DIAGSQL: A Diagnostic Validator for Text-to-SQL with Reward Allocation and Co-occurrence Shaping

Weibin Liao[♣*‡], **Bowen Cao**[♠*], **Dong Fang**[♢†], **Wai Lam**[♠]
[♣]Peking University
[♠]The Chinese University of Hong Kong
[♢]LIGHTSPEED STUDIOS
liaoweibin@stu.pku.edu.cn
https://github.com/MrBlankness/DiagSQL.git

Abstract

While large language models have significantly advanced Text-to-SQL generation, it remains largely an **open-loop** paradigm due to the absence of a reliable validator capable of systematically diagnosing generated queries. Training a dedicated diagnostic validator to provide targeted, reflective feedback is crucial for closing this loop. However, employing standard Reinforcement Learning paradigms, particularly Group Relative Policy Optimization (GRPO), reveals fundamental structural misalignments. Through exploratory analysis, we identify a critical limitation of standard GRPO in this context: a severe *credit assignment failure*, where uniform sequence-level rewards inadvertently reinforce hallucinated reasoning alongside correct sub-answers in complex structured outputs. Concurrently, we discover profound *structural dependencies* among SQL errors, revealing inherent co-occurrence semantics dictating that certain errors frequently appear together. To address the limitation and leverage this insight, we propose **DIAGSQL**, a novel framework that utilizes an improved GRPO algorithm tailored for diagnostic Text-to-SQL validation. DIAGSQL introduces `Fine-grained Token-level Reward Allocation (FTRA)`, which parses structured responses to precisely distribute rewards at the token level, effectively isolating and penalizing erroneous reasoning. Furthermore, it incorporates `Co-occurrence-aware Reward Shaping (CORS)`, which leverages our discovered pre-computed error co-occurrence matrix to dynamically adjust optimization objectives, encouraging logical error combinations while penalizing structural violations. Extensive experiments demonstrate that DIAGSQL significantly enhances the robustness and accuracy of SQL error diagnosis, providing actionable, high-quality feedback that effectively transforms Text-to-SQL into a **closed-loop** paradigm.

1 Introduction

Text-to-SQL translates natural language questions into executable SQL queries, bridging the gap between non-expert users and complex relational databases [1, 2, 3]. Despite significant progress driven by large language models (LLMs) [4, 5, 6], Text-to-SQL generation remains an ***open-loop paradigm***, lacking a reliable **validator** to diagnose the correctness of generated queries [7, 8]. Current external validation paradigms largely fall short: (1) execution-based validators detect only syntax errors and remain blind to semantic inaccuracies [9]; (2) LLM-as-judge approaches are prohibitively expensive and highly dependent on the judge's inherent capabilities [9, 10, 11, 12]; and (3) self-correction by the SQL generator often exacerbates mistakes due to its limited capacity [13, 14, 15, 16].

∗ Equal contribution. † Corresponding author. ‡ This work was done during an internship at LIGHTSPEED STUDIOS.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Therefore, training a dedicated, **specialized SQL validator to transform Text-to-SQL into a *closed-loop paradigm* is of paramount importance.** This validator should go beyond binary correctness verification to conduct in-depth error type analysis (e.g., `Attribute Mismatch`, `Join Condition Mismatch`), thereby providing targeted, reflective feedback to guide the refinement of SQL generators.

To train such reasoning models, Reinforcement Learning (RL) [17, 18, 19], particularly Group Relative Policy Optimization (GRPO) [18], has emerged as the common practice. However, diagnostic SQL validation presents a unique challenge: the target output is a *structured, multi-choice reasoning sequence* consisting of binary correctness labels, specific error types, and detailed reasoning paths. This structural complexity introduces a fundamental misalignment when optimized with standard GRPO. Specifically, standard GRPO distributes a sequence-level scalar reward uniformly across all generated tokens [20]. In a complex diagnostic response, a model might correctly identify the overall error label but hallucinate specific error types along with flawed reasoning. A sequence-averaged reward mechanism inadvertently reinforces these hallucinated reasoning tokens alongside the correct ones, resulting in a severe *credit assignment failure* [21, 22]. Concurrently, we observe that diverse SQL errors are not isolated events; they possess inherent *structural dependencies* dictated by SQL syntax [8], meaning certain errors inherently co-occur (e.g., omitting a table inevitably corrupts subsequent join conditions). To effectively train a SQL validator, we must address these unique challenges: *we need to optimize token-level credit assignment to prevent rewarding hallucinations, and explicitly incorporate co-occurrence-aware optimization to leverage structural SQL priors.*

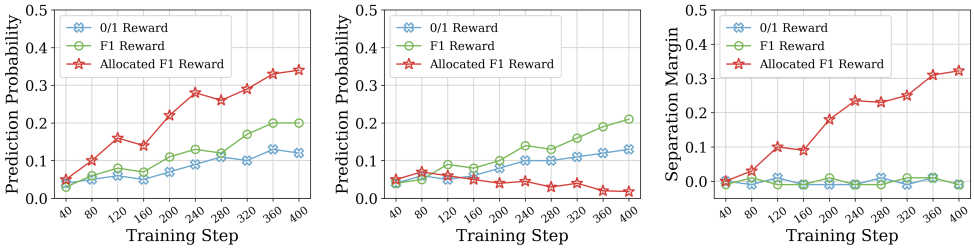
Motivated by these challenges, we conduct an exploratory analysis on fine-tuned LLMs with standard GRPO. As illustrated in Figure 1, standard uniform rewards (e.g., strict 0/1 or F1 reward) [23] struggle to differentiate *True Positive probability* (p_{TP}) from *False Positive probability* (p_{FP}), causing the probability of hallucinated errors to rise simultaneously with correct predictions. Meanwhile, the strong correlations observed in the error co-occurrence matrix in Figure 2 confirm that specific error combinations provide critical structural priors that remain unexploited by standard RL objectives.

To address these fundamental challenges, we propose **DIAGSQL**, a novel approach utilizing an improved GRPO algorithm to train a robust diagnostic SQL validator. Our algorithm introduces two core mechanisms: `Fine-grained Token-level Reward Allocation (FTRA)` and `Co-occurrence-aware Reward Shaping (CORS)`. Specifically, `FTRA` resolves the credit assignment failure by parsing the structured JSON response and independently evaluating each predicted error. It allocates correctness and F1-based rewards strictly to the tokens constituting the valid sub-answers, ensuring that hallucinated error types and their flawed reasoning parts—lacking valid grounding—receive zero or negative rewards. Furthermore, `CORS` explicitly introduces structural inductive biases. Using a pre-computed error co-occurrence matrix, `CORS` dynamically shapes the reward by providing bonuses for discovering concomitant errors, penalizing incomplete diagnoses of strongly correlated errors, and heavily penalizing illogical, syntax-violating error combinations.

Extensive experiments on standard benchmarks demonstrate that DIAGSQL remarkably improves both diagnostic precision and downstream execution accuracy. When deployed as a verifier to guide the refinement of upstream text-to-SQL generators, DIAGSQL enables small-scale open-weight models (e.g., 8B) to significantly outperform state-of-the-art proprietary LLMs.

Our contributions are summarized as follows:

1. **Identification of RL Misalignments in Structured Reasoning**: We expose a critical limitation of standard GRPO in training diagnostic validators—namely, a severe credit-assignment failure—and uncover the inherent error co-occurrence semantics within SQL diagnostics.
2. **Novel RL Framework (DIAGSQL)**: We propose an improved GRPO algorithm that features `FTRA` for precise, token-level reward distribution, and `CORS` for dynamic, structure-aware reward shaping.
3. **State-of-the-Art Closed-Loop Generation**: Extensive evaluations demonstrate that DIAGSQL provides highly actionable feedback, significantly enhancing SQL-generation robustness and enabling compact models to surpass much larger proprietary LLMs.



(a) p_{TP} (b) p_{FP} (c) $\Delta = p_{TP} - p_{FP}$

Figure 1: Performance comparison of different reward mechanisms (0/1 Reward, F1 Reward, and Allocated F1 Reward) over training steps. (a) p_{TP}: The prediction probability of the ground-truth error types (True Positives). (b) p_{FP}: The prediction probability of hallucinated error types (False Positives). (c) The separation margin ($\Delta = p_{TP} - p_{FP}$). The results demonstrate that our proposed Allocated F1 Reward effectively increases p_{TP} while suppressing p_{FP}, mitigating the credit assignment failure inherent in standard sequence-level rewards.

2 Background and Motivation

2.1 Problem Formulation and Standard GRPO

The task of diagnostic Text-to-SQL validation is formulated as a conditional generation problem within an RL paradigm. Given a natural language question X, a database schema S, and a generated SQL query Q (produced by an upstream SQL generator), we construct a prompt $P = (X, S, Q)$. The diagnostic validator, parameterized by a policy model π_θ, takes P as the state s and generates an action a, which is a structured response $Y = \langle y_{label}, E_{pred}, Z_{pred} \rangle$ comprising three semantic components: a binary correctness label $y_{label} \in \{\text{True}, \text{False}\}$ indicating whether Q is faithful to X under S; a set of K predicted specific error types $E_{pred} = \{e_1, \ldots, e_K\}$; and their corresponding detailed reasoning paths $Z_{pred} = \{z_1, \ldots, z_K\}$, where each z_k grounds e_k in the schema or query logic. The concrete JSON realization of Y used throughout this work is introduced in Section 3.1.

In standard GRPO [18, 24], the optimization relies on relative reward comparisons within a sampled group. For each prompt P, the model π_θ samples a group of G outputs $\mathcal{Y} = \{Y_1, \ldots, Y_G\}$. An evaluator computes a sequence-level scalar reward R_i for each $Y_i \in \mathcal{Y}$. The advantage A_i is computed by normalizing these rewards:

$$A_i = \frac{R_i - \mu_{\mathcal{R}}}{\sigma_{\mathcal{R}}} \tag{1}$$

where $\mu_{\mathcal{R}}$ and $\sigma_{\mathcal{R}}$ are the mean and standard deviation of the group's rewards. The standard GRPO objective maximizes:

$$\mathcal{J}_{GRPO}(\theta) = \mathbb{E}_{P,\mathcal{Y}\sim\pi_\theta}\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{N_i}\sum_{t=1}^{N_i}\min\left(\rho_{i,t}A_i, \text{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon)A_i\right) - \beta\mathbb{D}_{KL}(\pi_\theta||\pi_{ref})\right] \tag{2}$$

where N_i is the sequence length of Y_i, $\rho_{i,t} = \frac{\pi_\theta(y_{i,t}|s, y_{i,<t})}{\pi_{ref}(y_{i,t}|s, y_{i,<t})}$ is the probability ratio of the current policy π_θ to the reference policy π_{ref} for the t-th token $y_{i,t}$, ϵ is the clipping threshold, and β controls the KL divergence penalty.

We build on GRPO because it avoids training a same-size value model and therefore keeps the 8B validator lightweight, while matching the critic-free optimization family used by recent reasoning models. Our claim is not that GRPO dominates PPO [24]: PPO can obtain token-varying advantages from a learned critic and generalized advantage estimation. Rather, the pathology studied here is specific to critic-free, group-relative methods—including GRPO and variants such as DAPO [19]—that broadcast a trajectory-level scalar across tokens. Table 3 confirms that simply replacing vanilla GRPO with DAPO does not resolve this diagnostic credit-assignment failure; the task-structured attribution of `FTRA` is essential.

2.2 The Credit Assignment Failure of Standard GRPO

To investigate the efficacy of the standard GRPO objective (Eq. 2) for diagnostic validators, we fine-tune the QWEN3-8B [25] model using two baseline reward formulations: a strict sequence-level binary reward (1 if perfectly matching the ground truth, 0 otherwise) and a sequence-level F1 reward. The learning dynamics, illustrated in Figure 1, reveal a critical bottleneck in the optimization process. While the True Positive probability (p_{TP})—the likelihood of predicting a genuine ground-truth error—experiences sluggish growth, the False Positive probability (p_{FP})—the likelihood of hallucinating an ungrounded error—exhibits a continuous and alarming upward trend.

We attribute this pathological behavior to a fundamental structural flaw in how GRPO handles complex, multi-component outputs. As defined in Eq. 2, the advantage A_i is a single scalar broadcasted uniformly to every token $y_{i,t}$ in the generated sequence Y_i. However, a diagnostic response is inherently structured: it may contain a correct overall correctness label (y_{label}) alongside completely hallucinated specific error types (E_{pred}) and flawed reasoning paths (Z_{pred}). In such partially correct sequences, the sequence-level evaluation assigns a moderate, positive advantage A_i to the entire response. Consequently, the specific tokens responsible for the correct sub-answers do not receive the isolated, strong positive reinforcement they require. Even more detrimentally, the tokens constituting the hallucinated errors and logically flawed reasoning paths are improperly rewarded and reinforced simply because they belong to a sequence that achieved a positive overall score.

Observation I: *Standard GRPO's sequence-level, uniform reward distribution causes a severe credit assignment failure in complex diagnostic tasks. It fails to isolate correct sub-answers and inadvertently reinforces hallucinated reasoning steps within partially correct sequences.*

2.3 The Inherent Co-occurrence of SQL Errors

Beyond the challenge of credit assignment, standard RL formulations typically treat multi-label classification targets as isolated, independent events. However, we hypothesize that the strict, rule-based syntax of SQL inherently couples certain error types together. To empirically quantify these structural dependencies, we conduct an extensive analysis of the NL2SQL-Bugs [8] dataset. This benchmark provides fine-grained, human-annotated error labels that authentically reflect the real-world failure modes of contemporary LLMs in Text-to-SQL tasks.

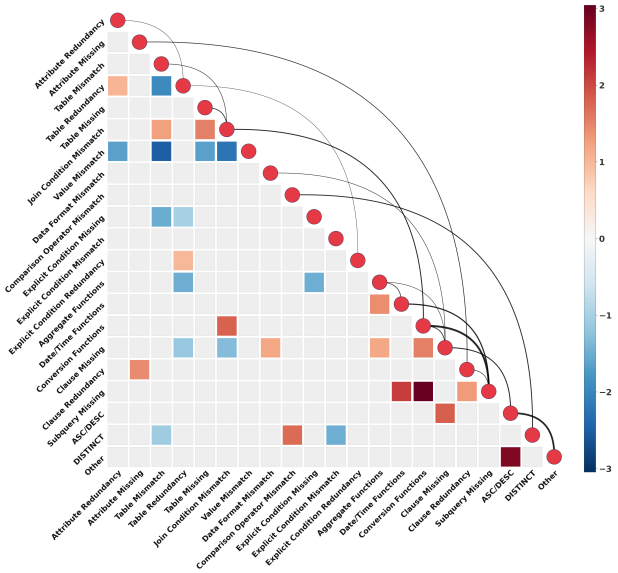


Figure 2: Co-occurrence matrix of error types within the NL2SQL-Bugs [8] dataset. Both axes represent the actual error types identified in the dataset. Cell colors indicate the $\log_2(\text{lift})$ value between pairs of error types. Grey cells denote pairs that either lack a significant co-occurrence relationship ($|\log_2(\text{lift})| < 1$) or have insufficient data to support such a conclusion. The curved lines in the upper-right area visualize these co-occurrence relationships, with line thickness corresponding to the relationship strength ($\log_2(\text{lift})$). To optimize presentation, rows and columns containing only grey cells have been excluded.

We measure the co-occurrence strength between any two error types, u and v, using the $\log_2(\text{Lift})$ metric [26, 27]. The *lift* mathematically quantifies the ratio of the joint probability of observing both errors simultaneously to the product of their independent marginal probabilities: $\text{Lift}(u, v) = \frac{P(u,v)}{P(u)P(v)}$. A $\log_2(\text{Lift})$ value significantly greater than zero indicates that the errors appear together far more frequently than would be expected by random chance.

