025) and CHESS (Talaei et al., 2024) are due to the difficulty of the model in understanding implicit physical formulas and domain-specific background knowledge. This action requires the model to list all known conditions and explicitly express all relevant formulas, units, key fields, and information, and then correct the

Case H: Result-Schema / Projection Detail (QID: 150)
Outcome: status = mismatch

Execution results:
Gold result: 49 rows of (BicycleID, ActualPower), e.g., (1, 285.0), (2, 225.0), (3, 184.0)
Predicted result: 49 scalar powers only, e.g., 285.0, 225.0, 184.0

Observed behavior:
✓ Join keys and arithmetic are largely correct
✗ BicycleID is omitted from the projected answer schema

Final SQL
```
SELECT T1.RiderPowerOutput * (T2.PowerTransferEfficiency / 100)
       AS ActualPowerDelivered
FROM bicyclerider AS T1
INNER JOIN bicycleperformance AS T2 ON T1.BicycleID = T2.BicycleID;
```

Interpretation:
The row cardinality matches gold, but the selected projection drops the primary-key column required by strict execution tuple matching.

Figure 15: Case H: projection error caused by omitting the required key column.

SQL query accordingly.

The search process consists of four steps:

- **Selection**: Starting from the root node, actions are selected downward along the tree based on the UCT values until an unexplored node or a terminal node is encountered.
- **Expansion**: Valid actions are expanded at the selected node to generate new child nodes.
- **Simulation**: From the newly expanded node, actions are sampled randomly and expansion continues until a terminal node is reached.
- **Backpropagation**: The reward of the terminal node is computed and propagated back along the path, updating the statistics of all nodes encountered.

After multiple N_{rollout} rollouts, all complete reasoning trajectories reach terminal nodes are collected, forming a candidate set of SQL queries for subsequent selection and verification.

At each step, the MCTS selects an action from this space and generates the next reasoning step based on the current state, with the aim of maximizing the reasoning capability of LLM_1.

C.2 Module Algorithm

The proposed framework consists of three stages, each summarized in the appended algorithms. First, *Information Understanding with Rule-Guided Verification Algorithm* converts a natural language query into structured elements (intent, entities, relations, numbers, units, operators) and filters them with a rule base to ensure semantic consistency. Second, *the Chain-of-Thought Reasoning Algorithm based on MCTS* explores possible SQL construction trajectories, where the search tree is expanded through predefined actions, and candidate paths are simulated and rewarded according to execution results and consistency checks. Finally, *Trajectory Selection with Mutual Reasoning Consistency Algorithm* re-evaluates these candidates with a secondary model and scores them based on execution accuracy, discriminator confidence, and mutual consistency, selecting the most reliable SQL trajectory. This pipeline integrates information understanding, search-based reasoning, and consistency verification, which is particularly suited for datasets that require complex numeric or physical computations, such as LogicCat or Archer.

Category	Parameter	Value
MCTS	reasoning_variant	full_iesr
MCTS	enable_sql_revision	true
MCTS	max_rollout_steps	16
MCTS	max_depth	16
MCTS	exploration_constant	1.414
MCTS	random_seed	42
MCTS	n_processes	10
LLM decoding	n	3
LLM decoding	top_p	0.8
LLM decoding	max_tokens	1024
LLM decoding	temperature	0.8
LLM decoding	n_strategy	single

Table 11: MCTS and local LLM decoding configuration used in our experiments.

Setting	EX (%)	Valid SQL (%)
$\gamma > 0$ configs	$18.7 \sim 21.5$	$54.2 \sim 57.0$
$\gamma = 0$ configs	$18.8 \sim 21.0$	$54.0 \sim 57.2$
Full 21-tuple grid	$18.7 \sim 21.5$	$54.1 \sim 57.2$

Table 12: Sensitivity summary of trajectory-selection weights on 500 LogicCat development queries using Qwen2.5-Coder-7B-Instruct. Candidate trajectories are fixed across all settings.

D Experiments Hyperparameter

Table 11 reports the MCTS and local decoding parameters used in our experiments.

E Hyperparameter Sensitivity Analysis

We analyze the sensitivity of two manually specified components in IESR: the trajectory selection weights (α, β, γ) and the schema filtering threshold τ. These experiments are designed to examine whether the final performance depends on a narrow heuristic configuration.

Trajectory selection weights. We first study the weighting coefficients in the trajectory scoring function:

$$\text{Score}(t) = \alpha \cdot \text{Exec}(t) + \beta \cdot \text{DiscConf}(t) + \gamma \cdot \text{ConsVote}(t).$$

Here, $\text{Exec}(t)$ denotes the execution based signal, $\text{DiscConf}(t)$ denotes the discriminator confidence or syntactic validity proxy, and $\text{ConsVote}(t)$ denotes the trajectory consistency signal derived from peer agreement. The execution signal provides the main selection criterion, while the two

Algorithm 1 Information Understanding with Rule-Guided Verification

Require: Natural language question q, database schema $S = (T, C)$, candidate column set $c_q \subseteq C$, rule base $\mathcal{B} = \{R_j\}$, LLM M_1, thresholds $\delta_{\text{match}}, \tau$

