R^3-SQL: Ranking Reward and Resampling for Text-to-SQL

Hojae Han[1*] **Yeonseok Jeong**[2*†] **Seung-won Hwang**[2‡] **Zhewei Yao**[3] **Yuxiong He**[3]
[1]ETRI, [2]Seoul National University, [3]Snowflake AI Research
hojae.han@etri.re.kr {jys3136,seungwonh}@snu.ac.kr
{zhewei.yao,yuxiong.he}@snowflake.com

Abstract

Modern Text-to-SQL systems generate multiple candidate SQL queries and rank them to judge a final prediction. However, existing methods face two limitations. First, they often score functionally equivalent SQL queries inconsistently despite identical execution results. Second, ranking cannot recover when the correct SQL is absent from the candidate pool. We propose R^3-SQL, a Text-to-SQL framework that addresses both issues through unified reward for ranking and resampling. R^3-SQL first groups candidates by execution result and ranks groups for consistency. To score each group, it combines a pairwise preference across groups with a pointwise utility from the best group rank and size, capturing relative preference, consistency, and candidate quality. To improve candidate recall, R^3-SQL introduces agentic resampling, which judges the generated candidate pool and selectively resamples when the correct SQL is likely absent. R^3-SQL achieves 75.03 execution accuracy on BIRD-dev, a new state of the art among methods using models with disclosed sizes, with consistent gains across five benchmarks.

1 Introduction

With advances in large language models (LLMs), Text-to-SQL systems adopt a generate-then-rank paradigm: an LLM samples multiple candidate SQL queries, then a ranker selects the best candidate from the pool (Pourreza et al., 2025; Agrawal and Nguyen, 2025; Liu et al., 2025) (Figure 1). Existing rankers can be categorized into *pointwise* (Agrawal and Nguyen, 2025), which scores each candidate s_i independently, or *listwise* comparing multiple candidates $\{s_1, \ldots, s_n\}$ jointly as Figure 1 illustrates.

However, both rankers face two limitations. First, rankers assign inconsistent scores to functionally equivalent SQL queries that differ in surface form but produce identical execution results (Launer et al., 2026; Li et al., 2026). For example, s_2 and s_4 in Figure 1 produce the same execution result but receive different scores in both pointwise and listwise rankers (Long et al., 2025). We term this *functional inconsistency*.

A common remedy is to group functionally equivalent candidates before ranking (Sheng and Xu, 2025; Launer et al., 2026; Li et al., 2026). However, existing methods rank groups by size alone (functional majority voting in Figure 1), so a small but correct group can still be outranked by a larger incorrect one. Second, rankers assume that the correct SQL already exists in the candidate pool. When the generator fails to produce a correct candidate, no ranking strategy can recover the correct answer. This limitation is well-known as *bounded recall* in the information retrieval literature (Wang et al., 2011; Rathee et al., 2025; MacAvaney et al., 2022; Zhang et al., 2021), addressed by resampling, yet remains unexplored in Text-to-SQL.

We propose R^3-SQL, a Text-to-SQL framework that addresses both limitations through unified modeling for reward ranking and resampling. First, to mitigate functional inconsistency, R^3-SQL groups candidates by execution result and ranks groups rather than individual SQL queries. Figure 1 illustrates our distinction, groupwise ranking combining two complementary signals. The first is listwise comparisons of pairs across groups, capturing relative preference. The second is a groupwise utility signal derived from both the rank of the representative sample and its size. Figure 2 illustrates how we unify three signals: relative inter-group preference, inner-group consistency, and individual candidate quality, and agentic resampling. Second, resampling ensures the candidate pool includes a correct generation, by predicting if a correct SQL query in

*Equal contribution.
†Work done while at Snowflake.
‡Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 43257–43275

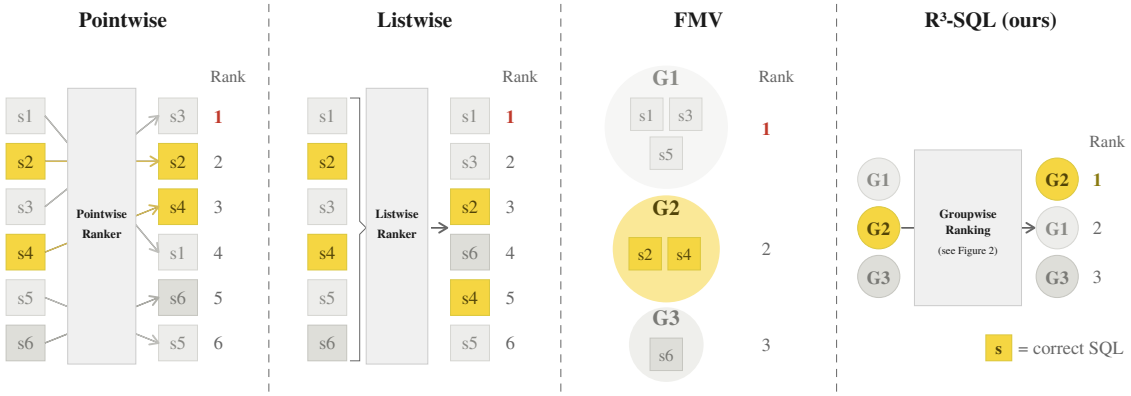

Figure 1: Comparison of ranking strategies for six SQL candidates, where s_2 and s_4 (yellow) are correct. Both pointwise and listwise rankers rank an incorrect candidate first. Functional majority voting (FMV) groups candidates by execution result but ranks the largest (incorrect) group G_1 first. R^3-SQL re-ranks groups using pointwise and listwise signals with group size, correctly placing G_2 at rank 1.

the pool and resampling otherwise.