The resulting co-occurrence matrix, visualized as a network heatmap in Figure 2, unambiguously confirms the existence of profound structural dependencies. For example, a

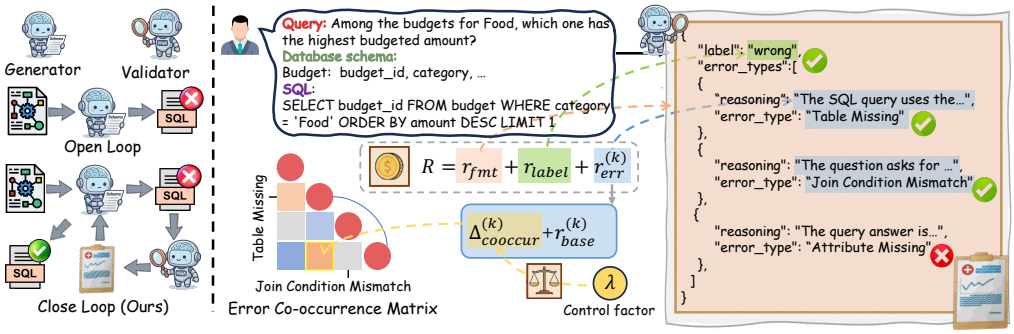


Figure 3: The overall framework of DIAGSQL. The upstream SQL generator produces a preliminary SQL query, which is parsed into a formatted JSON output to conduct structural validation. During the training phase, the model is first warmed up using SFT and then further optimized via an improved GRPO that integrates `Fine-grained Token-level Reward Allocation` (`FTRA`) and `Co-occurrence-aware Reward Shaping` (`CORS`).

highly positive correlation ($\log_2(\text{Lift}) = 1.523$) is explicitly evident between the *Table Missing* and *Join Condition Mismatch* errors. The underlying rationale is firmly grounded in SQL execution logic: if a generator omits a necessary table from the `FROM` clause, any subsequent `JOIN` conditions that depend on that missing table will inevitably be rendered invalid. This inherent error co-occurrence provides a rich source of structural prior knowledge. If harnessed properly, it can explicitly guide the policy model's diagnostic trajectories, penalizing illogical error combinations while rewarding the discovery of concomitant faults.

Observation II: *SQL errors exhibit strong, inherent structural dependencies dictated by SQL syntax constraints. This error co-occurrence provides critical structural prior knowledge that can be explicitly leveraged to guide the model's diagnostic trajectories.*

3 Methodology

Building on these insights, the **DIAGSQL** framework improves upon standard GRPO. As illustrated in Figure 3, DIAGSQL introduces a structured diagnostic pipeline and an improved RL algorithm comprising two core components: `Fine-grained Token-level Reward Allocation (FTRA)` to overcome *Observation I*, and `Co-occurrence-aware Reward Shaping (CORS)` to capitalize on *Observation II*.

3.1 Diagnostic Formatted Output

To enable fine-grained credit assignment and structured reasoning, DIAGSQL enforces a standardized JSON schema for all diagnostic responses $Y = \langle y_{label}, E_{pred}, Z_{pred} \rangle$ (defined in Section 2.1), in which each error type $e_k \in E_{pred}$ is explicitly paired with its reasoning path $z_k \in Z_{pred}$. Beyond guaranteeing parsability for downstream refinement engines, this schema serves as the *structural contract* that `FTRA` relies on: it deterministically delimits non-overlapping, independently verifiable token spans—the JSON skeleton, the correctness label, and each $\langle e_k, z_k \rangle$ pair—so that span-specific rewards can be computed and attributed without ambiguity in the next stage.

3.2 Fine-grained Token-level Reward Allocation (FTRA)

Building on these boundaries, `FTRA` consumes the same Y schema and realizes token-level credit assignment in three steps: (i) parse Y_i into non-overlapping spans, (ii) attach a span-specific scalar reward to each span, and (iii) broadcast the span rewards back to their constituent tokens and reduce the advantage accordingly.

Specifically, DIAGSQL employs a parsing function $f_{parse}(Y_i)$ that segments the token sequence into a set of spans: $\mathcal{S}_i = \{s_{fmt}, s_{label}, s_{err}^{(1)}, \ldots, s_{err}^{(K)}\}$, where s_{fmt} represents the JSON formatting

tokens, s_{label} represents the binary label y_{label} tokens, and $s_{err}^{(k)}$ encapsulates the tokens for the k-th predicted error type e_k and its reasoning z_k. DIAGSQL then defines independent reward functions for each span:

- **Format Reward (r_{fmt}):** A strict binary reward (1 or 0) verifying if the output adheres to the JSON schema.
- **Label Reward (r_{label}):** A binary reward evaluating the correctness of y_{label}.
- **Error Type Base Reward ($r_{base}^{(k)}$):** A continuous reward (e.g., F1 score) evaluating the k-th predicted error e_k and its reasoning z_k against the ground truth error set E_{gt}. Crucially, if the predicted error is hallucinated (i.e., $e_k \notin E_{gt}$), its reasoning has no valid grounding; therefore, $r_{base}^{(k)}$ is strictly set to 0 or a negative penalty.

`FTRA` allocates these rewards strictly to their corresponding token spans. The token-level reward $r_{i,t}$ for the t-th token $y_{i,t}$ is:

$$r_{i,t} = \sum_{s \in \mathcal{S}_i} r_s \cdot \mathbb{I}(y_{i,t} \in s) \tag{3}$$

where $\mathbb{I}(\cdot)$ is the indicator function. DIAGSQL then computes a **Token-level Advantage** $A_{i,t} = (r_s - \mu_s)/\sigma_s$, where μ_s and σ_s are the statistics of the corresponding span rewards across the group G. Substituting A_i with $A_{i,t}$ in Eq. 2 ensures that only the specific tokens contributing to correct sub-answers receive positive reinforcement.

3.3 Co-occurrence-aware Reward Shaping (CORS)

While `FTRA` guarantees precise credit assignment, it evaluates each error type independently. To fully exploit the inherent structural dependencies of SQL syntax, `CORS` dynamically shapes the error type base reward $r_{base}^{(k)}$ using the pre-computed co-occurrence matrix M, where $M_{u,v} = \log_2(\text{Lift}(u, v))$.

For each predicted error $u \in E_{pred}$, `CORS` adjusts its base reward $r_{base}^{(u)}$ to obtain the final error type reward $r_{err}^{(u)}$ by introducing a co-occurrence modifier $\Delta_{cooccur}^{(u)}$:

$$r_{err}^{(u)} = r_{base}^{(u)} + \Delta_{cooccur}^{(u)} \tag{4}$$

Let $\mathcal{E}$ be the set of all possible error types, and $b_v, \hat{b}_v \in \{0, 1\}$ be binary indicator variables for the presence of error v in the ground truth E_{gt} and the prediction E_{pred}, respectively. The modifier is given by:

$$\Delta_{cooccur}^{(u)} = \lambda \sum_{v \in \mathcal{E} \setminus \{u\}} \left[M_{u,v} \cdot b_v \hat{b}_v - |M_{u,v}| \cdot (b_v - \hat{b}_v)^2 \right] \tag{5}$$

where λ controls the strength of the structural inductive bias. This formulation effectively addresses the three critical structural constraints required for diagnostic reasoning:

1. **Associative Bonus ($b_v = 1, \hat{b}_v = 1$):** The model correctly predicts a concomitant error v. The penalty term vanishes, simplifying to $M_{u,v}$. If $M_{u,v} > 0$, the model naturally receives a bonus, encouraging the discovery of structurally coupled errors.
2. **Cascading Penalty ($b_v = 1, \hat{b}_v = 0$):** The ground truth contains a highly correlated error v, but the model misses it. The formulation simplifies to $-|M_{u,v}|$, penalizing the model proportionally to the correlation magnitude to discourage incomplete diagnoses.
3. **Illogical Combination Penalty ($b_v = 0, \hat{b}_v = 1$):** The model hallucinates error v when it is not present in the ground truth. It incurs a penalty of $-|M_{u,v}|$, heavily discouraging combinations of errors that are structurally incompatible or rarely co-occur.
4. **Inactive Pair ($b_v = 0, \hat{b}_v = 0$):** Both terms are exactly zero, so v contributes nothing to $\Delta_{cooccur}^{(u)}$. This fourth, degenerate case makes the shaping signal sparse: pairs that are inactive in both the target and prediction cannot perturb the reward.

Furthermore, the magnitude $|M_{u,v}|$ inherently functions as a continuous soft-threshold mechanism. Error pairs lacking structural relationships ($M_{u,v} \approx 0$) automatically vanish, resulting in a robust, parameter-efficient reward shaping paradigm.

Table 1: Comprehensive performance comparison of various LLMs and DIAGSQL on SQL error detection. Darker green shading indicates higher accuracy. DIAGSQL* utilizes a discretized co-occurrence matrix to strictly prevent data leakage of co-occurrence frequencies on the in-domain NL2SQL-Bugs dataset, while DIAGSQL uses the continuous matrix.

Evaluation Types		GPT-4o	GPT-4o-mini	Claude-3.5-Sonnet	DeepSeek-V3	Gemini-2.0-Flash	DiagSQL* Llama3.1-8B	DiagSQL* Qwen3-8B	DiagSQL Llama3.1-8B	DiagSQL Qwen3-8B
Semantic Error Detection										
		76.81%	71.56%	76.36%	75.02%	76.81%	83.15%	81.37%	88.70%	86.62%
Error Type Detection										
Attribute-related Errors	Attribute Mismatch	32.78%	18.61%	28.61%	24.17%	20.83%	52.64%	55.71%	55.81%	60.65%
	Attribute Redundancy	13.21%	9.43%	7.55%	1.89%	0.00%	22.35%	17.41%	20.00%	19.51%
	Attribute Missing	36.67%	30.00%	35.00%	43.33%	28.33%	20.82%	32.33%	22.45%	30.00%
Table-related Errors	Table Mismatch	14.58%	18.75%	16.67%	11.46%	11.46%	20.54%	27.41%	27.38%	29.33%
	Table Redundancy	36.97%	14.29%	37.82%	35.29%	28.57%	54.85%	58.71%	67.29%	68.32%
	Table Missing	46.97%	22.73%	40.91%	48.48%	36.36%	41.35%	37.65%	45.76%	41.82%
	Join Type Mismatch	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	Join Condition Mismatch	58.23%	46.84%	48.10%	46.84%	34.18%	37.47%	45.16%	39.73%	47.14%
Value-related Errors	Value Mismatch	48.74%	40.76%	40.76%	42.44%	34.45%	53.25%	50.00%	56.82%	48.65%
	Data Format Mismatch	21.43%	14.29%	21.43%	14.29%	7.14%	14.29%	14.29%	15.38%	15.38%
Operator-related Errors	Comparison Operator Mismatch	25.00%	8.33%	25.00%	0.00%	4.17%	25.00%	18.18%	18.18%	36.84%
	Logical Operator Mismatch	43.75%	18.75%	50.00%	31.25%	25.00%	18.75%	25.00%	12.50%	31.25%
Condition-related Errors	Explicit Condition Missing	66.95%	69.49%	71.19%	57.63%	35.59%	57.63%	35.59%	55.77%	39.81%
	Explicit Condition Mismatch	37.89%	45.26%	35.79%	33.68%	24.21%	24.21%	35.79%	21.18%	36.14%
	Explicit Condition Redundancy	31.03%	10.34%	10.34%	10.34%	10.34%	20.68%	23.65%	42.86%	34.62%
	Implicit Condition Missing	14.29%	3.57%	0.00%	14.29%	3.57%	3.57%	0.00%	4.76%	0.00%
Function-related Errors	Aggregate Functions	35.21%	15.49%	36.62%	30.99%	18.31%	35.21%	36.62%	42.86%	37.50%
	Window Functions	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	33.33%
	Date/Time Functions	9.52%	4.76%	14.29%	28.57%	9.52%	14.29%	9.52%	23.53%	15.79%
	Conversion Functions	0.00%	0.00%	0.00%	9.09%	0.00%	0.00%	25.00%	0.00%	25.00%
	Math Functions	40.00%	0.00%	30.00%	0.00%	0.00%	0.00%	14.29%	0.00%	14.29%
	String Functions	5.26%	0.00%	0.00%	10.53%	15.79%	10.53%	0.00%	6.25%	0.00%
	Conditional Functions	0.00%	0.00%	2.33%	2.33%	0.00%	14.86%	14.86%	29.73%	27.03%
Clause-related Errors	Clause Missing	19.15%	22.34%	18.09%	25.53%	17.02%	18.09%	17.02%	32.50%	16.67%
	Clause Redundancy	8.11%	13.51%	5.41%	16.22%	13.51%	13.51%	21.21%	25.81%	21.21%
Subquery-related Errors	Subquery Missing	12.12%	0.00%	3.03%	6.06%	9.09%	12.12%	12.12%	23.33%	18.52%
	Subquery Mismatch	14.29%	28.57%	28.57%	28.57%	14.29%	14.29%	14.29%	28.57%	14.29%
	Partial Query	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Other Errors	ASC/DESC	83.33%	58.33%	66.67%	58.33%	66.67%	27.27%	13.64%	9.09%	27.27%
	DISTINCT	6.82%	1.14%	4.55%	0.00%	0.00%	14.75%	14.75%	30.99%	14.75%
	Other	2.86%	0.00%	8.57%	0.00%	8.57%	0.00%	0.00%	3.03%	3.33%
Overall		33.02%	24.61%	29.95%	28.10%	21.37%	36.54%	37.86%	41.01%	39.55%

3.4 Training Pipeline

The training of DIAGSQL follows a two-stage pipeline. In the first stage, DIAGSQL utilizes Supervised Fine-Tuning (SFT) to warm up the model. This step familiarizes the language model with the strict JSON output schema and imparts basic diagnostic reasoning capabilities, providing a necessary starting policy that generates structurally valid responses. The SFT corpus consists of 2,000 (natural-language question, buggy SQL, diagnostic label) triples synthesized from OMNISQL [28] via controlled grammatical mutations and schema element perturbations, with the resulting error co-occurrence distribution explicitly aligned to NL2SQL-Bugs; see Appendix B for construction details. After SFT, 97.5% of responses are valid JSON. At inference time, the remaining malformed outputs are regenerated once; during RL, an unparsable output receives zero span rewards except for the negative format reward and therefore cannot create spurious token-level credit. In the second stage, DIAGSQL initializes the policy with the SFT checkpoint and applies the improved GRPO algorithm. This RL phase explores the vast reasoning space, utilizing `FTRA` to strictly penalize hallucinated reasoning steps and `CORS` to shape the trajectories according to SQL error co-occurrence semantics, ultimately converging to a highly precise and structurally coherent diagnostic validator.

4 Experiments

Experimental Setup. To evaluate the effectiveness of DIAGSQL, we design a comprehensive set of experiments across two distinct tasks: (1) **SQL Error Detection**, where the model must identify the presence and specific types of errors in a given SQL query, and (2) **Text-to-SQL Refinement**, where the fine-grained diagnostic feedback is utilized by an upstream generator to correct the erroneous SQL. Please refer to Appendix B for comprehensive details regarding the experimental setup, including dataset construction, evaluation metrics, and implementation specifics.

Diagnostic Performance on NL2SQL-Bugs. Table 1 presents a detailed performance comparison on the in-domain NL2SQL-Bugs dataset, spanning both coarse-grained Semantic Error Detection and fine-grained Error Type Detection. DIAGSQL delivers consistent gains over both its base open-weights models and state-of-the-art proprietary LLMs. While massive models like GPT-4o [29] and DEEPSEEK-V3 [30] demonstrate reasonable proficiency in identifying coarse semantic errors, they struggle significantly with the fine-grained diagnostic task, often failing to pinpoint the exact nature

Table 2: Text-to-SQL execution accuracy (EX, %) on the BIRD [31] and SPIDER [32] development sets, where each refinement method is applied to SQLs generated by GPT-4O [29] (above the double divider) or OpenSearch-SQL [33] (below). **Bold**/underlined mark the best/second-best result per column; values in parentheses are relative improvements (green) or degradations (red) over the corresponding baseline generator. See Appendix B for BIRD difficulty abbreviations, the $K{=}1$ single-pass protocol, and the role of DIAGSQL*.