Output: High-quality relation set $\mathcal{R}_{\text{high}}$

Inject (q, S) into prompt template
Use M_1 to generate preliminary results: $R = (i, r, E, R, N, U, P)$
$\mathcal{R}_{\text{init}} \leftarrow \text{extract}(R)$
$\mathcal{R}_{\text{cand}} \leftarrow \emptyset$
for each $r_i \in \mathcal{R}_{\text{init}}$ **do**
  for each $R_j \in \mathcal{B}$ **do**
    if $\text{sim}(r_i, \text{Uni}_j, \text{Equt}_j) > \delta_{\text{match}}$ **then**
      $\mathcal{R}_{\text{cand}} \leftarrow \mathcal{R}_{\text{cand}} \cup \{r_i\}$
      break
    end if
  end for
end for
$\mathcal{R}_{\text{high}} \leftarrow \emptyset$
for each $r_i \in \mathcal{R}_{\text{cand}}$ **do**
  $P_i \leftarrow \text{Plan}(r_i, \mathcal{B})$
  $s_i \leftarrow \text{Executor}(r_i, P_i)$
  if $s_i > \tau$ **then**
    $\mathcal{R}_{\text{high}} \leftarrow \mathcal{R}_{\text{high}} \cup \{r_i\}$
  end if
end for
return $\mathcal{R}_{\text{high}}$

consistency signals serve as auxiliary evidence for suppressing unstable reasoning trajectories.

To isolate the effect of these coefficients, we perform offline reranking over saved candidate trajectories produced by the full IESR pipeline. This setting keeps the candidate SQL set fixed and changes only the final scoring weights, introducing no additional LLM calls. Specifically, we use 500 LogicCat development queries with saved rollouts from Qwen2.5-Coder-7B-Instruct and exhaustively sweep all tuples satisfying $\alpha, \beta, \gamma \geq 0$ and $\alpha + \beta + \gamma = 1$ with step size 0.2, resulting in 21 configurations.

Table 12 summarizes the sweep results. The valid SQL rate stays within a narrow range across different weights, suggesting that these coefficients mainly affect final candidate selection among generated trajectories rather than syntactic validity. The default configuration $(\alpha, \beta, \gamma) = (0.4, 0.2, 0.4)$ achieves the best EX in this grid. Configurations with $\gamma > 0$ are generally competitive, indicating that trajectory consistency provides useful auxiliary evidence. Meanwhile, using only one signal, such as $(1.0, 0.0, 0.0)$ or $(0.0, 0.0, 1.0)$,

α	β	γ	Samples	EX (%)	Valid SQL (%)
0.0	0.0	1.0	500	19.3	~ 54.3
0.0	0.2	0.8	500	19.4	~ 55.8
0.0	0.4	0.6	500	19.6	~ 55.9
0.0	0.6	0.4	500	20.4	~ 55.7
0.0	0.8	0.2	500	19.0	~ 56.8
0.0	1.0	0.0	500	19.3	~ 54.9
0.2	0.0	0.8	500	21.1	~ 54.2
0.2	0.2	0.6	500	18.7	~ 55.5
0.2	0.4	0.4	500	21.2	~ 56.8
0.2	0.6	0.2	500	20.1	~ 56.8
0.2	0.8	0.0	500	20.2	~ 56.9
0.4	0.0	0.6	500	19.8	~ 54.9
0.4	0.2	0.4	500	**21.5**	~ 55.2
0.4	0.4	0.2	500	21.4	~ 57.0
0.4	0.6	0.0	500	21.0	~ 56.8
0.6	0.0	0.4	500	20.8	~ 54.3
0.6	0.2	0.2	500	21.3	~ 54.8
0.6	0.4	0.0	500	19.1	~ 57.2
0.8	0.0	0.2	500	18.7	~ 55.3
0.8	0.2	0.0	500	19.0	~ 56.1
1.0	0.0	0.0	500	18.8	~ 54.0

Table 13: Full grid over trajectory selection weights with step size 0.2. All configurations satisfy $\alpha + \beta + \gamma = 1$.

Algorithm 2 MCTS-based CoT Reasoning for SQL Generation

Require: Query x, target LLM M, action set $\mathcal{A} = \{\mathrm{A1}, \ldots, \mathrm{A6}\}$, rollout budget N_{rollout}, exploration constant c, sampling temperature T_{samp}, consistency vote size K, database D

Output: Candidate trajectory set $\mathcal{T}_{\text{cand}}$ with terminal rewards