Experiments on five Text-to-SQL benchmarks show that R^3-SQL consistently improves execution accuracy over prior ranking-based approaches. On BIRD-dev (Li et al., 2023), R^3-SQL achieves 75.03 execution accuracy, establishing a new state of the art among methods using models with disclosed sizes (Table 2). Further analysis confirms that each component contributes: groupwise scoring eliminates score variance among equivalent SQLs, the consistency objective improves input-order robustness by +11.89 pp, and agentic resampling raises candidate recall by +3.92 pp.

Our contributions are as follows:

- We identify two core limitations of ranking-based Text-to-SQL pipelines: functional inconsistency and bounded recall.
- We propose a group-based ranking framework that combines pointwise and listwise reward signals to resolve functional inconsistency.
- We introduce agentic resampling to selectively expand the candidate pool when the correct SQL is likely absent.
- We demonstrate state-of-the-art performance on five Text-to-SQL benchmarks and provide analysis of each component's contribution.

2 Related Work

Recent Text-to-SQL systems follow a generate-then-rank pipeline: sample multiple candidate SQL programs, then rank or filter to select a final prediction. As discussed in §1, two key challenges shape performance: functional inconsistency and bounded recall. Existing systems vary in how they address these challenges, as summarized in Table 1.

Contextual-SQL (Agrawal and Nguyen, 2025) trains a pointwise ranker to score each candidate independently without grouping, so functional inconsistency persists. It conducts extensive initial sampling but does not resample or refine candidates. **CHASE-SQL** (Pourreza et al., 2025) employs a pairwise ranker (listwise with n=2) that compares candidates relatively, but without execution-based grouping, functional inconsistency persists. It refines only syntax errors, so bounded recall from semantic errors remains unresolved. **Agentar-Scale-SQL** (Wang et al., 2025) employs a listwise ranker without grouping, and enhances recall by refining candidates through both syntax and semantic checks via an agent. **XiYan-SQL** (Liu et al., 2025) ensembles multiple generators and groups candidates by execution result, but ranks groups by size, so a small but correct group can be outranked by a larger incorrect one. It refines only syntax errors, so bounded recall from semantic errors remains unresolved. **OpenSearch-SQL** (Xie et al., 2025) adopts groupwise voting for selection, but this heuristic is limited compared to learned rankers and similarly relies on group size. It refines only syntax errors. **CSC-SQL** (Sheng and Xu, 2025) groups candidates by execution result and applies merge-revision over top-2 groups. However, the revision is applied indiscriminately even when a correct candidate already exists, which can introduce unnecessary errors. The absence of learned rankers limits precision in the final selection.

Method	LLM Size	Grouping	Ranker	Resample when	EX (%)
Model size undisclosed (UNK)					
Contextual-SQL	UNK	–	pointwise	never	73.50
CHASE-SQL	UNK	–	listwise	syntax errors	74.90
Agentar-Scale-SQL	UNK	–	listwise	syntax & semantic errors	74.90
XiYan-SQL	UNK	✓	listwise + FMV	syntax errors	73.34
OpenSearch-SQL	UNK	✓	FMV	syntax errors	69.30
Model size disclosed					
R^3-SQL	**32B**	✓	**groupwise (point + list) + FMV**	**syntax & semantic errors**	**75.03**
CSC-SQL	70B	✓	FMV	always	71.69
CSC-SQL	32B	✓	FMV	always	71.33
Agentar-Scale-SQL	32B	–	listwise	syntax & semantic errors	71.12
XiYan-SQL	32B	✓	listwise + FMV	syntax errors	67.01

Table 1: Comparison of recent Text-to-SQL systems on BIRD-dev. *Grouping* indicates whether candidates are grouped by execution result. *Ranker* specifies the scoring method; FMV denotes functional majority voting by group size. *Resample when* denotes the condition that triggers candidate resampling or refinement.

Our Distinction. As shown in Table 1, R^3-SQL addresses both limitations. For functional inconsistency, R^3-SQL groups candidates by execution result and ranks groups using both pointwise and listwise signals rather than group size alone. For bounded recall, R^3-SQL introduces agentic resampling that judges whether the candidate pool likely contains a correct SQL and selectively resamples when it does not. This achieves the highest EX on BIRD-dev among systems with disclosed model sizes.

3 R^3-SQL

As illustrated in Figure 2, R^3-SQL operates in two phases: exploration and exploitation. We first describe how candidates are grouped and ranked to resolve functional inconsistency (§3.1), then address bounded recall via agentic resampling (§3.2), followed by position-bias mitigation in the listwise ranker (§3.3). We summarize the overall framework in §3.4.