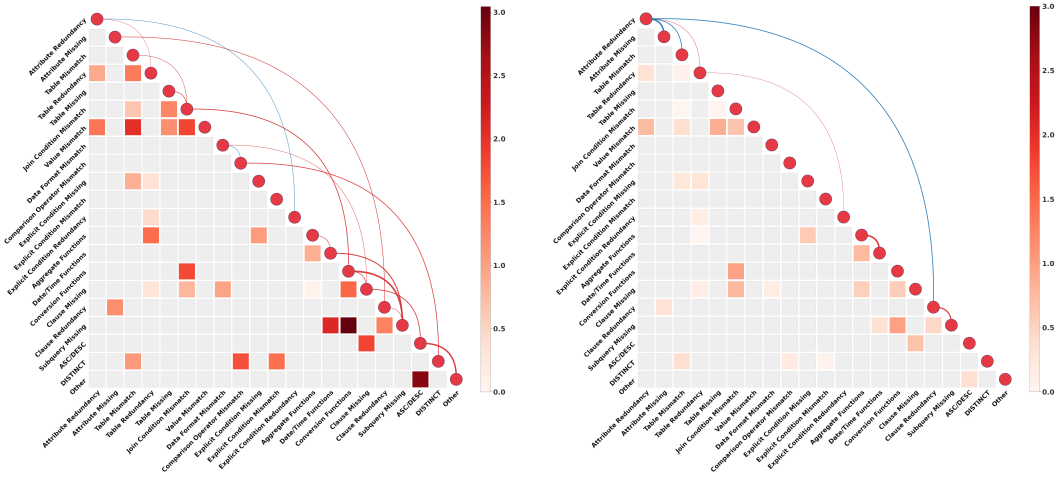
Methods	BIRD				SPIDER				
	Simp.	Mode.	Chal.	Overall	Easy	Medium	Hard	Extra	Overall
GPT-4o [34]	63.35	44.18	45.52	55.87	91.53	72.42	76.44	58.43	75.44
+Self-correction [13]	62.92 (-0.68%)	43.32 (-1.95%)	42.07 (-7.58%)	55.02 (-1.52%)	90.73 (-0.87%)	71.75 (-0.93%)	75.29 (-1.50%)	56.02 (-4.12%)	74.47 (-1.29%)
+Self-debugging [14]	64.00 (+1.03%)	45.04 (+1.95%)	46.90 (+3.03%)	56.65 (+1.40%)	93.55 (+2.21%)	74.22 (+2.49%)	77.01 (+0.75%)	58.43 (±0.00%)	76.79 (+1.79%)
+Self-consistency [35]	65.73 (+3.76%)	49.14 (+11.23%)	44.14 (-3.03%)	58.67 (+5.01%)	92.74 (+1.32%)	73.09 (+0.93%)	77.59 (+1.50%)	59.04 (+1.04%)	76.31 (+1.15%)
+Cross-consistency [15]	65.84 (+3.93%)	49.57 (+12.20%)	45.52 (±0.00%)	59.00 (+5.60%)	93.15 (+1.77%)	72.87 (+0.62%)	78.16 (+2.25%)	60.24 (+3.10%)	76.60 (+1.54%)
+Multiple-prompt [16]	66.38 (+4.78%)	48.06 (+8.78%)	44.83 (-1.52%)	58.80 (+5.24%)	91.94 (+0.45%)	73.32 (+1.24%)	78.74 (+3.01%)	57.83 (-1.03%)	76.21 (+1.02%)
+Refiner [9]	64.22 (+1.37%)	45.91 (+3.92%)	47.59 (+4.55%)	57.11 (+2.22%)	93.15 (+1.77%)	78.70 (+8.67%)	77.01 (+0.75%)	58.43 (±0.00%)	78.63 (+4.23%)
+SQLFixAgent [11]	65.62 (+3.58%)	48.28 (+9.28%)	48.28 (+6.06%)	58.74 (+5.14%)	93.55 (+2.21%)	90.36 (+24.77%)	77.59 (+1.50%)	69.28 (+18.57%)	85.59 (+13.45%)
+MAGIC [12]	67.57 (+6.66%)	49.57 (+12.20%)	46.21 (+1.52%)	60.10 (+7.57%)	–	–	–	–	85.69 (+13.59%)
+SHARE [34]	70.81 (+11.78%)	56.25 (+27.32%)	46.90 (+3.03%)	64.14 (+14.80%)	93.95 (+2.64%)	90.13 (+24.45%)	78.16 (+2.25%)	70.48 (+20.62%)	85.88 (+13.84%)
+ErrorLLM [7] (DEEPSEEK-V3)	71.08 (+12.20%)	56.90 (+28.79%)	47.59 (+4.55%)	64.99 (+16.32%)	94.76 (+3.53%)	91.26 (+26.01%)	78.16 (+2.25%)	69.28 (+18.57%)	86.36 (+14.48%)
+ErrorLLM [7] (GPT-4o)	73.08 (+15.36%)	57.97 (+31.21%)	48.97 (+7.58%)	66.23 (+18.54%)	**95.56** (+4.40%)	91.03 (+25.70%)	**79.89** (+4.51%)	70.48 (+20.62%)	86.94 (+15.24%)
+DIAGSQL* (LLAMA3.1-8B)	72.54 (+14.51%)	58.06 (+31.42%)	46.53 (+2.22%)	65.71 (+17.61%)	94.76 (+3.53%)	91.26 (+26.01%)	78.16 (+2.25%)	69.28 (+18.57%)	86.36 (+14.48%)
+DIAGSQL* (QWEN3-8B)	71.46 (+12.80%)	56.34 (+27.52%)	47.92 (+5.27%)	64.67 (+15.75%)	93.95 (+2.64%)	90.13 (+24.45%)	77.59 (+1.50%)	68.67 (+17.53%)	85.49 (+13.32%)
+DIAGSQL (LLAMA3.1-8B)	74.92 (+18.26%)	**58.92** (+33.36%)	50.00 (+9.84%)	67.73 (+21.23%)	**95.56** (+4.40%)	**91.70** (+26.62%)	78.74 (+3.01%)	70.48 (+20.62%)	**87.04** (+15.38%)
+DIAGSQL (QWEN3-8B)	**76.11** (+20.14%)	58.06 (+31.42%)	**51.39** (+12.90%)	**68.32** (+22.28%)	94.76 (+3.53%)	90.58 (+25.08%)	78.74 (+3.01%)	**71.08** (+21.65%)	86.46 (+14.61%)
OpenSearch-SQL [33]	75.46	64.66	66.90	71.38	92.74	88.57	79.31	**69.88**	85.01
+SQLFixAgent [11]	74.05 (-1.87%)	60.34 (-6.68%)	62.07 (-7.22%)	68.77 (-3.66%)	93.55 (+0.87%)	89.01 (+0.50%)	78.16 (-1.45%)	67.47 (-3.45%)	84.82 (-0.22%)
+SHARE [34]	73.19 (-3.01%)	57.76 (-10.67%)	47.59 (-28.86%)	66.10 (-7.40%)	–	–	–	–	–
+ErrorLLM [7] (DEEPSEEK-V3)	76.43 (+1.29%)	65.09 (+0.67%)	64.83 (-3.09%)	71.90 (+0.73%)	95.16 (+2.61%)	91.48 (+3.29%)	78.73 (-0.73%)	**69.88** (±0.00%)	86.75 (+2.05%)
+ErrorLLM [7] (GPT-4o)	76.65 (+1.58%)	65.52 (+1.33%)	66.21 (-1.03%)	72.29 (+1.27%)	**95.97** (+3.48%)	91.26 (+3.04%)	**79.89** (+0.73%)	**69.88** (±0.00%)	**87.04** (+2.39%)
+DIAGSQL (LLAMA3.1-8B)	76.11 (+0.86%)	64.95 (+0.45%)	66.67 (-0.34%)	71.84 (+0.64%)	95.16 (+2.61%)	**91.70** (+3.53%)	78.74 (-0.72%)	69.28 (-0.86%)	86.75 (+2.05%)
+DIAGSQL (QWEN3-8B)	**76.76** (+1.72%)	**65.81** (+1.78%)	**68.06** (+1.73%)	**72.62** (+1.74%)	94.76 (+2.18%)	91.48 (+3.29%)	79.31 (±0.00%)	69.28 (-0.86%)	86.65 (+1.93%)

of the SQL fault. In contrast, DIAGSQL substantially improves the diagnostic precision of compact 8B models, establishing a new state-of-the-art for accurate error typing.

Crucially, the performance gains are predominantly concentrated in error categories that exhibit strong structural couplings. This pattern indicates that explicitly modeling structural dependencies improves the validator's grasp of SQL syntax. By properly resolving the credit assignment challenge through token-level rewards (FTRA) and leveraging error co-occurrence priors (CORS), DIAGSQL enables small models to handle complex diagnostic reasoning that challenges much larger general-purpose LLMs. Furthermore, even under the strict non-leakage constraint of DIAGSQL*, the model maintains highly competitive diagnostic accuracy, suggesting that learning the structural shape of errors is fundamentally more valuable than merely memorizing their statistical frequencies. As a complement to this picture, we also observe that DIAGSQL is slightly weaker than GPT-4O on a few rare or structurally-isolated sub-categories (e.g., *ASC/DESC*, *Explicit/Implicit Condition Missing*, and *Logical Operator Mismatch*), where the co-occurrence prior carries little signal and the remaining gains must come from raw lexical pattern matching at which much larger proprietary LLMs retain an edge; this behavior is consistent with the qualitative analysis in Appendix F and the saturation trend of the co-occurrence prior discussed in Appendix D.

Text-to-SQL Refinement on BIRD and Spider. We further assess whether DIAGSQL's precise diagnostic feedback translates into tangible end-to-end performance gains by integrating it as a verifier in a refinement pipeline on the out-of-domain BIRD and Spider benchmarks (Table 2). Regardless of the backbone model utilized, DIAGSQL consistently drives substantial improvements in execution accuracy over the initial SQL queries. Notably, DIAGSQL achieves the best overall performance on BIRD and matches or slightly exceeds the strongest baseline (ErrorLLM with GPT-4O) on Spider, while operating entirely on lightweight 8B open-weights models rather than relying on costly proprietary APIs.

The improvements are especially pronounced on the more challenging data slices, where initial queries frequently suffer from complex, multi-layered logical faults that opaque self-correction methods fail to disentangle. Furthermore, DIAGSQL exhibits remarkable robustness across different strengths of upstream generators. Even when refining high-quality outputs from a powerful generator like OpenSearch-SQL—a scenario where many traditional self-correction techniques degrade performance due to unwarranted hallucinated edits—DIAGSQL provides stable, targeted corrections that strictly enhance the query. This confirms that DIAGSQL functions effectively as a generator-agnostic verifier, providing actionable, highly reliable feedback that transforms Text-to-SQL into a closed-loop system. All numbers in Table 2 are obtained with a **single** refinement pass ($K{=}1$), i.e., one validator-guided rewrite per query; sensitivity to the refinement budget is analyzed in Appendix G.



(a) GRPO-trained QWEN3-8B. (b) DIAGSQL-trained QWEN3-8B.

Figure 4: Absolute deviation $\left|\log_2 \text{lift}^{\text{pred}} - \log_2 \text{lift}^{\text{GT}}\right|$ between the predicted and ground-truth pairwise error co-occurrence structures on NL2SQL-Bugs. Blue arcs are pairs hallucinated by the model (absent in GT but co-firing in predictions); red arcs are genuine GT pairs that the model fails to co-predict.

Does DIAGSQL Truly Learn Error Co-occurrence? To examine whether CORS internalizes the structural co-occurrence prior of SQL errors, we compute, for each error pair (i, j), the absolute deviation $\left|\log_2 \text{lift}^{\text{pred}}(i,j) - \log_2 \text{lift}^{\text{GT}}(i,j)\right|$ between the predicted and ground-truth pairwise co-occurrence on NL2SQL-Bugs. The result is shown in Figure 4: blue arcs mark pairs hallucinated by the model (absent in GT but co-firing in predictions); red arcs mark genuine GT pairs that the model fails to co-predict.

The vanilla GRPO model (Figure 4a) shows two long-range blue hallucination arcs (*Attribute Redundancy*↔*Subquery Missing* and *Table Mismatch*↔*Table Redundancy*) together with diffuse, off-block deviation cells, indicating that GRPO fabricates implausible multi-error combinations rather than respecting the GT dependency graph. In contrast, DIAGSQL (Figure 4b) substantially suppresses these hallucinated arcs—with virtually no new hallucinated co-occurrences introduced, the residual mismatches are predominantly red recall-side gaps rather than additional blue hallucinations—and its heatmap mass concentrates on the structurally meaningful blocks (*Join Condition Mismatch*↔*Table Mismatch/Redundancy*, *Subquery Missing*↔*Aggregate/Date-Time Functions*) that coincide with the high-lift cliques used to build CORS. This shift from *globally diffuse, hallucinated* co-occurrence to *locally concentrated, GT-aligned* co-occurrence shows that *vanilla GRPO fails to acquire the inherent co-occurrence structure of SQL errors, while* DIAGSQL *genuinely internalizes this structural prior through the joint effect of* FTRA *and* CORS.

Ablation Study To systematically isolate the contributions of our proposed mechanisms, we conduct an ablation study across both diagnostic and refinement tasks (Table 3). The removal of FTRA inflicts the most severe single-component degradation across all metrics. This validates our hypothesis that sequence-level rewards tend to produce noisy credit assignment in structured tasks; without precise token-level allocation, the policy inadvertently reinforces hallucinated reasoning alongside correct judgments, severely compromising downstream refinement stability. Similarly, omitting CORS yields consistent performance drops, underscoring that without explicit SQL

Table 3: Ablation studies of DIAGSQL.

Methods	NL2SQL-Bugs		BIRD	Spider
	Semantic Error	Error Type	SQL Refinement	SQL Refinement
		QWEN3-8B		
DIAGSQL	86.62	39.55	68.32	86.46
w/o FTRA	83.40 (-3.22)	36.98 (-2.57)	63.10 (-5.22)	83.17 (-3.29)
w/o CORS	85.53 (-1.09)	38.22 (-1.33)	64.86 (-3.46)	84.14 (-2.32)
w/o SFT	80.13 (-6.49)	36.71 (-2.84)	60.23 (-8.09)	81.53 (-4.93)
→SFT + GRPO	81.42 (-5.20)	36.56 (-2.99)	61.41 (-6.91)	81.91 (-4.55)
→SFT + DAPO	81.96 (-4.66)	36.75 (-2.80)	62.91 (-5.41)	82.40 (-4.06)
w/o DIAGSQL	59.61 (-27.01)	33.14 (-6.41)	50.20 (-18.12)	72.24 (-14.22)
		LLAMA3.1-8B		
DIAGSQL	88.70	41.01	67.73	87.04
w/o DIAGSQL	56.99 (-31.71)	19.48 (-21.53)	48.24 (-19.49)	70.02 (-17.02)

co-occurrence priors, the model struggles to maintain coherence when diagnosing interdependent, multi-clause errors.