Initialize search tree $\mathcal{T}$ with root node v_0 representing x
Initialize visit counts $N(v, a)$ and cumulative values $Q(v, a)$ for all (v, a); $\mathcal{T}_{\text{cand}} \leftarrow \emptyset$
for $m = 1$ to N_{rollout} **do**
    $v \leftarrow v_0$; *path* $\leftarrow [\,]$ ▷ Selection
    while not terminal(v) **and** v is fully expanded **do**
        $a^\star \leftarrow \arg\max_{a \in \mathcal{A}} \left(\frac{Q(v,a)}{\max(1, N(v,a))} + c\sqrt{\frac{\ln \max(1, N(v))}{\max(1, N(v,a))}} \right)$
        path.append$\big((v, a^\star)\big)$; $v \leftarrow$ STEP$(v, a^\star, M)$
    end while ▷ Expansion
    if not terminal(v) **then**
        Select an untried valid action a at v
        $u \leftarrow$ STEP(v, a, M); add edge $(v, a) \rightarrow u$; initialize $N(u, \cdot), Q(u, \cdot)$
        path.append$\big((v, a)\big)$; $v \leftarrow u$
    end if ▷ Simulation (rollout)
    $(\hat{t}, \textit{terminal}) \leftarrow$ SIMULATE$(v, M, \mathcal{A}, T_{\text{samp}})$
    $r_t \leftarrow 0$
    if *terminal* = **true then**
        $r_t \leftarrow$ EVALUATEREWARD$(\hat{t}, x, D, K)$
        $\mathcal{T}_{\text{cand}} \leftarrow \mathcal{T}_{\text{cand}} \cup \{(\hat{t}, r_t)\}$
    end if ▷ Backpropagation
    BACKPROPAGATE(*path*, r_t)
end for
return $\mathcal{T}_{\text{cand}}$
function STEP(v, a, M)
    Apply action a to extend the current reasoning state at v using model M
    return new node u with updated partial trajectory
end function
function SIMULATE$(v, M, \mathcal{A}, T_{\text{samp}})$
    while not terminal(v) **and** depth limit not reached **do**
        Sample a from a rollout policy (e.g., uniform or softmax over UCT scores with temperature T_{samp})
        $v \leftarrow$ STEP(v, a, M)
    end while
    return (final trajectory $\hat{t}$ from root$\rightarrow v$, terminal(v))
end function
function EVALUATEREWARD$(\hat{t}, x, D, K)$
    Extract terminal SQL $\hat{y}$ from $\hat{t}$
    if EXECUTE$(\hat{y}, D)$ fails **then**
        return 0
    end if
    Sample K additional terminal SQL queries $\{\hat{y}_j\}_{j=1}^{K}$ under the same terminal context
    Execute all; let $w \leftarrow \frac{1}{K} \sum_{j=1}^{K} \mathbf{1}\big[\text{EXECUTE}(\hat{y}, D) = \text{EXECUTE}(\hat{y}_j, D)\big]$
    return $r_t = w$
end function
function BACKPROPAGATE(*path*, r_t)
    for each (v, a) in *path* **do**
        $N(v, a) \leftarrow N(v, a) + 1$; $Q(v, a) \leftarrow Q(v, a) + r_t$
    end for
end function

Algorithm 3 Trajectory Selection with Mutual Reasoning Consistency

Require: Candidate trajectory set $\mathcal{T}$ from LLM_1, discriminator LLM_2, weights α, β, γ
Output: Final selected SQL trajectory t^*

```
T_cand ← ∅
for each trajectory t = x ⊕ s_1 ⊕ ··· ⊕ s_d ∈ T do
    Randomly choose i < d, mask s_i, ..., s_d
    Feed (x ⊕ s_1 ⊕ ··· ⊕ s_{i-1}) into LLM_2 to generate completion t̂
    if t̂ semantically consistent with t then
        T_cand ← T_cand ∪ {t}
    end if
end for
for each t ∈ T_cand do
    Compute Exec(t)
    Compute DiscConf(t)
    Compute ConsVote(t)
    Score(t) ← α · Exec(t) + β · DiscConf(t) + γ · ConsVote(t)
end for
t* ← arg max_{t∈T_cand} Score(t)
return t*
```

τ	Macro F1	Macro P	Macro R	Col. Recall
0.3	25.4	24.5	66.8	72.8
0.4	30.3	39.2	69.3	71.3
0.5	37.9	38.6	66.3	68.3
0.6	**51.5**	**51.3**	**74.8**	**76.3**
0.7	49.2	48.9	69.0	75.0

Table 14: Sensitivity analysis of the schema filtering threshold τ on 500 LogicCat development queries. Metrics are computed on retrieved triplets against gold SQL literals.

gives lower EX, showing that execution and consistency signals are complementary.

Schema filtering threshold. We further analyze the LSH MinHash threshold τ used in schema filtering. This threshold controls the precision recall tradeoff of retrieved (table, column, value) triplets before downstream reasoning. We sweep τ on the same 500 LogicCat development queries using the Stage 1 pipeline with MySQL execution, local LLM keyword extraction, and local embedding filters. We report macro averaged precision, recall, and F1 against gold SQL literals, together with macro column recall over gold columns. Since 56 examples contain no parser extracted literal in the gold SQL, literal level F1 can be noisy; column recall is therefore included as a more stable schema grounding indicator.

The best macro F1 is obtained at $\tau = 0.6$, which is used as the default schema filtering threshold in all main experiments. Overall, the sensitivity analysis shows that IESR benefits from calibrated execution and consistency signals, while its performance does not rely on a single isolated weight choice. The schema filtering threshold mainly controls the expected precision recall tradeoff before MCTS reasoning, and the selected default corresponds to the best development F1 among the tested values.

F Dataset Details

- **LogicCAT (Liu et al., 2025a):** A challenging Text-to-SQL benchmark with 4,038 English questions covering multi-domain reasoning including physical knowledge, mathematical logic, common sense reasoning, and hypothetical scenarios.

- **Archer (Zheng et al., 2024):** It contains 1,042 questions from 20 databases that focus on complex, logical, hypothetical reasoning.