3.1 Improving Functional Inconsistency

Functional inconsistency arises when functionally equivalent SQL queries receive inconsistent scores due to superficial differences (e.g., token order or stylistic variations). R^3-SQL addresses this by evaluating candidates at the level of *execution semantics* rather than individually.

Groupwise Scoring. Instead of scoring each candidate independently, we group SQL candidates that yield the same execution result into a single cluster. Each group thus represents one distinct semantic outcome. Formally, let $\mathcal{G} = \{g_1, \ldots, g_M\}$ be the set of groups after executing all candidates and grouping those with identical results. By aggregating scores over execution-equivalent groups, we neutralize the noise introduced by superficial textual differences.

Grouping ensures consistent treatment within each group. The remaining question is how to rank across groups. We combine two signals: a cross-group preference from pairwise comparisons, which leverages multiple observations per group pair, and a utility from pointwise scoring and group size. The cross-group signal serves as the primary ranking criterion; when its margin is too small to be reliable, the pointwise utility breaks ties.

Cross-Group Preference Signal. We utilize a *pairwise*[1] ranker that compares candidates from different groups and produces a relative preference signal. Specifically, we train a dedicated pairwise ranker that takes the database schema db, the natural language question x, and two SQL candidates (along with their execution results) as input, and outputs a preference indicating which candidate is more likely to be correct.

Motivated by the Bradley–Terry model (Bradley and Terry, 1952), which views pairwise preferences as noisy observations of latent quality scores via

$$P(g_i \succ g_j) = \sigma(r_i - r_j), \tag{1}$$

where σ is the logistic sigmoid and r_i, r_j are latent utilities, we estimate the group-level preference $P(g_i \succ g_j)$ by aggregating pairwise comparisons between all candidates $s_i \in g_i$ and $s_j \in g_j$, i.e.,

[1]We use pairwise as a special case of listwise ranking with n=2.

$\frac{1}{|g_i||g_j|}\sum_{s_i\in g_i}\sum_{s_j\in g_j} v(s_i,s_j)$, where $v(s_i,s_j) \in \{0,1\}$ is the ranker's vote preferring s_i over s_j (1 if s_i is judged better, 0 otherwise). To avoid noise in close or ambiguous comparisons (e.g. if both candidates are incorrect), we apply a threshold τ to discount uncertain judgments.[2] Only a sufficiently large margin counts as a decisive preference of g_i:

$$\tilde{P}(g_i \succ g_j) = \begin{cases} +1, & \text{if } P(g_i \succ g_j) \geqslant \tau, \\ 0, & \text{otherwise,} \end{cases} \quad (2)$$

and we define the pairwise group score as the number of decisive wins against all other groups:

$$r_{\text{list}}(g_i) = \sum_{j\neq i} \tilde{P}(g_i \succ g_j)\,. \quad (3)$$

This pairwise procedure produces a group ordering that is robust to functional inconsistency. By comparing candidates side-by-side, our ranker can consistently select correct SQLs above those with only superficial token advantages.

Pointwise-Group Utility Signal. As complementary signal for group utility, we also use the *pointwise ranker* that provides scores independent of input order. We also incorporate a confidence measure function $w(g)$. For each execution-equivalent group g, we define

$$r_{\text{point}}(g) = w(g)\cdot u(g) \quad (4)$$

where $w(g)=|g|$ to reflect execution-level self-consistency and further stabilize rankings. We set $u(g)=\max_{s\in g} RR_s$, preserving the strongest evidence within each group, denoted as a colored rectangle in group g in Figure 2. This follows representation fusion literature (Liu and Croft, 2002), where RR_s denotes the reciprocal rank of candidate s according to the pointwise ranker. As $r_{\text{point}}(g)$ is independent of input position, it overcomes positional effects from r_{list}.

3.2 Mitigating the Bounded Recall Problem

As introduced in §1, even the best ranking strategy cannot succeed if the correct SQL is absent from the candidate pool. To address this, R^3-SQL augments the candidate sampling stage with an LLM-based agent that diagnoses coverage gaps and triggers targeted resampling when necessary, so the ranker operates on a pool more likely to contain a correct SQL.

[2]We fix τ=0.05 across the settings (see Appendix A.3).

Agentic Resampling. Our candidate sampling procedure is a two-pass process: initial pool generation and selective resampling. First, given a database schema db and natural language question x, we prompt a base LLM to produce the initial pool $S=\{s_1,\ldots,s_n\}$ of n SQL candidates. Then, an LLM agent f examines whether S contains the correct SQL, denoted as $f(S)\in\{0,1\}$. When the agent f decides S is insufficient, i.e., $f(S)$=0, it discards S and triggers the resampling:

$$S = \begin{cases} \tilde{S}, & \text{if } f(S)=0, \\ S, & \text{otherwise,} \end{cases} \quad (5)$$

where $\tilde{S}$ is the resampled pool. We sample a larger set $\tilde{S}=\{\tilde{s}_1,\ldots,\tilde{s}_m\}$ of m ($m>n$) new candidate SQLs from the base LLM. To focus on the most promising of these resampled candidates, we use the pointwise ranker (as used for anchoring) to score each $\tilde{s}_i$. The top-n scored queries are then selected to form the resampled pool, denoted simply as $\tilde{S}$ for brevity, matching the original pool size.