Furthermore, our analysis highlights the necessity of the complete training pipeline. Bypassing the SFT warm-up phase causes a sharp collapse in performance, demonstrating that a reasonable initialization within the structured JSON output space is a fundamental prerequisite for effective reward shaping. Substituting our tailored algorithm with off-the-shelf RL recipes such as vanilla GRPO or DAPO yields results that significantly underperform the full DIAGSQL framework. This confirms that the success of DIAGSQL stems directly from its diagnostic-specific reward architecture rather than the general application of reinforcement learning. Finally, completely removing the DIAGSQL results in a drastic deterioration across all metrics and backbones, demonstrating that our principled treatment of credit assignment and structural priors is the driving force behind the model's superior diagnostic capabilities.

5 Conclusion

In this work, we tackle the critical open-loop bottleneck in Text-to-SQL generation by introducing DIAGSQL, a dedicated diagnostic validator. Our empirical analysis reveals two fundamental challenges in training such validators via standard GRPO: severe credit assignment failures stemming from sequence-level uniform reward distributions, and the underutilization of structural error dependencies inherent in SQL syntax. To address these limitations, we propose an enhanced training framework that incorporates Fine-grained Token-level Reward Allocation (`FTRA`) to precisely isolate and reinforce correct reasoning steps, and `Co-occurrence-aware Reward Shaping (CORS)` to dynamically inject structural inductive biases using a pre-computed error co-occurrence matrix. Extensive out-of-domain evaluations demonstrate that despite its compact 8B-parameter scale, DIAGSQL significantly outperforms state-of-the-art proprietary LLMs in fine-grained SQL error detection. Moreover, when deployed as a verifier to guide iterative Text-to-SQL refinement, DIAGSQL yields substantial improvements in execution accuracy on both the BIRD and Spider benchmarks, successfully realizing an effective closed-loop Text-to-SQL paradigm. Ultimately, our findings suggest that integrating structural priors into reward shaping, together with fine-grained credit allocation, offers a promising avenue for advancing alignment algorithms in complex structured reasoning tasks.

Acknowledgments and Disclosure of Funding

This work was conducted while Weibin Liao was an intern at LIGHTSPEED STUDIOS. LIGHTSPEED STUDIOS provided internship and computational support for this research. The authors declare no other competing interests.

References

[1] Weibin Liao, Xin Gao, Tianyu Jia, Rihong Qiu, Yifan Zhu, Yang Lin, Xu Chu, Junfeng Zhao, and Yasha Wang. Learnat: Learning nl2sql with ast-guided task decomposition for large language models. *arXiv preprint arXiv:2504.02327*, 2025.

[2] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. A survey of text-to-sql in the era of llms: Where are we, and where are we going? *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2025.

[3] George Katsogiannis-Meimarakis and Georgia Koutrika. A survey on deep learning approaches for text-to-sql. *The VLDB Journal*, 32(4):905–936, 2023.

[4] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2):1–124, 2023.

[5] Weibin Liao, Jian-guang Lou, and Haoyi Xiong. Hyfunc: Accelerating llm-based function calls for agentic ai through hybrid-model cascade and dynamic templating. In *Conference on Knowledge Discovery and Data Mining (KDD)*, pages 819–830, 2026.

[6] Tianlong Wang, Yuhang Wang, Weibin Liao, Xin Gao, Xinyu Ma, Yang Lin, Yasha Wang, and Liantao Ma. Search for truth from reasoning: A dynamic representation editing framework for steering llm trajectories. *arXiv preprint arXiv:2606.28589*, 2026.

[7] Zijin Hong, Hao Chen, Zheng Yuan, Qinggang Zhang, Luyao Zhuang, Qing Liao, Feiran Huang, Yangqiu Song, and Xiao Huang. Errorllm: Modeling sql errors for text-to-sql refinement. *arXiv preprint arXiv:2603.03742*, 2026.

[8] Xinyu Liu, Shuyu Shen, Boyan Li, Nan Tang, and Yuyu Luo. Nl2sql-bugs: A benchmark for detecting semantic errors in nl2sql translation. In *Conference on Knowledge Discovery and Data Mining (KDD)*, pages 5662–5673, 2025.

[9] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. Mac-sql: A multi-agent collaborative framework for text-to-sql. In *International Conference on Computational Linguistics (COLING)*, 2024.

[10] Bowen Cao, Deng Cai, Zhisong Zhang, Yuexian Zou, and Wai Lam. On the worst prompt performance of large language models. *Advances in Neural Information Processing Systems (NeurIPS)*, 37:69022–69042, 2024.

[11] Jipeng Cen, Jiaxin Liu, Zhixu Li, and Jingjing Wang. Sqlfixagent: Towards semantic-accurate text-to-sql parsing via consistency-enhanced multi-agent collaboration. In *Conference on Artificial Intelligence (AAAI)*, 2025.

[12] Arian Askari, Christian Poelitz, and Xinye Tang. Magic: Generating self-correction guideline for in-context text-to-sql. In *Conference on Artificial Intelligence (AAAI)*, 2025.

[13] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024.

[14] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024.

[15] Zhishuai Li, Xiang Wang, Jingjing Zhao, Sun Yang, Guoqing Du, Xiaoru Hu, Bin Zhang, Yuxiao Ye, Ziyue Li, Hangyu Mao, et al. Pet-sql: A prompt-enhanced two-round refinement of text-to-sql with cross-consistency. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, 2025.

[16] Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. MCS-SQL: Leveraging multiple prompts and multiple-choice selection for text-to-SQL generation. In *International Conference on Computational Linguistics (COLING)*, 2025.

[17] Weibin Liao, Xu Chu, and Yasha Wang. Tpo: Aligning large language models with multi-branch & multi-step preference trees. In *International Conference on Learning Representations (ICLR)*, 2025.

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

[19] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

[20] Hongxin Ding, Baixiang Huang, Yue Fang, Weibin Liao, Xinke Jiang, Zheng Li, Junfeng Zhao, and Yasha Wang. Promed: Shapley information gain guided reinforcement learning for proactive medical llms. *arXiv preprint arXiv:2508.13514*, 2025.

[21] Prasanna Parthasarathi, Mathieu Reymond, Boxing Chen, Yufei Cui, and Sarath Chandar. Grpo-$\backslash$ $lambda$: Credit assignment improves llm reasoning. *arXiv preprint arXiv:2510.00194*, 2025.

[22] Hieu Tran, Zonghai Yao, and Hong Yu. Exploiting tree structure for credit assignment in rl training of llms. *arXiv preprint arXiv:2509.18314*, 2025.

[23] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:27730–27744, 2022.

[24] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[25] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[26] Min-Ling Zhang and Lei Wu. Lift: Multi-label learning with label-specific features. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 37(1):107–120, 2014.

[27] Maximillian Schich and Michele Coscia. Exploring co-occurrence on a meso and global level using network analysis and rule mining. In *Minign and Learning with Graphs, KDD Workshop*, 2011.

[28] Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, et al. Omnisql: Synthesizing high-quality text-to-sql data at scale. *International Conference on Very Large Data Bases (VLDB)*, 18(11):4695–4709, 2025.

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

[30] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[31] Jinyang Li, Binyuan Hui, GE QU, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM already serve as a database interface? a BIg bench for large-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[32] 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. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[33] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment. In *Conference on Management of Data (SIGMOD)*, 2025.

[34] Ge Qu, Jinyang Li, Bowen Qin, Xiaolong Li, Nan Huo, Chenhao Ma, and Reynold Cheng. SHARE: An SLM-based hierarchical action CorREction assistant for text-to-SQL. In *Association for Computational Linguistics (ACL)*, 2025.

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

[36] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, et al. A survey on in-context learning. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1107–1128, 2024.

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

[38] Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems (NeurIPS)*, 36:36339–36348, 2023.

[39] Hanchong Zhang, Ruisheng Cao, Lu Chen, Hongshen Xu, and Kai Yu. Act-sql: In-context learning for text-to-sql with automatically-generated chain-of-thought. In *Findings of Empirical Methods in Natural Language Processing (EMNLP)*, pages 3501–3532, 2023.

[40] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *International Conference on Very Large Data Bases (VLDB)*, 17(5):1132–1145, 2024.

[41] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. In *International Conference on Learning Representations (ICLR)*, 2025.

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

[43] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *Conference on Artificial Intelligence (AAAI)*, volume 37, pages 13067–13075, 2023.

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

[45] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

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

[47] Bowen Cao, Weibin Liao, Yushi Sun, Dong Fang, Haitao Li, and Wai Lam. Apex-sql: Talking to the data via agentic exploration for text-to-sql. *arXiv preprint arXiv:2602.16720*, 2026.

[48] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin SU, ZHAOQING SUO, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. In *International Conference on Learning Representations (ICLR)*, 2025.

[49] Jingwen Liu, Weibin Liao, Xin Gao, Junfeng Zhao, and Yasha Wang. Database context compression for text-to-sql on real-world large databases. *arXiv preprint arXiv:2606.28601*, 2026.

[50] Weibin Liao, Jingwen Liu, Rihong Qiu, Xin Gao, Xinyu Ma, Hongxin Ding, Yue Fang, Yifan Zhu, Yuheng Huang, Junfeng Zhao, et al. Schematoken: Tokenizing database schemas for efficient and robust text-to-sql linking. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 45993–46015, 2026.

[51] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

[52] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[53] Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. Longlora: Efficient fine-tuning of long-context large language models. In *International Conference on Learning Representations (ICLR)*, 2024.

[54] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations (ICLR)*, 2024.

[55] Bowen Cao, Deng Cai, and Wai Lam. InfiniteICL: Breaking the limit of context window size via long short-term memory transformation. In *Findings of Association for Computational Linguistics (ACL)*, pages 11402–11415, 2025.

[56] Yihan Wang, Peiyu Liu, and Xin Yang. Linkalign: Scalable schema linking for real-world large-scale multi-database text-to-sql. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 977–991, 2025.

A Related Work

LLM-based Text-to-SQL. Large Language Models (LLMs) have enabled two dominant paradigms for text-to-SQL: in-context learning (ICL) [36] and task-specific fine-tuning. ICL, or prompt engineering, has emerged as the primary approach for leveraging powerful proprietary and open-source LLMs. Techniques initially developed for other NLP tasks, such as Chain-of-Thought (CoT) [37] prompting, have been successfully adapted to text-to-SQL and shown to substantially improve generation quality. Building on CoT, methods such as DIN-SQL [38], ACT-SQL [39], and DAIL-SQL [40] augment LLMs with richer exemplars or external knowledge sources, while other approaches employ multi-agent collaboration and iterative refinement [9, 41, 42] or dynamically construct demonstrations via online example generation. In parallel, fine-tuning has emerged as a complementary line of work, particularly important for handling complex queries and domain-specific or real-world database environments. Recent advances include two-stage frameworks for SQL generation and refinement, decomposing schema understanding and SQL synthesis into dual models [43, 44], and applying reinforcement learning to align generated queries with execution-level feedback [1]. *However, despite these advances in generating preliminary SQL queries, both ICL and fine-tuning approaches still struggle to produce error-free outputs in complex schema environments during the first pass, highlighting the critical necessity for a dedicated diagnostic and refinement stage.*

Text-to-SQL Refinement. Given that LLM-based systems still frequently misinterpret complex questions or violate syntactic constraints during first-pass generation, a refinement stage has become an essential component in modern text-to-SQL pipelines. Dedicated refinement frameworks such as MAC-SQL [9], SQLFixAgent [11], MAGIC [12], and SHARE [34] have been proposed to correct logical and semantic errors in generated queries. Two paradigms dominate this landscape: Self-debugging [14] and Self-correction [13, 38]. Self-debugging iteratively revises SQL using execution feedback, but is fundamentally limited by the fact that only a small fraction of erroneous queries trigger explicit execution failures. Self-correction relies on the LLM's internal reasoning to analyze and fix potential mistakes without external feedback. *However, without explicit error modeling grounded in fine-grained credit assignment, and by ignoring the inherent structural dependencies among SQL errors, existing refinement methods exhibit low detection precision and frequently hallucinate corrections on already-correct SQL queries, which directly motivates the structured diagnostic pipeline and co-occurrence-aware reinforcement learning in* DIAGSQL.

B Experimental Setup Details

Training Dataset Construction. Since gathering a large-scale dataset of SQL errors paired with detailed reasoning is challenging, we utilize a synthetic data generation approach. Based on correct queries from Omnisql [28], we systematically synthesize erroneous SQL statements by applying grammatical mutations and schema element perturbations (e.g., dropping clauses, altering join keys). Crucially, to ensure the validity of our `Co-occurrence-aware Reward Shaping (CORS)`, we strictly align the error co-occurrence distribution in our synthetic dataset with the NL2SQL-Bugs [8] dataset, as it accurately reflects the real-world error patterns and failure modes of contemporary LLMs in Text-to-SQL generation. For the SFT warm-up stage, we randomly select 2,000 diverse instances from the synthesized data.

Evaluation Datasets. To rigorously evaluate our models, we employ three distinct datasets: NL2SQL-Bugs [8], Spider [32], and BIRD [31].

- **NL2SQL-Bugs** serves as our primary dataset for the Error Detection task, containing expert-annotated fine-grained SQL errors based on the BIRD development set.
- **Spider** and **BIRD** are utilized for the end-to-end Text-to-SQL Refinement task to assess whether our diagnostic feedback practically advances query correction. Following their official splits, SPIDER queries are grouped into *Easy*, *Medium*, *Hard*, and *Extra* difficulty tiers, while BIRD queries are grouped into *Simple (Simp.)*, *Moderate (Mode.)*, and *Challenging (Chal.)* tiers; we report per-tier and overall EX accordingly.

Because NL2SQL-Bugs is used for evaluating error detection and its co-occurrence distribution guides our synthetic training data, evaluating on NL2SQL-Bugs is considered an *in-domain* assess-

ment. In contrast, the end-to-end refinement evaluations on Spider and BIRD assess *out-of-domain* generalization. To prevent any potential data leakage regarding error frequencies on the test set during the in-domain evaluation, we introduce a rigorous variant denoted as **DIAGSQL***. DIAGSQL* utilizes a discretized co-occurrence matrix for `CORS`: the $\log_2(\text{lift})$ values are clamped to 1 if > 1, 0 if between -1 and 1, and -1 if < -1. This ensures DIAGSQL* only leverages the binary existence of structural dependencies without leaking exact co-occurrence frequencies from the evaluation set. Accordingly, in Table 2 we report DIAGSQL* only in the GPT-4O block—to corroborate the non-leakage property via the in-domain control—while in the OpenSearch-SQL block we report only DIAGSQL, since Spider/BIRD are strictly out-of-domain and the DIAGSQL*/DIAGSQL distinction becomes moot there.

Implementation Details. We utilize QWEN3-8B [25] and LLAMA3.1-8B [45] as our backbone models. For the SFT stage, we train the models for 3 epochs with a global batch size of 128. We employ a cosine learning rate scheduler with a peak learning rate of 2.0e-5 and a 5% warm-up ratio, ensuring the model acquires basic JSON formatting and diagnostic reasoning capabilities. During the subsequent RL stage, we set the training batch size to 64, group size $G = 16$, total optimization steps to 400, and maximum generation length to 768 tokens. We optimize the policy using a learning rate of 2.0e-6 with a linear scheduler and 100 warm-up steps. For the reward weights in `FTRA`, we assign 0.1 to the format reward, 1.0 to the label reward, and 3.0 to the fine-grained error type reward to ensure the gradient signals from specific reasoning steps are prominent. To mitigate conservative prediction biases (where the model avoids predicting errors due to symmetric penalties), we employ an asymmetric reward structure: correct identification of a buggy SQL yields +1.5, whereas misclassifying a correct SQL as buggy incurs only a -0.5 penalty. A minor entropy coefficient of 0.0003 is applied to prevent premature convergence. All training and evaluation are conducted on four 96GB NVIDIA H20 GPUs using bfloat16 precision, memory-efficient attention, and gradient checkpointing. Unless stated otherwise, all Text-to-SQL refinement numbers reported in the main paper (including Table 2) use a single refinement pass (K=1); the sensitivity of DIAGSQL to a larger refinement budget is analyzed in Appendix G.