- **BIRD (Wretblad et al., 2024):** BIRD contains more than 12,751 unique questions and 95 databases. We used the development set, which includes 1,534 questions.

- **Spider (Yu et al., 2019):** Spider contains 10,181 unique questions in 166 databases. We use its development set of 1,034 questions.

G Details of Public Method

- **DIN-SQL:** DIN-SQL is a Text-to-SQL framework that splits query generation into steps (schema linking, decomposition, intermediate representation, correction) and reaches SOTA results without fine-tuning.
- **DAIL-SQL:** DAIL-SQL is a prompt-engineering method optimizing how examples and question representations are used in a few shots based on LLM Text-to-SQL.
- **CHESS:** CHESS is a multi-agent LLM framework for text-to-SQL, combining an information retriever, schema selector, candidate generator, and unit tester to prune large schemas, generate and refine SQL, and validate outputs. Reduce token usage and model calls while improving accuracy, achieving strong results on large-scale benchmarks such as BIRD.
- **LinkAlign:** It is a scalable schema linking framework for multi-database Text-to-SQL, combining semantic retrieval and schema grounding to accurately select relevant databases and columns in large, complex settings.
- **DTS-SQL:** DTS-SQL is a two-stage fine-tuning method that splits the Text-to-SQL task into schema linking and SQL generation subtasks, allowing smaller open source LLMs to approach the performance of larger models.
- **Alpha-SQL:** Alpha-SQL is a zero-shot Text-to-SQL framework that uses Monte Carlo Tree Search (MCTS) plus dynamic action proposals from an LLM to iteratively build SQL, with a self-supervised reward function to guide the search, achieving strong results.
- **SQL-O1:** SQL-o1 introduces a self-reward heuristic dynamic search method for Text-to-SQL. It leverages execution-aware feedback to guide iterative refinement of candidate SQL queries, enabling effective search without external annotators or learned critics.

H Prompt

In this section, we summarize the prompt templates used in our modular agent, organized as one front-end understanding prompt and six action prompts that constitute the heterogeneous reasoning space. The Information Understanding prompt constructs a structured semantic state from the user question by extracting intent, entities and relations, numerical values and units, and the schema elements that are likely required (Figure 17). Based on this intermediate representation, the agent performs reasoning through six specialized actions. The Equation Explain prompt makes implicit formulas and numerical transformations explicit, supporting unit conversion and arithmetic constraints before SQL construction (Figure 18). The Schema Selection prompt narrows the search space by selecting task-relevant tables and columns from the database schema (Figure 19). To further resolve ambiguous fields, the Identify Column Information prompt disambiguates candidate columns using semantic cues and column-level descriptions/examples (Figure 20). The Entity Extraction prompt grounds query entities and their associated relations to concrete values or schema mentions, providing reliable constraints for WHERE/JOIN conditions (Figure 16). Conditioned on the above structured signals, the SQL Generation prompt synthesizes an executable SQL query, including necessary joins, aggregations, and nested structures (Figure 21). Finally, the SQL Revision prompt corrects the draft SQL by diagnosing logical, join, and schema-mismatch errors (optionally using execution feedback when available), and outputs a repaired query (Figure 22)

Entity Extraction Prompt

Objective: Analyze the given question and hint to identify and extract keywords, keyphrases, and named entities. These elements are crucial for understanding the core components of the inquiry and the guidance provided. This process involves recognizing and isolating significant terms and phrases that could be instrumental in formulating searches or queries related to the posed question.

Instructions: Read the Question Carefully: Understand the primary focus and specific details of the question. Look for any named entities (such as organizations, locations, etc.), technical terms, and other phrases that encapsulate important aspects of the inquiry.
Analyze the Hint: The hint is designed to direct attention toward certain elements relevant to answering the question. Extract any keywords, phrases, or named entities that could provide further clarity or direction in formulating an answer.
List Keyphrases and Entities: Combine your findings from both the question and the hint into a single Python list. This list should contain:
Keywords: Single words that capture essential aspects of the question or hint.
Keyphrases: Short phrases or named entities that represent specific concepts, locations, organizations, or other significant details.
Ensure to maintain the original phrasing or terminology used in the question and hint.

Example 1:
Question: "What is the annual revenue of Acme Corp in the United States for 2022?"
Hint: "Focus on financial reports and U.S. market performance for the fiscal year 2022."
```python
["annual revenue", "Acme Corp", "United States", "2022", "financial reports", "U.S. market performance", "fiscal year"]
```
Example 2:
Question: "In the Winter and Summer Olympics of 1988, which game has the most number of competitors? Find the difference of the number of competitors between the two games."
Hint: "the most number of competitors refer to MAX(COUNT(person_id)); SUBTRACT(COUNT(person_id where games_name = '1988 Summer'), COUNT(person_id where games_name = '1988 Winter'));"
```python
["Winter Olympics", "Summer Olympics", "1988", "1988 Summer", "Summer", "1988 Winter", "Winter", "number of competitors", "difference", "MAX(COUNT(person_id))", "games_name", "person_id"]
```
Example 3:
Question: "How many Men's 200 Metres Freestyle events did Ian James Thorpe compete in?"
Hint: "Men's 200 Metres Freestyle events refer to event_name = 'Swimming Men"s 200 metres Freestyle'; events compete in refers to event_id;"
```python
["Swimming Men's 200 metres Freestyle", "Ian James Thorpe", "Ian", "James", "Thorpe", "compete in", "event_name", "event_id"]
```
Task:
Given the following question and hint, identify and list all relevant keywords, keyphrases, and named entities.