3.3 Improving Listwise Ranking

While the above ranking addresses functional inconsistency at the group level, the listwise ranker itself can be sensitive to the input order of candidates, known as position bias. R^3-SQL counters this by training the listwise ranker with a position-consistency objective.

Position-Consistency Objective. We train a listwise ranker to be robust to input order by presenting each correct–incorrect SQL pair (s^+, s^-) in both original and swapped positions. Within the GRPO framework (Shao et al., 2024), we introduce a consistency reward, inspired by the position-invariance objective of J1 (Whitehouse et al., 2025), to reinforce preferences that remain correct under both orderings. Specifically, we decompose the reward into a base term and an auxiliary consistency term:

$$R = R_{\text{base}} + \lambda_c\, R_c, \quad (6)$$

where $R_{\text{base}}\in\{0,1\}$ indicates whether the preference decision is correct, and $R_c\in\{0,1\}$ indicates whether the correct decision is preserved under order perturbation. We set λ_c=0.5 without additional tuning, following common RL practice of assigning a modest weight to auxiliary objectives (Brockman et al., 2016; Schulman et al., 2017). As a result, each correct–incorrect pair receives reward 0 if the

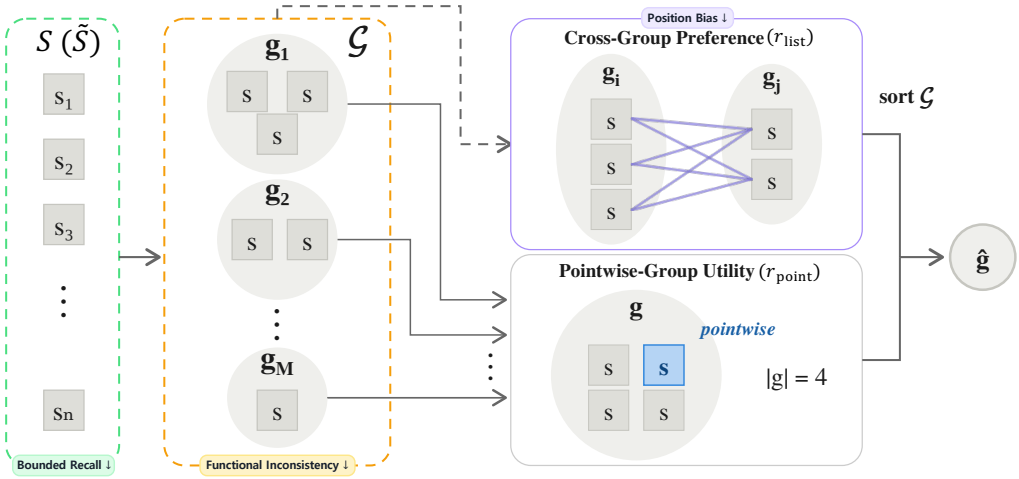

Figure 2: Groupwise Ranking by R^3-SQL.

decision is incorrect, 1 if it is correct, and 1.5 if it is correct and remains consistent across both input orders.

3.4 Overall Framework

As illustrated in Figure 2, R^3-SQL operates as an exploration-exploitation framework: the first phase expands the candidate pool via agentic resampling to address bounded recall, while the second phase ranks candidates through group-based ranking to resolve functional inconsistency.

Agentic resampling for Recall. A base LLM generates an initial pool S of candidate SQLs. An LLM agent f audits S, deciding to retain or replace it with a resampled pool $\tilde{S}$ according to the selection rule in Eq. (5).

Pointwise-listwise ranking for Precision. Execute all candidates in S (or $\tilde{S}$) and group those with identical execution results into $\mathcal{G} = \{g_1, \ldots, g_M\}$, each representing a distinct semantic outcome. Compute listwise preferences using Eqs. (1)–(2), and combine them with the pointwise group utility $r_{\text{point}}(g)$ from Eq. (4). Groups are then sorted by the lexicographic ordering:

$$\hat{\mathcal{G}} = \text{sort}(\mathcal{G}) \;\text{ by }\; \big(r_{\text{list}}(g),\; r_{\text{point}}(g)\big), \qquad (7)$$

where tuples are compared left-to-right. Because the threshold τ in Eq. (2) discards preference margins too small to distinguish from noise, groups with similar listwise quality receive identical r_{list} scores. For such ties, r_{point} provides a fallback based on absolute candidate quality and group consensus. Since the lexicographic ordering operates on group-level signals, we perform one final comparison at the individual SQL level. Let $\{g', g''\} = \text{Top2}(\hat{\mathcal{G}})$; the final group is selected as:

$$\hat{g} = \begin{cases} g', & \text{if } P(g' \succ g'') > 1/2, \\ g'', & \text{otherwise,} \end{cases} \qquad (8)$$

and the candidate with the highest pointwise rank within $\hat{g}$ is returned as the final prediction.