Models and Baselines. We systematically benchmark DIAGSQL against state-of-the-art proprietary LLMs (e.g., GPT-4O, CLAUDE-3.5-SONNET, DEEPSEEK-V3) for zero-shot diagnostic evaluation. For the Text-to-SQL Refinement task, we compare against a comprehensive suite of refinement paradigms in a unified setting for fair comparison. To avoid accuracy bias introduced by additional experimental errors, we directly adopt the reported results from baseline methods when their experimental settings match ours. For the remaining baselines, we reproduce their results using the official code provided by their respective authors. We utilize the same backbone generator (GPT-4O and OpenSearch-SQL) for fair comparison. The detailed baseline configurations are as follows:

- **Prompt-based Self-correction:**
  - *Self-correction* [13]: Based on the prompt "Review your previous SQL and find problems with your SQL", which includes an implicit hint that the SQL is problematic.
- **Self-debugging with Execution Feedback:**
  - *Self-debugging* [14]: Refines the query iteratively based on the prompt "Fix the generated SQL with the following execution feedback".
- **Self-consistency:**
  - *C3-SQL / Self-consistency* [35]: The vanilla self-consistency module extracted from C3-SQL.
  - *PET-SQL / Cross-consistency* [15]: The cross-consistency module extracted from PET-SQL.
  - *MCS-SQL / Multiple-prompt* [16]: The multiple-prompt consistency module, using the default configuration provided by their official repository.
- **Dedicated Multi-agent Refinement:**
  - *MAC-SQL / Refiner* [9]: We utilize the Refiner module component with its original prompt.
  - *SQLFixAgent* [11]: Implemented using the default setting while replacing the backbone LLM with GPT-4O. The prompt is slightly adjusted for a unified output protocol.

– *MAGIC* [12]: Re-generates the corresponding guideline using GPT-4o and feeds the generated guideline into the subsequent text-to-SQL refinement.
– *SHARE* [34]: We directly report their results on refining GPT-4o generated SQLs, which utilizes their fine-tuned open-source models alongside a proprietary LLM.
– *ErrorLLM* [7]: A recently proposed framework that explicitly models SQL errors via LLMs. We adopt their official settings to perform fine-grained error detection and subsequent refinement.

Evaluation Metrics.

- **NL2SQL-Bugs**: We use Type-Specific Accuracy (TSA_t) to precisely evaluate the model's ability to identify specific error categories. It is defined as $\text{TSA}_t = \frac{TP_t}{TP_t+FP_t+FN_t}$, where TP_t, FP_t, FN_t represent the true positives, false positives, and false negatives for error type t, respectively. The reported "Overall" result is micro-averaged over all 2,018 test instances rather than a macro average over error types. Per-type entries should be interpreted together with support: rare types such as *Window Functions* and *Partial Query* have very small support, so their extreme values are less stable than the micro-averaged aggregate.
- **Spider & BIRD**: We use Execution Accuracy (EX) to evaluate the proportion of refined SQL queries that yield the exact same execution results as the ground truth queries.

C Prompt Design for SQL Error Diagnosis

To enable a large language model (LLM) to behave as a *SQL error analysis expert*, we design a structured two-part prompt consisting of (i) a **system message** that specifies the role, the taxonomy of error types, and the required output format, and (ii) a **user template** that instantiates a concrete diagnostic instance with the natural language question, the external evidence, the database schema, the candidate SQL query, and its execution result. The full prompt is shown in Listing 1 and Listing 2.

System message. The system message (Listing 1) instructs the LLM to assess whether a generated SQL query faithfully answers the user's question *given* the supplied evidence, schema, and execution result, and, if not, to identify the specific error categories. It enumerates a hierarchical taxonomy of nine top-level error categories covering attribute-, table-, value-, operator-, condition-, function-, clause-, subquery- and other miscellaneous errors, each with concrete sub-types. This taxonomy aligns the model's diagnostic vocabulary with the fine-grained error labels defined in our benchmark, and allows the diagnosis to be machine-parseable.

Output format. We constrain the model output to a JSON object: `{"label": "right"}` when the SQL is judged correct, and otherwise a structured object that contains a list of error instances. Each instance includes a free-form `reasoning` field (forcing the model to verbalize a justification before committing to a label, in the spirit of chain-of-thought prompting), as well as the categorical `error_type` and `sub_error_type` fields drawn from the taxonomy. This design makes the prediction directly comparable to the ground-truth annotation and supports per-category evaluation.

User template. The user template (Listing 2) packs the five inputs — *question*, *evidence*, *schema*, *generated SQL*, and *execution result* — into a single prompt slot. Including the execution result is critical: a syntactically valid SQL with a plausible-looking single-row result is the most common failure mode of LLM-based NL2SQL systems, and surfacing the actual returned rows (rather than only the SQL text) gives the diagnoser a chance to detect semantic mismatches that are invisible from the SQL alone.

Listing 1: System message used for SQL error diagnosis.

```
You are a SQL error analysis expert. Given a user question, evidence,
database schema, generated SQL query, and execution result, you need
to analyze whether the SQL correctly answers the question and identify
any error types if it's wrong.

### Error Types Description:
1. Attribute-Related Errors:
   - Attribute Mismatch: Wrong attribute selection
   - Attribute Redundancy: Unnecessary attributes included
```

```
   - Attribute Missing: Required attributes not included

2. Table-Related Errors:
   - Table Mismatch: Wrong table selection
   - Table Redundancy: Unnecessary tables included
   - Table Missing: Required tables not included
   - Join Condition Mismatch: The join condition between tables is incorrect
   - Join Type Mismatch: Wrong join type (e.g., LEFT JOIN when INNER JOIN needed)

3. Value-Related Errors:
   - Value Mismatch: Incorrect values in conditions
   - Data Format Mismatch: Wrong data type or format

4. Operator-Related Errors:
   - Comparison Operator Mismatch: Wrong operator (e.g., = when != needed)
   - Logical Operator Mismatch: Wrong logical operator (e.g., AND when OR needed)

5. Condition-Related Errors:
   - Explicit Condition Missing: Required conditions not included
   - Explicit Condition Mismatch: Wrong conditions
   - Explicit Condition Redundancy: Unnecessary conditions
   - Implicit Condition Missing: Missing implicit conditions

6. Function-Related Errors:
   - Aggregate Functions: Incorrect use of SUM, AVG, etc.
   - Window Functions: Incorrect use of OVER, PARTITION BY, etc.
   - Date/Time Functions: Incorrect use of JULIANDAY, strftime, etc.
   - Conversion Functions: Incorrect use of CAST, etc.
   - Math Functions: Incorrect use of ROUND, etc.
   - String Functions: Incorrect use of SUBSTR, etc.
   - Conditional Functions: Incorrect use of IIF, CASE WHEN, etc.

7. Clause-Related Errors:
   - Clause Missing: Required clauses not included
   - Clause Redundancy: Unnecessary clauses included

8. Subquery-Related Errors:
   - Subquery Missing: Required subquery not included
   - Subquery Mismatch: Incorrect subquery logic
   - Partial Query: The error SQL query is a partial query

9. Other Errors:
   - ASC/DESC: Incorrect sorting order
   - DISTINCT: Incorrect use of DISTINCT
   - Other: Major logical errors requiring complete rewrite

### Output Format:
You must output your analysis in the following JSON format:
- If the SQL is correct, output: {"label": "right"}
- If the SQL is wrong, output:
  {
      "label": "wrong",
      "error_types": [
          {
              "reasoning": "explanation of why this error type applies",
              "error_type": "error category from the above list",
              "sub_error_type": "specific error type from the above list"
          }
      ]
  }
```

Listing 2: User template that instantiates a single diagnostic instance.

```
Question: {question}
Evidence: {evidence}
Database Schema:
{schema}
Generated SQL: {sql}
Execution Result: {execution_result}

Please analyze whether this SQL query correctly answers the question
based on the provided evidence, database schema, and execution result.
If the SQL is incorrect, identify the specific error types.
```

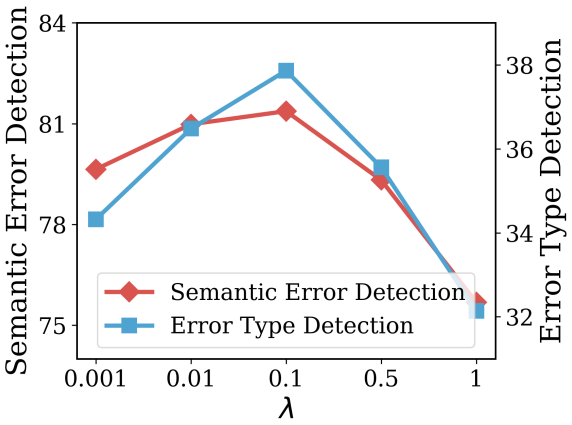


Figure 5: Sensitivity of DIAGSQL (QWEN3-8B) to the structural-bias coefficient λ in CORS, evaluated on NL2SQL-Bugs. The left axis (red) reports Semantic Error Detection accuracy and the right axis (blue) reports Error Type Detection accuracy. Both metrics peak at $\lambda = 0.1$, drop mildly for under-shaped settings ($\lambda \in \{0.001, 0.01\}$), and degrade sharply when the structural prior is over-weighted ($\lambda \to 1$).

D Hyperparameter Sensitivity

In this section we analyze the sensitivity of DIAGSQL to the trade-off coefficient λ introduced in Section 3.3 (`Co-occurrence-aware Reward Shaping`), which controls the strength of the structural inductive bias injected by CORS relative to the task reward. All experiments use QWEN3-8B as the backbone LLM and are evaluated on NL2SQL-Bugs along the two complementary axes used in the main paper—coarse-grained *Semantic Error Detection* and fine-grained *Error Type Detection*. We sweep $\lambda \in \{0.001, 0.01, 0.1, 0.5, 1.0\}$ on a logarithmic grid, retraining DIAGSQL from the same SFT-initialized checkpoint for each value while keeping every other hyperparameter fixed; the resulting accuracies are plotted in Figure 5.

The curves exhibit a clear, well-behaved unimodal pattern. As λ grows from 0.001 to 0.1, both metrics improve monotonically: Semantic Error Detection rises from $\sim$79.7% to its peak of $\sim$81.5%, and Error Type Detection rises more steeply from $\sim$34.4% to its peak of $\sim$38.0%. This monotonic improvement in the small-λ regime indicates that the co-occurrence prior is genuinely informative—when its signal is too weak, the policy effectively reduces to vanilla FTRA and loses the structural guidance needed for correlated multi-label errors. The fact that Error Type Detection benefits more than Semantic Error Detection is also expected, since fine-grained categorization is precisely the task for which structural co-occurrence priors carry the most discriminative information.

Beyond the optimum at $\lambda = 0.1$, however, performance deteriorates rapidly: at $\lambda = 0.5$ both metrics give back roughly half of the gains, and at $\lambda = 1.0$ Semantic Error Detection collapses to $\sim$75.5% and Error Type Detection to $\sim$32.2%—worse than the under-shaped $\lambda = 0.001$ setting. The interpretation is that an over-weighted λ allows CORS to dominate the task reward, encouraging the policy to over-commit to high-lift error pairs even when the underlying SQL evidence is weak, which directly harms precision on both detection tasks. The asymmetric, fast-falling right tail of the curve thus mirrors the qualitative *hallucinated co-occurrence* failure mode discussed in Section 4, only here it is induced by an over-strong prior rather than by missing credit assignment.

Overall, DIAGSQL is robust within roughly an order of magnitude around the optimum—any $\lambda \in [0.01, 0.1]$ yields near-peak performance on both metrics—and we adopt $\lambda = 0.1$ as the default in all main experiments. The sweep also offers a practical guideline for transferring DIAGSQL to new SQL-error taxonomies: λ should be set just large enough for the structural prior to take effect but kept well below the task reward magnitude so that the validator still grounds its diagnoses in the actual SQL semantics.

Table 4: Overall EX (%, mean ± std.; five independent runs).

Generator	Validator	BIRD	Spider
GPT-4o	ErrorLLM (GPT-4o)	66.08 ± 0.44	86.79 ± 0.42
	DiagSQL (Llama3.1-8B)	67.88 ± 0.46	87.19 ± 0.35
	DiagSQL (Qwen3-8B)	$\mathbf{68.47 \pm 0.41}$	86.61 ± 0.72
OpenSearch-SQL	ErrorLLM (GPT-4o)	72.13 ± 0.44	86.88 ± 0.38
	DiagSQL (Llama3.1-8B)	72.00 ± 0.43	$\mathbf{86.91 \pm 0.44}$
	DiagSQL (Qwen3-8B)	$\mathbf{72.78 \pm 0.36}$	86.81 ± 0.41

Table 5: Overall EX (%) with a noise-perturbed, top-K co-occurrence prior.

Method	BIRD	Spider
DiagSQL$^+$ (Llama3.1-8B)	65.04	85.11
DiagSQL (Llama3.1-8B)	67.73	87.04
DiagSQL$^+$ (Qwen3-8B)	65.71	86.02
DiagSQL (Qwen3-8B)	68.32	86.46

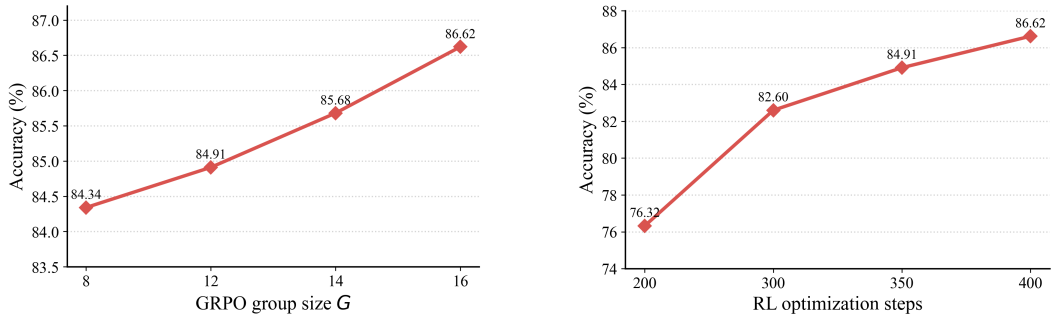
E Additional Robustness and Generalization Analyses

Stability across independent runs. We repeat the strongest ErrorLLM [7] and DiagSQL configurations with five independent seeds and report mean ± standard deviation of overall execution accuracy (EX) in Table 4. Across both upstream generators and both benchmarks, the standard deviations remain below one percentage point, indicating stable optimization. With GPT-4o as the generator, DiagSQL (Qwen3-8B) improves BIRD from 66.08 ± 0.44 to 68.47 ± 0.41, despite using an 8B validator rather than the substantially larger model used by ErrorLLM. On the other settings, DiagSQL remains competitive while retaining this model-scale advantage.

Robustness and transferability of the co-occurrence prior. We stress-test CORS by adding random noise to M and retaining only the top-K highest-confidence error pairs; this confidence-pruned variant is denoted DiagSQL$^+$. Although the prior is deliberately degraded, Table 5 shows that DiagSQL$^+$ preserves most of the full model's EX, suggesting that CORS relies more on the strongest dependency topology than on exact co-occurrence magnitudes. This conclusion is consistent with DiagSQL* in the main results, which discards magnitudes and retains only the sign of each dependency. Because NL2SQL-Bugs is annotated from BIRD-dev, applying its prior to BIRD is in-distribution; Spider is the genuine cross-dataset transfer test. The preserved Spider performance therefore provides direct evidence that the useful structure transfers beyond the source benchmark. This behavior is also mechanistically plausible: dependencies such as *Table Missing→Join Condition Mismatch* arise from SQL semantics rather than benchmark-specific vocabulary.