Question: {QUESTION}
Hint: {HINT}

Please provide your findings as a Python list within ```python ``` tags, capturing the essence of both the question and hint through the identified terms and phrases. Only output the Python list, no explanations needed.

Figure 16: An Example of Entity Extraction Prompt.

Information Extraction Prompt

You are an expert information extraction assistant for Text-to-SQL tasks. Your task is to analyze the given question and database schema, then extract structured information that will help generate accurate SQL queries.
You need to extract the following information from the question and schema:
1. **Entities**: Key entities mentioned in the question (e.g., objects, concepts, attributes)
2. **Relations**: Relationships between entities or between entities and database elements
3. **Intent**: The main intent or goal of the question
4. **Reasoning Type**: The type of reasoning required (e.g., "mathematical logic reasoning", "physical knowledge reasoning", "common sense understanding reasoning", "ideal hypothesis reasoning")
5. **Numerical Values**: Any numerical values mentioned in the question
6. **Units**: Units of measurement or conversion information
7. **Foreign Keys**: Foreign key relationships relevant to the question
8. **Primary Keys**: Primary key information for relevant tables
9. **Table Info**: Detailed information about relevant tables and columns
****
Example:
Database Schema:
CREATE TABLE aircraft
(
  `Aircraft_ID` INTEGER NOT NULL,
  `Aircraft` VARCHAR(50), -- Column Description: 飞机的名称或型号（例如：Robinson R-22）
  `Total_Disk_Area` FLOAT, -- Column Description: 飞机的总旋翼盘面积（例如：497 ft² (46.2 m²)）
  `Max_Disk_Loading` FLOAT, -- Column Description: 飞机的最大旋翼盘载荷（例如：2.6 lb/ft² (14 kg/m²)）
  `Max_Gross_Weight` FLOAT, -- Column Description: 飞机的最大总重量（例如：1,370 lb (635 kg)）
  PRIMARY KEY (`Aircraft_ID`)
);
Question:
If a Robinson R-22 helicopter has a total rotor disc area of 46.2 square meters and a maximum rotor disc loading of 14 kilograms per square meter, what is its lift under maximum gross weight?
Hint:
Robinson R-22 refers to Aircraft = 'Robinson R-22'; lift refers to Max_Gross_Weight = Total_Disk_Area × Max_Disk_Loading
Extracted Information:
```python
{
  'entities': ['Robinson R-22 helicopter', 'total rotor disc area 46.2 m²', 'maximum rotor disc loading 14 kg/m²'],
  'relations': ['helicopter_to_specs (Robinson R-22 → aircraft specifications)', 'lift_calculation (Max_Gross_Weight = Total_Disk_Area × Max_Disk_Loading)'],
  'intent': 'calculate_lift_under_max_gross_weight',
  'reasoning_type': 'physical knowledge reasoning',
  'numerical_values': [46.2, 14],
  'units': {
    'area': 'm² (square meters)',
    'loading': 'kg/m² (kilograms per square meter)',
    'lift': 'kg (kilograms, derived from 46.2 m² × 14 kg/m²)',
    'conversion': 'Lift = Total_Disk_Area × Max_Disk_Loading'
  },
  'foreign_keys': 'This issue does not provide foreign keys',
  'primary_keys': [{'pk_table': 'aircraft', 'pk_column': 'Aircraft_ID'}],
  'tableinfo': {
    'AirCraft.aircraft': [
      {'column': 'Aircraft', 'type': 'VARCHAR(50)', 'primary_key': False, 'comment': '飞机的名称或型号（例如：Robinson R-22）', 'examples': ['Robinson R-22', 'Bell 206B3 JetRanger']},
      {'column': 'Total_Disk_Area', 'type': 'FLOAT', 'primary_key': False, 'comment': '飞机的总旋翼盘面积（例如：497 ft² (46.2 m²)）', 'examples': ['46.2', '81.1']},
      {'column': 'Max_Disk_Loading', 'type': 'FLOAT', 'primary_key': False, 'comment': '飞机的最大旋翼盘载荷（例如：2.6 lb/ft² (14 kg/m²)）', 'examples': ['14.0', '18.0']},
      {'column': 'Max_Gross_Weight', 'type': 'FLOAT', 'primary_key': False, 'comment': '飞机的最大总重量（例如：1,370 lb (635 kg)）', 'examples': ['635.0', '1451.0']}
    ]
  }
}
```
****
Now, extract the information for the real question. You must output a valid Python dictionary in the exact format shown above.
Important guidelines:
- **Entities**: Extract all key entities mentioned in the question, including objects, attributes, and concepts
- **Relations**: Identify relationships between entities, or mappings from entities to database elements
- **Intent**: Summarize the main goal in a concise phrase (use snake_case)
- **Reasoning Type**: Identify the reasoning type(s) required (common types: "mathematical logic reasoning", "physical knowledge reasoning", "common sense understanding reasoning", "ideal hypothesis reasoning")
- **Numerical Values**: Extract all numerical values mentioned in the question as a list
- **Units**: Extract units of measurement and any conversion formulas
- **Foreign Keys**: Extract foreign key relationships as a list of dictionaries with 'fk_table', 'fk_column', 'ref_table', 'ref_column', or use the string "This issue does not provide foreign keys" if no foreign keys are relevant
- **Primary Keys**: Extract primary key information for relevant tables as a list of dictionaries with 'pk_table' and 'pk_column'
- **Table Info**: Extract detailed information about relevant tables and columns, including column names, types, primary_key status, comments, and examples

Database Schema:{SCHEMA_CONTEXT}

Question:{QUESTION}

Hint:{HINT}

Please output only the Python dictionary in the exact format shown in the example, without any additional explanation or markdown formatting. The output should be a valid Python dictionary that can be evaluated with `ast.literal_eval()`.