4 Experiments

4.1 Experimental Setup

Appendix C explains the implementation details.

Benchmarks. We evaluate R^3-SQL on 5 widely-used Text-to-SQL benchmarks: BIRD (Li et al., 2023), Spider (Yu et al., 2018), Spider-DK (Gan et al., 2021), EHR-SQL (Lee et al., 2022), and ScienceBenchmark (Zhang et al., 2023). Unlike the first three cross-domain benchmarks, EHRSQL and ScienceBenchmark are domain-specific, testing out-of-domain generalization capabilities. Detailed statistics of these benchmarks are shown in Appendix Table 19.

Metrics. Throughout the experiments, we use execution accuracy (EX) as the main metric, which is the official evaluation metric for the benchmarks. EX evaluates whether the execution result of the predicted SQL matches that of the ground-truth SQL when executed on SQLite. For fair evaluation, we exclude questions where the ground-truth SQL produces empty execution result due to issues such as database size limitation or timeout error.

Baselines. We use CSC-SQL (Sheng and Xu, 2025), Contextual-SQL (Agrawal and Nguyen,

SQL Selection Method	Ranker	BIRD-dev	Spider-test	Spider-DK	EHRSQL	Science Benchmark	**Avg.**
CSC-SQL	FMV	71.58	86.64	76.97	41.04	56.68	66.58
Contextual-SQL	Pointwise	73.14	86.36	75.50	41.41	63.13	67.91
CHASE-SQL	Listwise	73.34	86.18	75.94	44.44	63.59	68.70
XiYan-SQL	Listwise + FMV	72.03	85.89	75.28	43.43	63.59	68.04
R^3-SQL	Groupwise (Point + List) + FMV	**75.03**	**87.19**	**77.92**	**46.30**	**66.82**	**70.65**

Table 2: EX comparison of different selection methods across five benchmarks, where SQL candidates were generated by Arctic-Text2SQL-R1-32B (T=0.8). The same pointwise (R^3-POINT-32B) and listwise (R^3-7B; §3.3) rankers were consistently used across the methods.

Method	Score Variance (z-score; ↓)	EX (%)
Contextual-SQL	0.8571	73.14
R^3-SQL *w/o* R^3-7B	**0.0000**	73.47
R^3-SQL	**0.0000**	**75.03**

Table 3: Functional inconsistency mitigation by R^3-SQL on BIRD-dev. Score variance is measured among SQL candidates that yield identical execution results (lower is better).

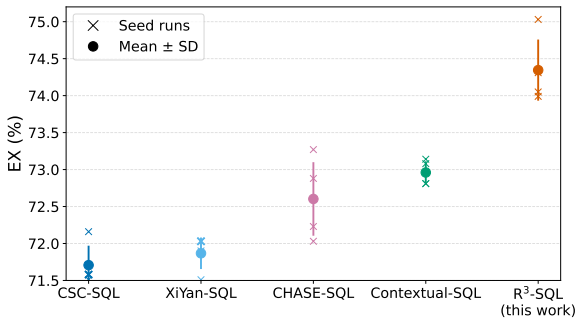

Figure 3: EX stability across 4 random seeds on BIRD-dev. Crosses indicate individual runs, and circles with error bars represent the mean ± SD. Candidates were generated by Arctic-Text2SQL-R1-32B (T=0.8).

2025), CHASE-SQL (Pourreza et al., 2025), and XiYan-SQL (Liu et al., 2025) as our baselines. To ensure fair comparison of selection algorithm, we employ the same pointwise ranker (R^3-POINT-32B) and listwise ranker (R^3-7B; §3.3) across all baseline methods.[3] R^3-POINT-32B is trained from the pointwise ranker used in Contextual-SQL (Contextual-RM-32B) on BIRD-train; details in Appendix C.

4.2 Experimental Results

Table 2 presents the comparative results across five diverse benchmarks. R^3-SQL consistently outperforms all baselines, achieving the highest EX on every dataset. R^3-SQL is the only method to break the 70% ceiling, achieving an average EX of 70.65%.

5 Analysis

5.1 Functional Inconsistency Mitigation

We define functional inconsistency as occurring when functionally equivalent SQLs with identical execution results but different surface forms receive (i) inconsistent scores, and (ii) one such incorrect variant is ranked above the correct SQL. In our evaluation, (i) is measured by the score variance among SQLs with the same execution results, and (ii) is reflected in EX. As shown in Table 3, Contextual-SQL solely using a pointwise ranker produces non-zero score variance. R^3-SQL's groupwise scoring assigns a single score per execution result, reducing this variance to 0.0 by design. Adding the listwise ranker R^3-7B raises EX by +1.56 pp via better aligning group scores with correctness.