Group-size and training-step sensitivity. Figure 6 isolates two RL training-scale factors for DiagSQL (Qwen3-8B) on NL2SQL-Bugs. Larger rollout groups improve accuracy from 84.34% at $G = 8$ to 86.62% at $G = 16$ (Figure 6a), because they provide a more informative relative-reward distribution, albeit at higher memory and sampling cost. Extending RL optimization from 200 to 400 steps yields a steady gain from 76.32% to 86.62% (Figure 6b). Together, the two monotonic trends indicate predictable scaling rather than sensitivity to a narrow RL training configuration.

Execution feedback is complementary. Figure 7 compares structural diagnosis alone with execution feedback followed by DiagSQL. The combined pipeline consistently improves overall EX: BIRD/Spider rise from $67.73/87.04$ to $67.86/87.33$ with Llama3.1-8B, and from $68.32/86.46$ to $68.64/87.23$ with Qwen3-8B. The gains are modest because execution primarily reveals failures such as missing tables or columns that DiagSQL can already diagnose from SQL structure. Execution signals are therefore complementary, but most of their detectable error coverage is already captured by the structural validator.



(a) GRPO group size.

(b) RL optimization steps.

Figure 6: RL training-scale sensitivity of DIAGSQL (QWEN3-8B) on NL2SQL-Bugs. Each sweep changes one factor while holding the remaining training configuration fixed.

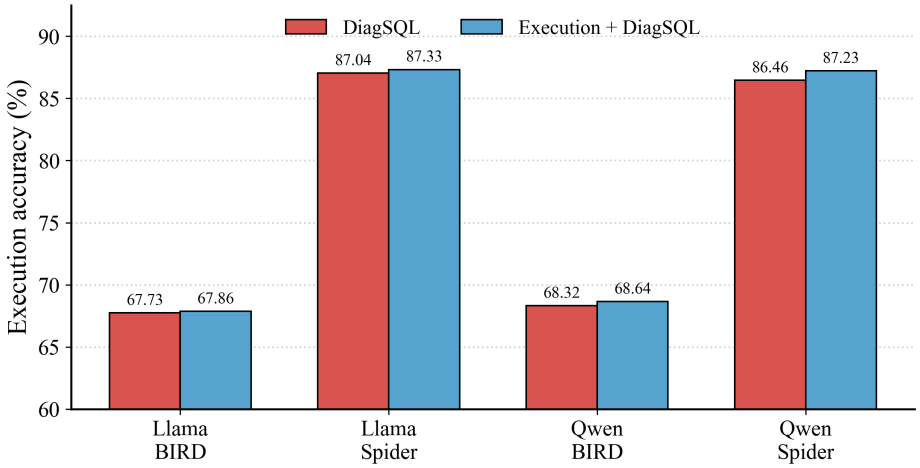


Figure 7: Overall EX on BIRD and Spider with DIAGSQL alone and with execution feedback added before diagnosis.

Complex databases and modern generators. We next attach DIAGSQL to two stronger SQL generators, DeepEye-SQL [46] and APEX-SQL [47]. As Figure 8 shows, DIAGSQL improves every BIRD difficulty tier: DeepEye-SQL rises from 80.4/67.5/59.0 to 81.1/68.6/59.7 and APEX-SQL from 75.9/64.4/57.2 to 78.7/66.5/58.3 on simple/moderate/challenging questions. Spider 2.0 [48] introduces a different obstacle: enterprise schemas can exceed the context budget of the 8B validator. We therefore use DBCC [49] to compress the database context before generation and diagnosis. For DeepEye-SQL, EX increases from 50.5 to 50.8 with DBCC and to 51.7 with DBCC plus DIAGSQL; for APEX-SQL, DBCC obtains 52.8 versus the 53.0 base result, while adding DIAGSQL raises EX to 54.1. Thus compression addresses context feasibility, whereas DIAGSQL contributes an orthogonal diagnostic signal that improves the final SQL.

F DIAGSQL vs. GPT-4O on NL2SQL-Bugs: A Detailed Head-to-Head

To complement the aggregate results in Section 4, we provide a fine-grained head-to-head comparison between our **DIAGSQL** (QWEN3-8B, pass@1) and **GPT-4O**—a strong, widely-used closed-source reference—on the NL2SQL-Bugs benchmark. All 2,018 test instances are evaluated under identical zero-guide prompts, yielding 2,016 parsable DIAGSQL predictions and 1,990 parsable GPT-4O predictions. We analyze four complementary views: (i) binary *wrong*-vs-*right* detection, (ii) aggregate multi-label classification over the 9 coarse categories and 32 fine-grained sub-types, (iii) per-class F1 differentials, and (iv) per-sample paired Jaccard similarity. All metrics in this section are computed at

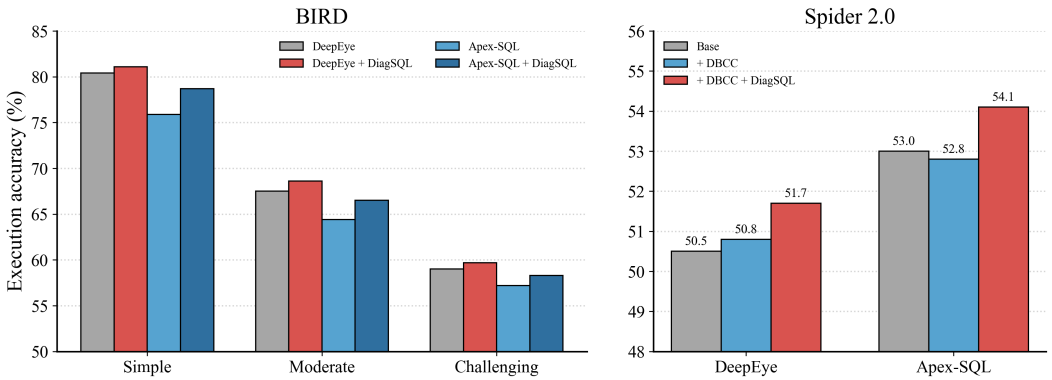


Figure 8: DIAGSQL with modern generators. Left: EX by BIRD difficulty. Right: overall EX on Spider 2.0, where DBCC provides database-context compression.

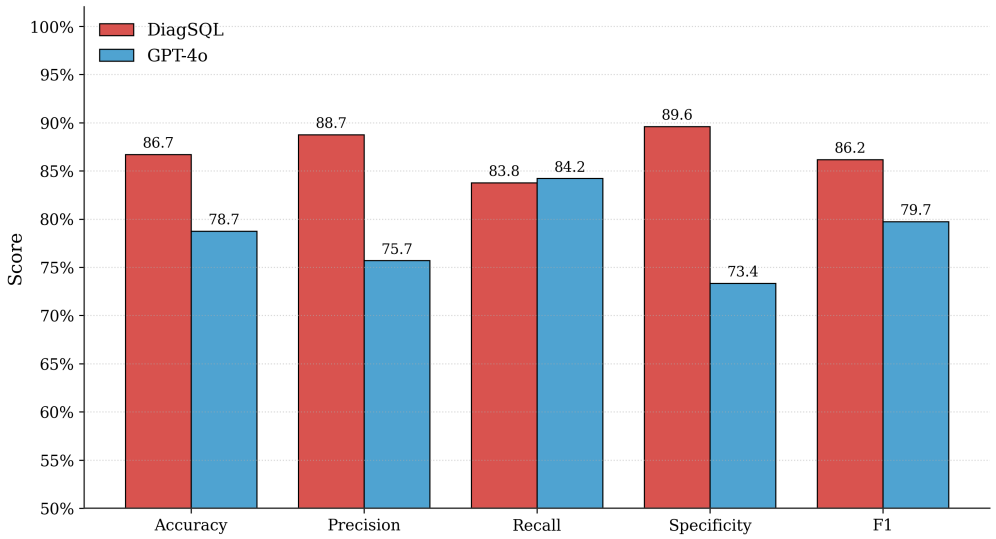


Figure 9: Binary bug-detection performance on NL2SQL-Bugs (positive class = "wrong SQL"). DIAGSQL matches the recall of GPT-4O while lifting precision and specificity by +13.0 and +16.2 percentage points, respectively.

the F1/Jaccard/exact-match level rather than as accuracies, and therefore complement (rather than replace) the accuracy numbers reported in Section 4.

Binary bug detection (Fig. 9). DIAGSQL reaches **86.7%** accuracy and **F1 = 86.2%**, compared with 78.7% / 79.7% for GPT-4O. The gain decomposes into two complementary effects. First, DIAGSQL is substantially more *conservative on correct SQL*: specificity rises from 73.4% to **89.6%** (+16.2%), reducing false alarms from 267 to 106 (−60.3%). Second, this collapse in false positives lifts precision from 75.7% to **88.7%** (+13.0%), while recall is essentially preserved (83.8% vs. 84.2%). In other words, DIAGSQL flags virtually every buggy SQL that GPT-4O flags, but suppresses GPT-4O's tendency to over-trigger on valid SQL—exactly the property required when the validator is deployed as a quality gate inside the closed-loop refinement pipeline of Section 4.

Multi-label error-type classification (Fig. 10). On the 32 sub-error-type classes, DIAGSQL attains Micro-F1 **36.3%**, Macro-F1 **23.8%**, and mean sample-level Jaccard **0.268**, against 31.0% / 16.4% / 0.230 for GPT-4O. The +7.4% gap on Macro-F1 (a 45% relative improvement) is particularly informative: because Macro-F1 weights every sub-type equally, this gap shows that DIAGSQL does not merely improve on head classes but also recovers a non-trivial share of the long-tail error types that GPT-4O misses entirely. The same trend holds at the 9-class coarse granularity (Micro-F1

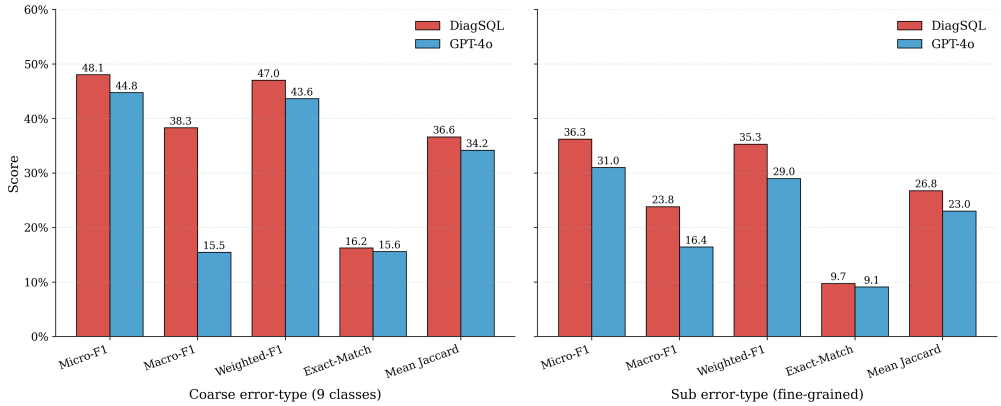


Figure 10: Multi-label error-type classification. Left: 9 coarse categories. Right: 32 fine-grained sub-types. DIAGSQL strictly outperforms GPT-4O on every aggregate metric, with the largest relative gains on Macro-F1 and mean Jaccard.

48.1%, Macro-F1 38.3%, mean Jaccard 0.366), where DIAGSQL strictly dominates GPT-4O on all five reported metrics. Exact-match accuracy—requiring the model to predict the *entire* error-type set of a sample correctly—also rises from 9.1% to 9.7% at the sub-type level and from 12.9% to 16.2% at the coarse level, indicating that the gain in set-level coherence is real rather than an artifact of any single label.

Where does DIAGSQL improve the most? (Fig. 11). Decomposing the gain by class, DIAGSQL wins on **22 of the 31** non-trivial sub-types (71%). The largest absolute improvements occur on structurally hard errors that demand schema-aware, multi-step reasoning—*conditional functions* (+31.3%), *attribute mismatch* (+20.8%), *table redundancy* (+14.0%), *logical operator mismatch* (+13.5%), and *value mismatch* (+10.6%)—precisely the categories where Section 4 also reports the largest per-category gains and where Section 4 identifies the strongest co-occurrence couplings. The nine sub-types on which GPT-4O is ahead are either extremely rare (support ≤ 10) or pathological cases in which GPT-4O predicts labels (e.g., *join condition missing*, *clause mismatch*) that essentially never occur in the ground truth—the same *hallucinated co-occurrence* failure mode that Section 4 visualizes via long blue arcs. The mean ΔF1 on these losing sub-types is only -3.9%, an order of magnitude smaller than the $+12\%$ average gain on the winning sub-types, confirming that DIAGSQL's advantage is broad-based rather than concentrated on a few dominant classes.

Per-sample head-to-head analysis (Fig. 12). To complement the aggregate view, we pair the two models on every buggy SQL case and compute the per-sample Jaccard similarity between the predicted and ground-truth sub-error-type sets. Out of 986 paired buggy cases, DIAGSQL *strictly beats* GPT-4O on 309 (31.3%), the two models *tie* on 417 (42.3%, predominantly when both predict identical sets or both miss the same labels), and GPT-4O wins on only 260 (26.4%). Excluding ties, this corresponds to a 54.3% **win rate for DIAGSQL**. The ΔJaccard distribution is right-skewed with a mean of $+0.035$ and a heavy positive tail, confirming that DIAGSQL's paired advantage is systematic rather than driven by a handful of outliers.

Take-away. Across all four views—binary detection (Fig. 9), aggregate multi-label metrics (Fig. 10), per-class F1 differentials (Fig. 11), and per-sample paired Jaccard (Fig. 12)—DIAGSQL is consistently the stronger NL2SQL bug diagnoser. Two structural advantages stand out: (i) a substantially lower false-alarm rate, which is what makes DIAGSQL safe to wire into the closed-loop Text-to-SQL refinement pipeline; and (ii) broadly higher F1 across the long tail of fine-grained error types, which is the direct downstream manifestation of the FTRA token-level credit assignment and the GT-aligned co-occurrence structure visualized in Figure 4. Together, these results show that the gains DIAGSQL achieves over GPT-4O are not a metric-specific artifact but a coherent, mechanism-level upgrade in how the validator localizes and combines fine-grained SQL errors.