Figure 17: Understanding Prompt including Intent Recognition, Unit Understanding, Relation and Entity Extracting, and Pseudo-Schema Understanding

Equation Expaination Prompt

You are an AI assistant to help me identify and explain mathematical formulas or calculations mentioned in the question. Your task is to recognize any formulas, equations, or calculation expressions in the question and convert them into SQL-compatible expressions that can be used in the SQL query.
****
Example 1:
Database Schema:
CREATE TABLE aircraft
(
    `aircraft_id` INTEGER NOT NULL,
    `max_gross_weight` REAL, -- Column Description: maximum gross weight in pounds
    `rotor_diameter` REAL, -- Column Description: rotor diameter in feet
    PRIMARY KEY (`aircraft_id`)
);
Question:
Calculate what is the rotor disk loading of the Robinson R-22 helicopter at its maximum gross weight?
Hint:
rotor disk loading refers to max_gross_weight / (π * (rotor_diameter / 2)^2); Robinson R-22 refers to aircraft_name = 'Robinson R-22'
Answer:
The question asks for "rotor disk loading", which is a calculated value. The formula is: rotor disk loading = max_gross_weight / (π * (rotor_diameter / 2)^2). In SQL, this can be expressed as: `max_gross_weight / (3.14159 * POWER(rotor_diameter / 2, 2))` or `max_gross_weight / (3.14159 * (rotor_diameter / 2) * (rotor_diameter / 2))`.
****
Example 2:
Database Schema:
CREATE TABLE products
(
    `product_id` INTEGER NOT NULL,
    `price` REAL, -- Column Description: product price in dollars
    `discount_rate` REAL, -- Column Description: discount rate as percentage (0-100)
    `quantity` INTEGER, -- Column Description: quantity purchased
    PRIMARY KEY (`product_id`)
);
Question:
What is the total cost after applying a 10% discount for products with quantity greater than 5?
Hint:
total cost after discount refers to (price * quantity) * (1 - discount_rate / 100); quantity greater than 5 refers to quantity > 5
Answer:
The question involves a calculation: total cost after discount = (price * quantity) * (1 - discount_rate / 100). In SQL, this can be expressed as: `(price * quantity) * (1 - discount_rate / 100.0)`. Additionally, for the 10% discount case, it can be written as: `(price * quantity) * 0.9`.
****
Example 3:
Database Schema:
CREATE TABLE employees
(
    `employee_id` INTEGER NOT NULL,
    `salary` REAL, -- Column Description: annual salary in dollars
    `bonus_percentage` REAL, -- Column Description: bonus percentage (0-100)
    `years_of_service` INTEGER, -- Column Description: years of service
    PRIMARY KEY (`employee_id`)
);
Question:
Calculate the average compensation (salary + bonus) for employees with more than 10 years of service, where bonus = salary * bonus_percentage / 100.
Hint:
compensation refers to salary + (salary * bonus_percentage / 100); more than 10 years refers to years_of_service > 10
Answer:
The question requires calculating "compensation" which is defined as salary + bonus, where bonus = salary * bonus_percentage / 100. The formula is: compensation = salary + (salary * bonus_percentage / 100). In SQL, this can be expressed as: `salary + (salary * bonus_percentage / 100.0)` or simplified as: `salary * (1 + bonus_percentage / 100.0)`.
****
Now, answer the real question, and you need to follow the answer style of the above examples (identify the formula/calculation and explain how to express it in SQL, answer in 2-3 sentences).

Database Schema:
{SCHEMA_CONTEXT}

Question:
{QUESTION}

Hint:
{HINT}

Answer:

Figure 18: An Example of Equation Explain Prompt.

Schema Selection Prompt

You are an expert and very smart data analyst.
Your task is to examine the provided database schema, understand the posed question, and use the hint to pinpoint the specific columns within tables that are essential for crafting a SQL query to answer the question.

The schema offers an in-depth description of the database's architecture, detailing tables, columns, primary keys, foreign keys, and any pertinent information regarding relationships or constraints. Special attention should be given to the examples listed beside each column, as they directly hint at which columns are relevant to our query.

For key phrases mentioned in the question, we have provided the most similar values within the columns denoted by "-- Value Examples" in front of the corresponding column names. This is a critical hint to identify the columns that will be used in the SQL query.

The hint aims to direct your focus towards the specific elements of the database schema that are crucial for answering the question effectively.