5.2 Reproducibility across Seeds

To assess robustness against sampling randomness, we performed four independent runs on BIRD-dev following the setup in Table 2. As illustrated in Figure 3, while the relative rankings of baselines fluctuate due to variations in candidate pools, R^3-SQL consistently maintains the lead across all seeds. Most notably, the lowest performance recorded for R^3-SQL (73.99) surpasses the absolute peak performance of the strongest baseline (CHASE-SQL: 73.27), demonstrating that our method ensures superior performance regardless of generation noise.

5.3 Group Scoring Strategy

Table 4 compares group scoring strategies. Among single-signal approaches, Listwise performs best by leveraging multiple observations per group pair, followed by Pointwise (max-based). Averaging pointwise scores within a group (Pointwise avg.) performs worst, as weaker members dilute

[3]Except for Contextual-SQL, their original rankers are unreleased.

Group Scoring	BIRD-dev	Spider-test	Spider-DK	EHRSQL	Science Benchmark	**Avg.**
Pointwise	73.14	86.36	75.50	41.41	63.13	67.91
Pointwise (avg.)	71.90	86.03	74.39	39.73	62.21	66.85
Listwise	73.34	86.66	76.82	**46.63**	**67.28**	70.15
R^3-SQL	**75.03**	**87.19**	**77.92**	46.30	66.82	**70.65**

Table 4: Comparison of group scoring strategies across five benchmarks. *Pointwise* scores each candidate independently without grouping. *Pointwise (avg.)* ranks groups by the average pointwise score of their members. *Listwise* uses only the pairwise preference signal r_{list}. R^3-SQL combines r_{list} and r_{point} (max-based) via lexicographic ordering (Eq. 7).

SQL Selection Method	BIRD-dev	Spider-test	Spider-DK	EHRSQL	Science Benchmark	**Avg.**
R^3-SQL	**84.62**	**93.57**	**88.30**	**67.85**	**79.26**	**82.72**
w/o agentic resampling	81.23	92.21	87.20	58.25	75.11	78.80

Table 5: Bounded recall mitigation by R^3-SQL's agentic resampling. Values represent the recall of generated candidates, which corresponds to the ranking upper bound (i.e., the maximum achievable EX by an optimal ranker).

Method	Input Consistency (%;↑)
R^3-7B	**57.49**
w/o consistency reward	45.60
w/o GRPO	37.82

Table 6: Position bias mitigation in R^3-SQL's listwise ranker via consistency reward during GRPO training. We evaluate consistency across swapped candidate orders (pos-neg vs. neg-pos) on BIRD-dev using candidates from Arctic-Text2SQL-R1-32B (T=0.8)

Method	EX (%)
R^3-SQL	**75.03**
w/o R^3-POINT-32B	74.19

Table 7: Position bias mitigation by R^3-SQL's pointwise ranker on BIRD-dev.

the strongest candidate's signal. R^3-SQL combines r_{list} and r_{point} via lexicographic ordering and achieves the highest average EX (70.65). On in-domain benchmarks, R^3-SQL outperforms Listwise by +1.11 pp on average. On out-of-domain benchmarks, Listwise slightly outperforms R^3-SQL, as the domain-limited pointwise ranker introduces noise in unseen domains (see Appendix A.1, Table 12).

5.4 Bounded Recall Mitigation

Table 5 demonstrates the effectiveness of R^3-SQL's agentic resampling in mitigating the bounded recall problem. With the auditing agent f effectively triggering resampling when necessary, we observe a consistent increase in candidate recall across all benchmarks, raising the average ranking upper bound by +3.92 pp (78.80→82.72). This indicates that the agent successfully identifies scenarios where the initial candidate pool lacks the correct SQL.

This improvement is orthogonal to the ranking process: rankers select the best SQL within a fixed set, while the resampling module expands the search space to include correct candidates. Together, they ensure the ranker operates on a higher-quality candidate pool.

5.5 Position Bias Mitigation

To verify whether the position-consistency objective mitigates positional bias, we compare R^3-7B against two variants: without consistency reward and without GRPO training (Table 6). The consistency reward yields the largest gain, improving input consistency by +11.89 pp over GRPO alone (45.60→57.49). Table 7 further shows that removing the pointwise ranker drops EX by −0.84 pp (75.03→74.19), confirming its role as an order-insensitive anchor.

5.6 Computational Overhead

Table 8 illustrates the trade-off between computational overhead and selection accuracy. Among the high-performing listwise approaches, CHASE-SQL serves as a strong baseline but incurs the highest computational cost (1.68 sec/query). In contrast, R^3-SQL achieves 75.03 EX while reducing inference time by 0.12 sec/query compared to CHASE-SQL.

SQL Selection Method	# Pointwise Ranking Calls per Query	# Listwise Ranking Calls per Query	Inference Time per Query (sec/query; ↓)	EX (%)
CSC-SQL	–	–	0.00	71.58
Contextual-SQL	32	–	0.26	73.14
XiYan-SQL	–	2	0.03	72.03
CHASE-SQL	–	138	1.68	73.34
R^3-SQL	32	107	1.56	**75.03**
always-resample	32	138	1.96	74.25

Table 8: Computational overhead and performance trade-off of SQL selection methods on BIRD-dev. All inference times were measured on the same machine equipped with 8×H200 GPUs.