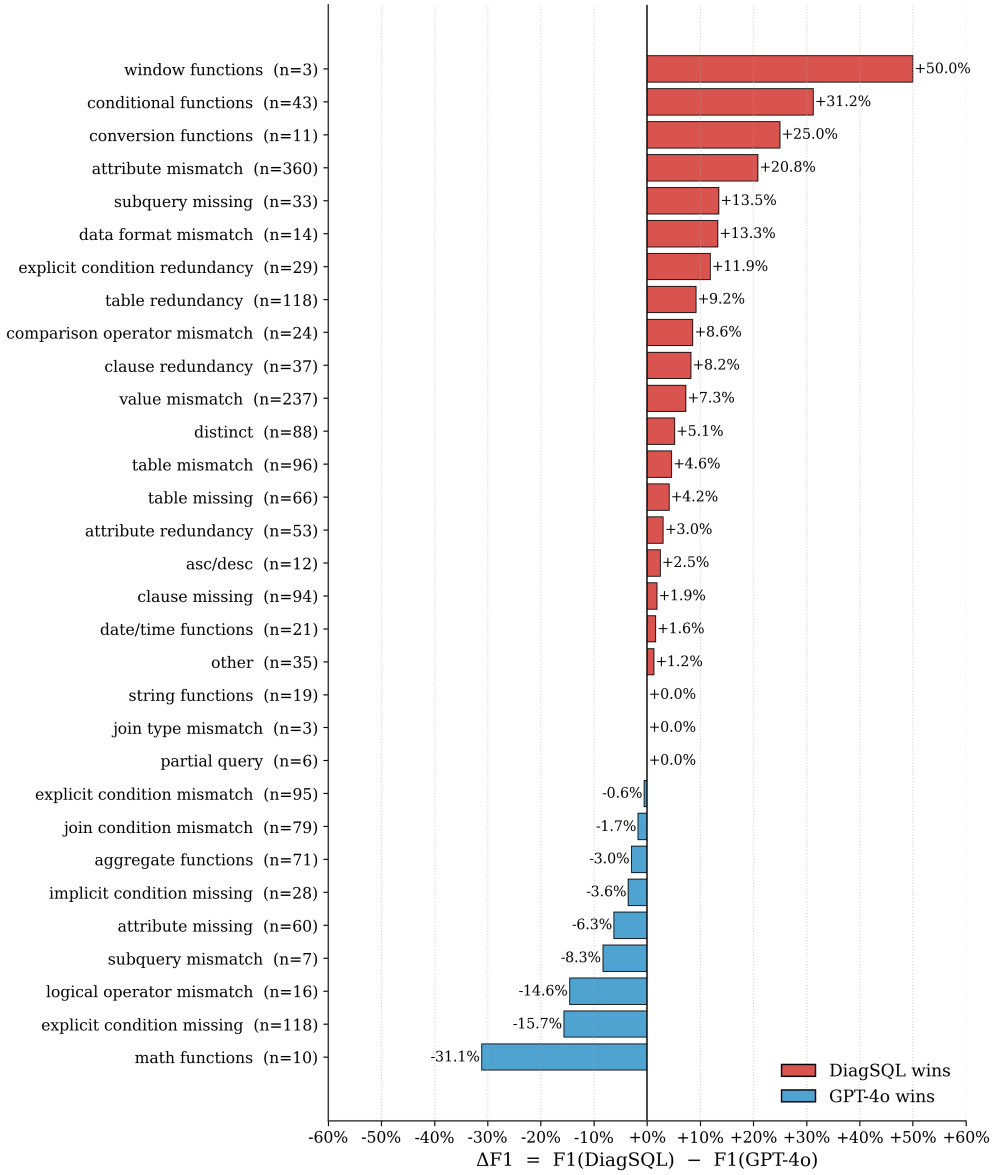


Figure 11: Per sub-error-type ΔF1 = F1(DiagSQL) $-$ F1(GPT-4o) (diverging bars, sorted). Green bars mark sub-types where DIAGSQL wins; purple bars mark those where GPT-4O wins. DIAGSQL dominates GPT-4O on 22 of 31 sub-types with non-zero support.

G Multi-Round Analysis of Text-to-SQL Refinement

We study how the closed-loop refinement of Section 4 behaves when the refinement budget K is extended beyond the single pass used in the main paper, with GPT-4O as the SQL generator, QWEN3-8B as the DIAGSQL backbone, and BIRD as the benchmark. We compare DIAGSQL-driven refinement against GPT-4O *Self-Correction*, where the generator critiques and rewrites its own output without an external validator. Round 0 is the GPT-4O single-shot result (55.87% EX), and round 1 corresponds to the numbers reported in Table 2 of Section 4; each subsequent round feeds the previous SQL plus the validator's feedback back into the generator.

As shown in Figure 13, DIAGSQL jumps from 55.87 to $\sim$68.3 at round 1 and peaks at $\sim$68.6 at round 2, then mildly drops to $\sim$66.9 and $\sim$64.5 at rounds 3–4. The early gains come from DIAGSQL's precise, GT-aligned diagnoses (Sections 4 and F), which let the generator fix most genuine bugs in

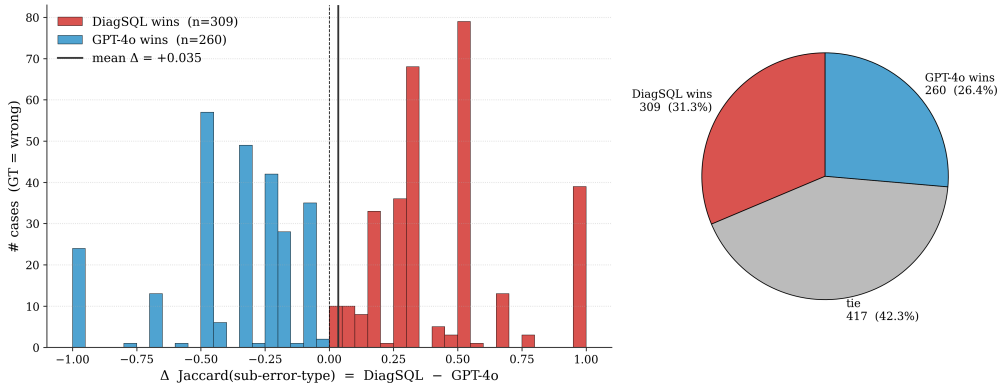


Figure 12: Sample-level head-to-head comparison on the 986 buggy cases. Left: distribution of ΔJaccard $=$ Jaccard(DiagSQL) $-$ Jaccard(GPT-4o) on the sub-error-type label sets. Right: win/tie/loss breakdown. The mean ΔJaccard is $+0.035$ and DIAGSQL wins on 54.3% of non-tied cases.

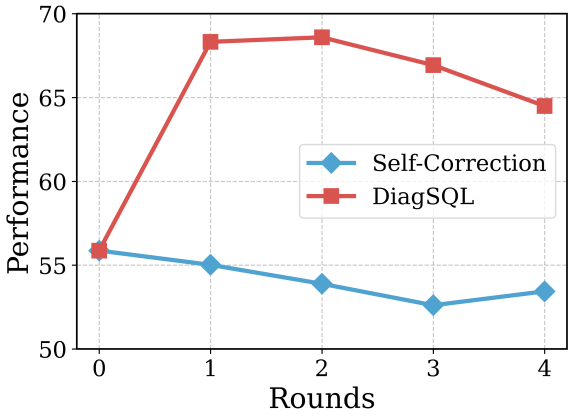


Figure 13: Multi-round refinement on BIRD. Round 1 reproduces the single-pass result used in Table 2; DIAGSQL peaks at round 2 and then mildly regresses, whereas GPT-4O Self-Correction degrades monotonically from round 1 onward.

one or two passes; the late-round dip is an over-correction effect, since once the truly buggy SQLs are repaired the remaining queries are mostly already correct, and DIAGSQL's residual false-positive rate (about 10%, Section F) gradually pushes a small fraction of clean SQLs into harmful rewrites.

GPT-4O Self-Correction, in contrast, never improves and instead drops to $\sim$55.0/53.9/52.6/53.4 across rounds 1–4. Asking the same generator to also judge its own SQL recycles its biases—low specificity (73.4%, Section F) and hallucinated co-occurrence patterns (Section 4)—so each round flips many correct SQLs into wrong rewrites while only weakly fixing genuinely buggy ones. This open-loop bottleneck is exactly what an externally-trained, GT-aligned validator like DIAGSQL is designed to break.

More broadly, any verifier-driven refinement loop will eventually saturate: as rounds progress, the pool of remaining buggy SQLs shrinks while the pool of already-correct SQLs grows, so the validator's false positives gradually outweigh its true positives. Where this saturation point lies depends mainly on the validator's specificity (DIAGSQL's 89.6% vs. GPT-4O's 73.4% explains why DIAGSQL still gains in early rounds while Self-Correction never does), the generator's intrinsic capacity on hard schema-heavy queries, and the diversity of feedback across rounds—repeated diagnoses quickly drive the generator into a fixed point. In our setup the round-1 gain already accounts for the large majority of the total attainable improvement (from 55.87 to $\sim$68.3, versus an additional $+0.3$ at round 2), and the curve flattens almost immediately after that. Accordingly, we adopt $K{=}1$ (a single refinement pass) as the default budget for all main BIRD/Spider results reported in Section 4, since it delivers

essentially the full benefit of closed-loop refinement while keeping inference cost minimal; we use the additional rounds here only as a saturation diagnostic rather than as an operating point.

H Case Study

To probe the practical advantage of DIAGSQL, we present four qualitative cases that target four distinct failure modes a SQL error diagnoser can exhibit. All four cases are drawn from the NL2SQL-BUGS benchmark and share the prompt template defined in Section C (Listing 1 and Listing 2); methods only differ in the underlying diagnoser. For brevity, the schema slot of the user template is shown by an excerpt of the relevant tables; full schemas can be reproduced from the prompt in Section C. The four "downstream" fixes are produced by feeding each diagnoser's output back into a fixed SQL re-generator—specifically GPT-4O invoked with the same refinement prompt used for the GPT-4O block of Table 2, matching the Table 2 refinement pipeline exactly—so the only varying factor across methods within each case is the diagnostic signal itself.

The four cases are designed to answer four orthogonal questions:

- **Case H.1.** Can DIAGSQL catch a *very* subtle semantic error that even GPT-4O waves through as correct?
- **Case H.2.** How much does the `reasoning` field in our output schema actually contribute to downstream repair, beyond the categorical label?
- **Case H.3.** When both methods agree the SQL is wrong, does the *quality* of error categories and reasoning matter for repair?
- **Case H.4.** Does *co-occurrence shaping* let DIAGSQL recover a complete error set that a non-shaping variant misses?

H.1 Case 1: Detecting a Subtle Error That GPT-4O Misjudges

Case 1 input (instantiating the slots of Listing 2)

Question. *What is the average score in writing for the schools that were opened after 1991 or closed before 2000? List the school names along with the score. Also, list the communication number of the schools if there is any.*

Evidence. *Communication number refers to phone number.*

Schema (excerpt).

```
CREATE TABLE `schools` (
    CDSCode TEXT NOT NULL,
    School TEXT, Phone TEXT,
    OpenDate DATE, ClosedDate DATE, ...
    PRIMARY KEY (CDSCode));
CREATE TABLE `satscores` (
    cds TEXT NOT NULL,
    AvgScrWrite INTEGER, ...
    PRIMARY KEY (cds),
    FOREIGN KEY (cds) REFERENCES `schools`.`CDSCode`);
```

Generated SQL (under diagnosis).

```
SELECT T1.School, T2.AvgScrWrite, T1.Phone
FROM schools AS T1 INNER JOIN satscores AS T2
    ON T1.CDSCode = T2.cds
WHERE strftime('%Y', T1.OpenDate) > '1991'
   OR strftime('%Y', T1.ClosedDate) < '2000';
```

Exec. `success=true, 100 rows; first row: ('Crescent View West Charter', NULL, '(559) 222-8439').`

GPT-4O output.

GPT-4O (Case 1)

```
{"label": "right"}
```

DIAGSQL output.

DIAGSQL (Case 1)

```
{
  "label": "wrong",
  "error_types": [
    {
      "reasoning": "The question requires listing the phone of every
        qualifying school 'if there is any', which implies retaining
        schools that have no satscores row. INNER JOIN drops such
        schools, so the join type is wrong; LEFT JOIN should be used.",
      "error_type": "Table-Related Errors",
      "sub_error_type": "Join Type Mismatch"
    }
  ]
}
```

Analysis. GPT-4O is misled by three superficial cues: the SQL is syntactically valid, the result has the expected columns, and both date conditions cite the entities mentioned in the question. It fails the only piece of reasoning that distinguishes correct from incorrect—namely, that "*if there is any*" is a soft existential hint and therefore mandates a left-outer join. DIAGSQL not only flips the binary label to `wrong` but also pinpoints the exact sub-type *Join Type Mismatch*, matching the gold annotation $1/1$. This is the canonical case where GPT-4O produces a *false negative*: a downstream pipeline that trusts the verifier would silently ship an incorrect SQL.

H.2 Case 2: Reasoning Drives Successful Repair, Even When Categories Are Identical

Case 2 input (instantiating the slots of Listing 2)

Question. *Which different county has the most number of closed schools? Please provide the name of each school as well as the closure date.*

Evidence. *Closure date and closed date are synonyms; 'Closed' was mentioned in schools.StatusType.*

Schema (excerpt).

```
CREATE TABLE `schools` (
    CDSCode TEXT NOT NULL, County TEXT NOT NULL,
    School TEXT, ClosedDate DATE,
    StatusType TEXT NOT NULL, ...
    PRIMARY KEY (CDSCode));
```

Generated SQL (under diagnosis).

```
SELECT County, School, ClosedDate
FROM schools
WHERE StatusType = 'Closed'
GROUP BY County, School, ClosedDate
ORDER BY count(School) DESC LIMIT 1;
```

Exec. `success=true, 1 row; ('Los Angeles', 'Lincoln Elementary', '1989-06-30').`

Both diagnosers agree on the category. Both DIAGSQL and the ablated variant DIAGSQL (w/o reasoning) correctly flag this SQL as `wrong` and predict the same sub-type *Implicit Condition Missing* (matching the gold annotation $1/1$ on category). The two diagnostic outputs differ only in whether they verbalize the reasoning chain.

DIAGSQL (w/o reasoning) (Case 2)

```
{
  "label": "wrong",
  "error_types": [
    {"error_type": "Condition-Related Errors",
     "sub_error_type": "Implicit Condition Missing"}
  ]
}
```

DIAGSQL (Case 2)

```
{
  "label": "wrong",
  "error_types": [
    {
      "reasoning": "The question asks for THE county with the most
        closed schools, then enumerate every closed school in that
        county. The current GROUP BY (County, School, ClosedDate)
        produces one row per (county,school) so ORDER BY count(School)
        is always 1, making LIMIT 1 return an arbitrary row. The
        implicit two-step structure --- first pick the county whose
        COUNT is the largest, then list its schools --- is missing.
        A nested subquery selecting that county is required, plus a
        non-null guard on School to avoid empty rows.",
      "error_type": "Condition-Related Errors",
      "sub_error_type": "Implicit Condition Missing"
    }
  ]
}
```

Downstream repair attempts. We feed each diagnostic output back into the same SQL re-generator.

Re-generation guided by DIAGSQL (w/o reasoning). Knowing only that some implicit condition is missing, the re-generator adds a non-null guard on `School`, but does not restructure the top-level query:

```
-- Still wrong: returns one (county,school) pair, not the whole county.
SELECT County, School, ClosedDate
FROM schools
WHERE StatusType = 'Closed' AND School IS NOT NULL
GROUP BY County, School, ClosedDate
ORDER BY COUNT(School) DESC LIMIT 1;
```

Re-generation guided by DIAGSQL. Given the explicit two-step reasoning, the re-generator produces:

```
-- Correct: matches the gold SQL.
SELECT DISTINCT County, School, ClosedDate
FROM schools
WHERE County = (
        SELECT County FROM schools
        WHERE StatusType = 'Closed'
        GROUP BY County
        ORDER BY COUNT(School) DESC LIMIT 1)
  AND StatusType = 'Closed' AND School IS NOT NULL;
```

Analysis. This case isolates the contribution of the `reasoning` field. Both diagnosers reach the same categorical verdict, yet only the reasoning-equipped variant transmits the *actionable structure* of the missing implicit condition ("select the top county first, then enumerate") to the re-generator. The bare label *Implicit Condition Missing* is a search space far too large for an LLM-based re-generator to navigate without further guidance, and the ablated variant's repair therefore inherits the same structural defect.

H.3 Case 3: When Both Methods Flag It Wrong, Reasoning Quality Decides Repair

Case 3 input (instantiating the slots of Listing 2)

Question. *Which district has highest active loan?*

Evidence. *A3 refers to district names; Active loan refers to running contracts; Status = 'C' stands for running contract, OK so far; Status = 'D' stands for running contract, client in debt.*

Schema (excerpt).

```
CREATE TABLE `district` (district_id INT, A2 TEXT, A3 TEXT, ...);
CREATE TABLE `account`  (account_id INT, district_id INT, ...);
CREATE TABLE `loan` (loan_id INT, account_id INT, amount INT,
                     status TEXT, ...);
```

Generated SQL (under diagnosis).

```
SELECT T3.A2
FROM loan AS T1
```

```sql
INNER JOIN account AS T2 ON T1.account_id = T2.account_id
INNER JOIN district AS T3 ON T2.district_id = T3.district_id
WHERE T1.status = 'C' OR T1.status = 'D'
GROUP BY T3.A2
ORDER BY COUNT(T1.loan_id) DESC LIMIT 1;
```

Exec. `success=true`, 1 row; (‘Hl.m. Praha’).