Task:
Based on the database schema, question, and hint provided, your task is to identify all and only the columns that are essential for crafting a SQL query to answer the question.
For each of the selected columns, explain why exactly it is necessary for answering the question. Your reasoning should be concise and clear, demonstrating a logical connection between the columns and the question asked.

Please respond with your reasoning process and a JSON object structured as follows:
Reasoning Process:
(Here, output your reasoning for selecting the columns, be concise and clear)
Final JSON Result:
```json
{{
  "table_name1": ["column1", "column2", ...],
  "table_name2": ["column1", "column2", ...]
  ...
}}
```

Make sure your response includes the table names as keys, each associated with a list of column names that are necessary for writing a SQL query to answer the question.
For each aspect of the question, provide a clear and concise explanation of your reasoning behind selecting the columns.
Take a deep breath and think logically. If you do the task correctly, I will give you 1 million dollars.

Database Schema Overview:
{SCHEMA_CONTEXT}

Question:
{QUESTION}

Hint:
{HINT}

Please output your reasoning process first after "# Reasoning Process", and output a JSON object within ```json``` tags.

Figure 19: An Example of Schema Selection Prompt.

Indentify Column Information Prompt

You are an AI assistant to help me identify the potential column values and column functions (if needed to be used in the SQL query) that are essential for answering the question.

Your task is to:
1. Identify the specific column values that should be used in WHERE clauses or other filtering conditions
2. Identify any column functions (like date functions, string functions, etc.) that might be needed for the SQL query
****
Example 1: Identifying Column Values
Database Schema:
```
CREATE TABLE generalinfo
(
        id_restaurant INTEGER not null primary key,
        food_type TEXT null, -- Value Examples: `thai`| Column Description: the food type
        city TEXT null, -- Column Description: the city where the restaurant is located in
);
CREATE TABLE location
(
        id_restaurant INTEGER not null primary key,
        street_name TEXT null, -- Value Examples: `ave`, `san pablo ave`, `pablo ave`| Column Description: the street name of the restaurant
        city TEXT null, -- Column Description: the city where the restaurant is located in
        foreign key (id_restaurant) references generalinfo (id_restaurant) on update cascade on delete cascade,
);
```
Question:
How many Thai restaurants can be found in San Pablo Ave, Albany?
Hint:
Thai restaurant refers to food_type = 'thai'; San Pablo Ave Albany refers to street_name = 'san pablo ave' AND T1.city = 'albany'
Answer:
Since the restaurants are located in Albany, based on the schema information and the hint, I need to use `location`.`street_name` = 'san pablo ave' AND `generalinfo`.`city` = 'albany'.
****
Example 2: Identifying Column Functions
Database Schema:
```
CREATE TABLE businesses
(
   `business_id` INTEGER NOT NULL,
   `name` TEXT NOT NULL, -- Column Description: the name of the eatery
   PRIMARY KEY (`business_id`),
);
CREATE TABLE inspections
(
   `business_id` INTEGER NOT NULL, -- Column Description: the unique id of the business
   `score` INTEGER DEFAULT NULL, -- Column Description: the inspection score
   `date` TEXT NOT NULL, -- Value Examples: `2014-01-24`
   FOREIGN KEY (`business_id`) REFERENCES `businesses` (`business_id`),
);
```
Question:
What are the names of the businesses that passed with conditions in May 2012?
Hint:
name of business refers to dba_name; passed with conditions refers to results = 'Pass w/ Conditions'; in May 2012 refers to inspection_date like '2012-05%'
Answer:
Since the businesses passed with conditions in May 2012, I should consider a date-related function to filter the `inspections`.`date` column. I find that column is of type TEXT, so I can use the strftime('%Y-%m', `inspections`.`date`) = '2012-05' to filter the date.
****
Example 3: Identifying Both Column Values and Functions
Database Schema:
```
CREATE TABLE products
(
   `product_id` INTEGER NOT NULL,
   `price` REAL, -- Column Description: product price in dollars
   `discount_rate` REAL, -- Column Description: discount rate as percentage (0-100)
   `quantity` INTEGER, -- Column Description: quantity purchased
   `purchase_date` TEXT, -- Value Examples: `2023-05-15` | Column Description: purchase date
   PRIMARY KEY (`product_id`)
);
```
Question:
What is the total cost after applying a 10% discount for products with quantity greater than 5 purchased in May 2023?
Hint:
total cost after discount refers to (price * quantity) * (1 - discount_rate / 100); quantity greater than 5 refers to quantity > 5; May 2023 refers to purchase_date like '2023-05%'
Answer:
For column values, I need to filter products where quantity > 5. For column functions, I need to use strftime('%Y-%m', `products`.`purchase_date`) = '2023-05' to filter the date to May 2023. Additionally, I need to calculate the discounted cost using the formula (price * quantity) * (1 - discount_rate / 100.0).
****
Now, answer the real question, and you need to follow the answer style of the above examples. Your answer should cover both column values (if any) and column functions (if any) that are needed for the SQL query. Answer in 2-3 sentences.

Database Schema:{SCHEMA_CONTEXT}

Question:{QUESTION}

Hint:{HINT}

Answer: {xxx}

Figure 20: An Example of Identify Column Information Prompt.

SQL Generation Prompt

You are an experienced database expert. Now you need to generate a SQL query given the database information, a question and some additional information. The database structure is defined by the following table schemas (comments after '--' provide additional column descriptions). Note that the "Value Examples" are actual values from the column. Some column might contain the values that are directly related to the question. Use it to help you justify which columns to use.