Method	EX (%)
R^3-SQL	**75.03**
w/o Agentic Resampling	74.25
w/o Pointwise Pruning	73.92
w/o Exec. Group Scoring	73.47
w/o Pointwise Ranker	73.34
w/o Listwise Ranker	73.14

Table 9: Ablation study of R^3-SQL on BIRD-dev. All variants rank SQL candidates generated by Arctic-Text2SQL-R1-32B (T=0.8).

Resampling	Candidate Pool	EX (%)
None	Original candidates (S)	74.25
Always	Resampled candidates ($\tilde{S}$)	74.32
Agentic	Union ($S \cup \tilde{S}_{\text{top-}n}$)	73.92
	Replace ($S \rightarrow \tilde{S}$) where $m{=}n$	74.05
	Replace ($S \rightarrow \tilde{S}$) where $m{>}n$	**75.03**

Table 10: Effect of agentic resampling in R^3-SQL on BIRD-dev EX. All variants rank SQL candidates generated by Arctic-Text2SQL-R1-32B (T=0.8).

This efficiency arises from ranking only distinct execution groups to avoid redundancy, and employing agentic resampling that activates for just 37.01% of test instances. This targeted approach is 0.40 sec/query faster than the always-resample variant, while achieving even higher accuracy.

5.7 Ablation Study

To verify the contribution of the ranking and resampling modules in R^3-SQL, we conduct an ablation study on BIRD-dev. As shown in Table 9, every variant underperforms the full R^3-SQL system (75.03). Specifically, removing agentic resampling causes a significant drop to 74.25 (−0.78 pp), confirming the importance of candidate coverage for bounded recall. Next, disabling pointwise pruning further reduces EX to 73.92 (−0.33 pp).

Based on this setup, we further ablate the ranking methodology. Removing execution-group scoring leads to a decline to 73.47 (−0.45 pp), demonstrating the importance of consistent rewards for execution-equivalent SQLs. Finally, using single rankers proves less effective than our dual-reward design: relying solely on the listwise ranker (*w/o pointwise*) yields 73.34 (−0.58 pp), while relying only on the pointwise ranker (*w/o listwise*) drops performance to 73.14 (−0.78 pp).

Decision Class	Precision	Recall	F1-Score
Trigger Resampling	93.27	56.02	70.00
Skip Resampling	31.22	83.17	45.40

Table 11: Precision, recall, and F1-score of Agent f on resampling decisions. Candidates were generated by Arctic-Text2SQL-R1-32B (T=0.8).

5.8 Agentic Resampling

To validate the effectiveness of our agentic resampling module, we analyze both the downstream impact on execution accuracy (Table 10) and the agent's decision capability (Table 11).

As shown in Table 10, naive strategies fail to yield significant gains. *Always* resampling provides only a marginal improvement (+0.07 pp), while simply taking the *Union* of original and resampled candidates actually degrades performance to 73.92, suggesting that indiscriminately adding candidates introduces noise that distracts the ranker. In contrast, our proposed approach selectively replaces the pool with a larger set ($m{>}n$) only when triggered, achieving the highest EX of 75.03.

The success of our method is explained by the agent's decision metrics in Table 11. The agent demonstrates high precision (93.27) in triggering resampling, ensuring that valid candidate pools are rarely discarded (minimizing false positives). It targets only the most problematic queries for repair, and the high recall for the *Skip* class (83.17) prevents unnecessary computational costs.

6 Conclusion

In this paper, we introduced R^3-SQL, a Text-to-SQL framework that addresses functional inconsistency and bounded recall through unified reward modeling. R^3-SQL groups candidates by execution result and ranks groups using two complementary signals: a cross-group preference from pairwise comparisons and a single-group utility from pointwise scoring and group size. To address bounded recall, an agentic resampling module selectively expands the candidate pool when the correct SQL is likely absent. On BIRD-dev, R^3-SQL achieves 75.03 EX, a new state of the art among methods using models with disclosed sizes.

Acknowledgment

This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2022-0-00995, Automated reliable source code generation from natural language descriptions) and ITRC(Information Technology Research Center) support program(IITP-2026-RS-2020-II201789) supervised by the IITP.

7 Limitation

While R^3-SQL excels in in-domain settings, the reliance on a supervised pointwise ranker constrains generalization. As observed in Tables 2 and 12, the domain gap in the pointwise ranker leads to a marginal performance difference (0.46–0.67) compared to the pointwise-ablated variant on out-of-domain benchmarks. Importantly, however, R^3-SQL consistently outperforms all baselines even with this module included. Integrating a domain-generalized pointwise ranker remains a promising direction for future work.

References

Sheshansh Agrawal and Thien Nguyen. 2025. Open-sourcing the best local text-to-sql system.

Ralph Allan Bradley and Milton E Terry. 1952. Rank analysis of incomplete block designs: I. The method of paired comparisons. *Biometrika*, 39(3/4):324–345.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

Yujian Gan, Xinyun Chen, and Matthew Purver. 2021. Exploring underexplored limitations of cross-domain text-to-SQL generalization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8926–8931, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Tim Launer, Jonas Hübotter, Marco Bagatella, Ido Hakimi, and Andreas Krause. 2026. Majority voting for code generation. In *Third Workshop on Test-Time Updates (Main Track)*.