Gold error categories (4 in total). *Attribute Mismatch* (`A2` → `A3`), *Table Mismatch* (the projection should come from `A3`, not `A2`, but more importantly the join order is wrong), *Comparison Operator Mismatch* (`= 'C' OR = 'D'` is a roundabout form of an `IN` membership test the rest of the benchmark expects), and *Clause Missing* (the ranking criterion should be `SUM(amount)`, not `COUNT(loan_id)`).

GPT-4o output (categories partially wrong, reasoning misleading).

GPT-4o (Case 3)

```
{
  "label": "wrong",
  "error_types": [
    {
      "reasoning": "The question says 'highest active loan',
        which most likely means the largest single loan amount,
        so the query should ORDER BY T1.amount instead of
        COUNT(T1.loan_id).",
      "error_type": "Clause-Related Errors",
      "sub_error_type": "Clause Missing"
    },
    {
      "reasoning": "Status condition is fine; only the aggregation
        target is wrong.",
      "error_type": "Other Errors",
      "sub_error_type": "Other"
    }
  ]
}
```

DIAGSQL output.

DIAGSQL (Case 3)

```
{
  "label": "wrong",
  "error_types": [
    {
      "reasoning": "Evidence states 'A3 refers to district names';
        the query selects A2 instead. Replace A2 with A3 in both
        SELECT and GROUP BY.",
      "error_type": "Attribute-Related Errors",
      "sub_error_type": "Attribute Mismatch"
    },
    {
      "reasoning": "The intended driving table is ‘account‘, not
        ‘loan‘; the gold pattern joins account first and then loan.
        The current alias chain T1=loan -> T2=account -> T3=district
        does not match the schema's natural FK direction.",
      "error_type": "Table-Related Errors",
      "sub_error_type": "Table Mismatch"
    },
    {
      "reasoning": "status='C' OR status='D' should be expressed as
        status IN ('C','D'); the disjunction-of-equalities form is a
        comparison-operator mismatch w.r.t. the canonical pattern.",
      "error_type": "Operator-Related Errors",
      "sub_error_type": "Comparison Operator Mismatch"
    },
    {
      "reasoning": "'Highest active loan' means the largest TOTAL of
        active-loan amounts per district, so the ORDER BY should be
        SUM(loan.amount), not COUNT(loan_id). The aggregate clause
        SUM(...) is missing.",
      "error_type": "Clause-Related Errors",
      "sub_error_type": "Clause Missing"
    }
```

```
  ]
}
```

Downstream repair. *Re-generation guided by* GPT-4O. Following GPT-4O's "largest single loan" interpretation, the re-generator drops the aggregation entirely:

```
-- Still wrong: returns the district of one biggest loan,
-- not the district with the largest TOTAL active-loan volume.
SELECT T3.A2
FROM loan AS T1 INNER JOIN account AS T2
      ON T1.account_id = T2.account_id
INNER JOIN district AS T3
      ON T2.district_id = T3.district_id
WHERE T1.status IN ('C','D')
ORDER BY T1.amount DESC LIMIT 1;
```

Re-generation guided by DIAGSQL.

```
-- Correct: matches the gold SQL semantics.
SELECT T2.A3
FROM account AS T1
INNER JOIN district AS T2 ON T1.district_id = T2.district_id
INNER JOIN loan     AS T3 ON T1.account_id  = T3.account_id
WHERE T3.status IN ('C','D')
GROUP BY T2.A3
ORDER BY SUM(T3.amount) DESC LIMIT 1;
```

Analysis. GPT-4O catches that something is off but commits to a wrong *interpretation* ("highest active loan = single largest loan amount"). Its reasoning then misleads the re-generator into a fix that is internally consistent but semantically wrong. DIAGSQL covers all four gold sub-types ($4/4$ vs $1/4$ for GPT-4O, where the "Other" label is a coarse miss) and, crucially, supplies the *right* reasoning for the aggregation step (`SUM(amount)` not `COUNT(loan_id)`), which the re-generator can act on faithfully.

H.4 Case 4: Co-occurrence Shaping Recovers a Complete Error Set

Case 4 input (instantiating the slots of Listing 2)

Question. *Rank schools by their average score in Writing where the score is greater than 499, showing their charter numbers.*

Evidence. *Valid charter number means the number is not null.*

Schema (excerpt).

```
CREATE TABLE `schools` (
    CDSCode TEXT NOT NULL, Charter INTEGER, CharterNum TEXT, ...);
CREATE TABLE `satscores` (
    cds TEXT NOT NULL, AvgScrWrite INTEGER, ...,
    FOREIGN KEY (cds) REFERENCES `schools`.`CDSCode`);
```

Generated SQL (under diagnosis).

```
SELECT schools.charter
FROM schools INNER JOIN satscores
     ON schools.cdscode = satscores.cds
WHERE satscores.avgscrwrite = 499;
```

Exec. `success=true`, 6 rows; first row: `(NULL,)`.

Gold error categories (4 in total, with strong co-occurrence). *Attribute Mismatch* (`schools.Charter` should be `schools.CharterNum`), *Explicit Condition Mismatch* (`= 499` should be `> 499`), *Implicit Condition Missing* (`CharterNum IS NOT NULL`, required by the evidence), and *Window Functions* (the question says "rank", so a `RANK() OVER (ORDER BY AvgScrWrite DESC)` is required). The four sub-types co-occur consistently in our training corpus on "rank … where …" patterns: missing *Window Functions* and missing the implicit non-null guard tend to appear together with the value-condition mismatches that misuse a strict equality against what should be a ranked range.

DIAGSQL (w/o Co-occurrence Shaping). The ablated diagnoser examines each potential category independently and only commits to those it is locally confident about. It correctly identifies the two "locally obvious" errors but misses the two that are typically inferred from co-occurrence:

DIAGSQL (w/o Co-occurrence Shaping) (Case 4)

```
{
  "label": "wrong",
  "error_types": [
    {
      "reasoning": "Charter is a 0/1 flag, while CharterNum is the
        textual charter number requested by the question.",
      "error_type": "Attribute-Related Errors",
      "sub_error_type": "Attribute Mismatch"
    },
    {
      "reasoning": "The question says 'greater than 499', not equal
        to 499; the comparison should be > 499.",
      "error_type": "Condition-Related Errors",
      "sub_error_type": "Explicit Condition Mismatch"
    }
  ]
}
```

DIAGSQL (full). Through CORS training, the policy has internalized the conditional co-occurrence structure among SQL errors as a statistical bias baked into its parameters, rather than as an explicit inference-time operation: once *Attribute Mismatch* on `Charter` $\rightarrow$ `CharterNum` is detected, the conditional likelihood of *Implicit Condition Missing* (`CharterNum IS NOT NULL`, because the evidence binds "valid charter number" to non-null) is elevated, and the keyword "rank" in the question similarly elevates the likelihood of *Window Functions*. Both thus cross the decision threshold during a single-pass forward inference and are emitted together.

DIAGSQL (Case 4)

```
{
  "label": "wrong",
  "error_types": [
    {"reasoning": "Charter is a 0/1 flag; CharterNum is the textual
        charter number that the question asks for.",
     "error_type": "Attribute-Related Errors",
     "sub_error_type": "Attribute Mismatch"},
    {"reasoning": "The question says 'greater than 499', not equal
        to 499; use > 499.",
     "error_type": "Condition-Related Errors",
     "sub_error_type": "Explicit Condition Mismatch"},
    {"reasoning": "Evidence binds 'valid charter number' to a
        non-null value; once the projection is changed to CharterNum,
        the implicit guard CharterNum IS NOT NULL becomes required.",
     "error_type": "Condition-Related Errors",
     "sub_error_type": "Implicit Condition Missing"},
    {"reasoning": "The verb 'rank' demands an explicit ordering
        function; a window function such as
        RANK() OVER (ORDER BY AvgScrWrite DESC) should accompany the
        projection.",
     "error_type": "Function-Related Errors",
     "sub_error_type": "Window Functions"}
  ]
}
```

Downstream repair. *Re-generation guided by* DIAGSQL *(w/o Co-occurrence Shaping).* The re-generator only fixes the two reported errors; the result remains wrong because `NULL` charter numbers still appear and no ranking function is produced:

```
-- Still wrong: emits NULL CharterNum rows; no ranking column.
SELECT schools.CharterNum
FROM schools INNER JOIN satscores
     ON schools.cdscode = satscores.cds
WHERE satscores.avgscrwrite > 499;
```

Re-generation guided by DIAGSQL *(full).*

```
-- Correct: matches the gold SQL.
SELECT CharterNum, AvgScrWrite,
      RANK() OVER (ORDER BY AvgScrWrite DESC) AS WritingScoreRank
```

```
FROM schools AS T1
INNER JOIN satscores AS T2 ON T1.CDSCode = T2.cds
WHERE T2.AvgScrWrite > 499 AND CharterNum IS NOT NULL;
```

Analysis. The two "locally obvious" categories (*Attribute Mismatch*, *Explicit Condition Mismatch*) are detectable from individual SQL tokens and are recovered by both variants. The remaining two (*Implicit Condition Missing*, *Window Functions*) are *co-occurrence-driven* signals: they become much more probable *once* the first attribute substitution is identified or once the verb "rank" is interpreted in the context of an aggregating query. Co-occurrence shaping in DIAGSQL explicitly models these joint distributions and therefore recovers the full $4/4$ gold set, whereas the ablated variant stops at $2/4$ and the resulting partial fix is still incorrect.

H.5 Cross-Case Summary

Table 6 consolidates the four cases. Across *all* four diagnostic axes—binary correctness label, sub-type recall, downstream repair correctness, and source of advantage—the full DIAGSQL dominates every alternative we tested.

Table 6: Side-by-side comparison across the four cases. "Sub-types" counts the number of gold sub-types correctly recovered. "Repair OK" indicates whether a fixed SQL re-generator, when fed the diagnostic output, regenerates a SQL semantically equivalent to the gold.

Case	Method	Label	Sub-types	Repair OK
Case 1 (§H.1)	GPT-4o	`right` (✗)	0 / 1	✗
	DIAGSQL	`wrong` (✓)	1 / 1	✓
Case 2 (§H.2)	DIAGSQL (w/o reasoning)	`wrong` (✓)	1 / 1	✗
	DIAGSQL	`wrong` (✓)	1 / 1	✓
Case 3 (§H.3)	GPT-4o	`wrong` (✓)	1 / 4	✗
	DIAGSQL	`wrong` (✓)	4 / 4	✓
Case 4 (§H.4)	DIAGSQL (w/o Co-occ.)	`wrong` (✓)	2 / 4	✗
	DIAGSQL	`wrong` (✓)	4 / 4	✓

The four cases jointly establish three claims about DIAGSQL: (i) it can detect subtle semantic errors that even a strong general LLM such as GPT-4o waves through (Case H.1); (ii) the `reasoning` field in our output schema is not a stylistic adornment but a structurally necessary signal for downstream repair (Case H.2), and (iii) when both methods agree the SQL is wrong, DIAGSQL delivers strictly more—and more accurate—categorical and reasoning information, which is what eventually decides whether the regenerated SQL is correct (Cases H.3 and H.4). Co-occurrence shaping (Case H.4) is the ingredient that lets DIAGSQL recover entire *groups* of correlated errors rather than only the ones that are detectable in isolation.

I Limitations and Future Work

Scalability to ultra-large databases. DIAGSQL is deliberately built on compact 8B-parameter backbones (QWEN3-8B and LLAMA3.1-8B) to keep the validator deployable as a lightweight companion to a stronger upstream generator. This design choice inherits the moderate context window of the base models and constrains the schema footprint we can condition on. On SPIDER 2.0 [48], whose enterprise-scale databases span hundreds of tables and thousands of columns [50], the raw serialized schema can exceed this budget. The additional results in Appendix E show that DBCC database-context compression [49] makes DIAGSQL effective in this regime, improving DeepEye and Apex-SQL to 51.7 and 54.1 EX, respectively. Nevertheless, context construction remains an external prerequisite rather than a capability learned by the validator itself. Context-window extension methods such as Position Interpolation [51], YaRN [52], LongLoRA [53], StreamingLLM [54], and InfiniteICL [55], together with schema-linking approaches such as LinkAlign [56], are promising routes toward an end-to-end long-context variant.

Saturation under multi-round refinement. Our multi-round analysis (Appendix G) shows that DIAGSQL-driven refinement peaks early (around round 2) and then mildly regresses, rather than

improving monotonically with the refinement budget K. We identify three compounding factors behind this saturation. First, a *train–inference distribution mismatch*: DIAGSQL is trained exclusively on single-round diagnostic instances, so at rounds ≥ 2 the validator is forced to diagnose partially repaired SQLs whose error distribution differs markedly from its training data, leading to degraded calibration and occasional spurious diagnoses. Second, a *shifting population effect*: as genuinely buggy SQLs are progressively repaired in early rounds, the evaluation pool becomes increasingly dominated by already-correct queries, so the validator's residual false-positive rate (about 10%, Section F) gradually outweighs its true-positive gains and pushes a small fraction of clean SQLs into harmful rewrites. Third, an *upstream-generator ceiling*: a fraction of the remaining buggy queries are failure modes rooted in the generator's own schema-linking or reasoning capacity, and no amount of additional, accurate diagnostic feedback can fix them without improving the generator itself—repeated rounds therefore drive the generator into a fixed point rather than a better solution. Taken together, these factors imply that the headroom of a validator-only closed loop is fundamentally bounded by both its training regime and the capabilities of the paired generator. Closing this gap calls for *jointly* advancing the generator and the validator: adapting DIAGSQL to multi-round diagnostic trajectories (e.g., training on intermediate repair states or optimizing a multi-round cumulative reward), and co-designing stronger generators that can act on the fine-grained signals DIAGSQL provides. We believe such generator–validator co-evolution is a crucial next direction for pushing the Text-to-SQL community toward truly reliable closed-loop systems.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The main claims in the abstract and introduction accurately reflect the paper's contributions and scope.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The limitations of the work are discussed in detail in Appendix I.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: The paper does not present formal theoretical results that require proofs.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The de-anonymized public code repository is linked on the first page, and the training, evaluation, and implementation details are documented in Appendix B.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: Open access to the implementation and its documentation is provided through the de-anonymized repository linked on the first page.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: Full experimental details are included in Appendix B to ensure reproducibility.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Table 4 reports mean $\pm$ standard deviation of overall execution accuracy over five independent runs for the principal end-to-end configurations.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [No]

   Justification: The GPU type, count, memory, numerical precision, and memory-saving techniques are reported in Appendix B, but per-run wall-clock time and total project-level compute are not recorded.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conducted in this paper conforms to all aspects of the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [No]

    Justification: The paper focuses on technical reliability and limitations of Text-to-SQL validation but does not provide a dedicated analysis of both positive and negative societal impacts.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not involve the release of any new datasets or models.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [No]

    Justification: The paper credits and cites the datasets, models, and baseline systems that it uses, but it does not enumerate the license and terms of use of every third-party asset.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The implementation is documented in the de-anonymized public repository linked on the first page, together with the experimental details provided in the appendix.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: The proposed method trains and evaluates LLM-based SQL diagnostic validators. The roles of the LLMs and their configurations are described in Sections 3 and B.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.