Given the table schema information description and the `Question`. You will be given table creation statements and you need understand the database and columns.

You will be using a way called "recursive divide-and-conquer approach to SQL query generation from natural language".

Here is a high level description of the steps.
1. **Divide (Decompose Sub-question with Pseudo SQL):** The complex natural language question is recursively broken down into simpler sub-questions. Each sub-question targets a specific piece of information or logic required for the final SQL query.
2. **Conquer (Real SQL for sub-questions):** For each sub-question (and the main question initially), a "pseudo-SQL" fragment is formulated. This pseudo-SQL represents the intended SQL logic but might have placeholders for answers to the decomposed sub-questions.
3. **Combine (Reassemble):** Once all sub-questions are resolved and their corresponding SQL fragments are generated, the process reverses. The SQL fragments are recursively combined by replacing the placeholders in the pseudo-SQL with the actual generated SQL from the lower levels.
4. **Final Output:** This bottom-up assembly culminates in the complete and correct SQL query that answers the original complex question.

Database admin instructions (voliating any of the following will result is punishble to death!):
1. **SELECT Clause:**
   - Only select columns mentioned in the user's question.
   - Avoid unnecessary columns or values.
2. **Aggregation (MAX/MIN):**
   - Always perform JOINs before using MAX() or MIN().
3. **ORDER BY with Distinct Values:**
   - Use `GROUP BY <column>` before `ORDER BY <column> ASC|DESC` to ensure distinct values.
4. **Handling NULLs:**
   - If a column may contain NULL values, use `JOIN` or `WHERE <column> IS NOT NULL`.
5. **FROM/JOIN Clauses:**
   - Only include tables essential to answer the question.
6. **Strictly Follow Hints:**
   - Adhere to all provided hints.
7. **Thorough Question Analysis:**
   - Address all conditions mentioned in the question.
8. **DISTINCT Keyword:**
   - Use `SELECT DISTINCT` when the question requires unique values (e.g., IDs, URLs).
9. **Column Selection:**
   - Carefully analyze column descriptions and hints to choose the correct column when similar columns exist across tables.
10. **String Concatenation:**
   - Never use `|| ' ' ||` or any other method to concatenate strings in the `SELECT` clause.
11. **SQLite Functions Only:**
   - Use only functions available in SQLite.
12. **Date Processing:**
   - Utilize `STRFTIME()` for date manipulation (e.g., `STRFTIME('%Y', SOMETIME)` to extract the year).
13. **Schema Syntax:**
   - Use `table_name`.`column_name` to refer to columns from different tables, when table name or column name contains whitespace.
14. **JOIN Preference:**
   - Prioritize `INNER JOIN` over nested `SELECT` statements. Do not use `CROSS JOIN` or `LEFT/RIGHT JOIN`.

Repeating the question and hint, and generating the SQL with Recursive Divide-and-Conquer, and finally try to simplify the SQL query using `INNER JOIN` over nested `SELECT` statements IF POSSIBLE.

Please respond with xml tags structured as follows:
```xml
<think>
Your detailed reasoning for the SQL query generation, with Recursive Divide-and-Conquer approach.
</think>
<sql>
The final SQL query that answers the question.
</sql>
```
【Table creation statements】
{SCHEMA_CONTEXT}

【Question】
{QUESTION}

【HINT】
{HINT}

Only output xml format (starting with ```xml and ending with ```) as your response.

Figure 21: Understanding Prompt including Intent Recognition, Unit Understanding, Relation and Entity Extracting, and Pseudo-Schema Understanding.

SQL Revision Prompt

Task Description:
You are an SQL database expert tasked with correcting a SQL query. A previous attempt to run a query did not yield the correct results, either due to errors in execution or because the result returned was empty or unexpected. Your role is to analyze the error based on the provided database schema and the details of the failed execution, and then provide a corrected version of the SQL query.

Procedure:
1. Review Database Schema:
    - Examine the table creation statements to understand the database structure.
2. Analyze Query Requirements:
    - Original Question: Consider what information the query is supposed to retrieve.
    - Hint: Use the provided hints to understand the relationships and conditions relevant to the query.
    - Executed SQL Query: Review the SQL query that was previously executed and led to an error or incorrect result.
    - Execution Result: Analyze the outcome of the executed query to identify why it failed (e.g., syntax errors, incorrect column references, logical mistakes), note that the execution result is possibly empty for some queries.
3. Correct the Query:
    - Modify the SQL query to address the identified issues, ensuring it correctly fetches the requested data according to the database schema and query requirements.

Based on the question, table schemas, the previous query, and the execution result, analyze the result following the procedure, and try to fix the query.
You cannot modify the database schema or the question, just output the corrected query.

Please respond with xml tags structured as follows:
```xml
<think>
Your detailed reasoning for the SQL query revision.
</think>
<sql>
The final SQL query that answers the question.
</sql>
```

****
【Table creation statements】
{SCHEMA_CONTEXT}

【Question】
{QUESTION}

【HINT】
{HINT}
****
Only output xml format (starting with ```xml and ending with ```) as your response.

Figure 22: An Example of Revising SQL Reasoning Prompt.