Gyubok Lee, Hyeonji Hwang, Seongsu Bae, Yeonsu Kwon, Woncheol Shin, Seongjun Yang, Minjoon Seo, Jongyeup Kim, and Edward Choi. 2022. Ehrsql: a practical text-to-sql benchmark for electronic health records. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin SU, ZHAOQING SUO, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. 2025. Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows. In *The Thirteenth International Conference on Learning Representations*.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, and 1 others. 2025. Omnisql: Synthesizing high-quality text-to-sql data at scale. *arXiv preprint arXiv:2503.02240*.

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. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *Thirty-Seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Zhuohao Li, Wenqing Chen, Jianxing Yu, and Zhichao Lu. 2026. Functional consistency of llm code embeddings: A self-evolving data synthesis framework for benchmarking. *Expert Systems with Applications*, 298:129523.

Xiaoyong Liu and W. Bruce Croft. 2002. Passage retrieval based on language models. In *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, CIKM '02, page 375–382, New York, NY, USA. Association for Computing Machinery.

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

Kehan Long, Shasha Li, Chen Xu, Jintao Tang, and Ting Wang. 2025. Precise zero-shot pointwise ranking with llms through post-aggregated global context information. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2384–2394.

Sean MacAvaney, Nicola Tonellotto, and Craig Macdonald. 2022. Adaptive re-ranking with a corpus graph. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 1491–1500.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2025. CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. In *The Thirteenth International Conference on Learning Representations*.

Mandeep Rathee, Sean MacAvaney, and Avishek Anand. 2025. Guiding retrieval using llm-based listwise rankers. In *European Conference on Information Retrieval*, pages 230–246. Springer.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Lei Sheng and Shuai-Shuai Xu. 2025. Csc-sql: Corrective self-consistency in text-to-sql via reinforcement learning. *arXiv preprint arXiv:2505.13271*.

Lidan Wang, Jimmy Lin, and Donald Metzler. 2011. A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 105–114.

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

Chenxi Whitehouse, Tianlu Wang, Ping Yu, Xian Li, Jason Weston, Ilia Kulikov, and Swarnadeep Saha. 2025. J1: Incentivizing thinking in llm-as-a-judge via reinforcement learning. *arXiv preprint arXiv:2505.10320*.

Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment. *Proceedings of the ACM on Management of Data*, 3(3):1–24.

Zhewei Yao, Guoheng Sun, Lukasz Borchmann, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. 2025. Arctic-text2sql-r1: Simple rewards, strong reasoning in text-to-sql. *arXiv preprint arXiv:2505.20315*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task.

Yi Zhang, Jan Deriu, George Katsogiannis-Meimarakis, Catherine Kosten, Georgia Koutrika, and Kurt Stockinger. 2023. Sciencebenchmark: A complex real-world benchmark for evaluating natural language to sql systems. *arXiv preprint arXiv:2306.04743*.

Yue Zhang, ChengCheng Hu, Yuqi Liu, Hui Fang, and Jimmy Lin. 2021. Learning to rank in the age of muppets: Effectiveness–efficiency tradeoffs in multi-stage ranking. In *Proceedings of the Second Workshop on Simple and Efficient Natural Language Processing*, pages 64–73.

Appendices

A Further Analysis

A.1 Domain Generalization

To validate the effectiveness of R^3-SQL across in-domain and out-of-domain (OOD), we conduct the experiments shown in Table 12. The *w/o* R^3-POINT-32B applies our groupwise selection while treating all candidates as having equal pointwise ranker scores.

As shown in Table 12, R^3-SQL achieves slightly higher gains on OOD benchmarks when the pointwise ranker is excluded. We attribute this to the difference in training data diversity between the two ranking components. Our listwise ranker, R^3-7B, is based on OmniSQL-7B, which is trained on a broad corpus including BIRD, Spider, and SynSQL-2.5M (Li et al., 2025), which covers a wide spectrum of domains. In contrast, our pointwise ranker R^3-POINT-32B is based on Contextual-RM-32B, which is fine-tuned exclusively on BIRD. Consequently, the pointwise scores are less reliable on unseen domains, creating a bottleneck that marginally offsets the generalized performance of the listwise ranker. Integrating a domain-generalized pointwise ranker in the future would further improve OOD performance.

A.2 Binary Accuracy of Pairwise Rankers

To validate R^3-7B (§3.3), which operates as a listwise ranker over candidate pairs, we report binary selection accuracy on BIRD-dev (Table 14). We compare our final ranker (GRPO + *consistency reward*) against both GRPO and an SFT-based selector, which represents the selection module used in approaches such as CHASE-SQL. R^3-7B consistently outperforms the strongest baseline, GRPO, across different base models. Specifically, it achieves 78.86 on OmniSQL-7B (+2.78 pp over GRPO) and 76.85 on Qwen2.5-Coder-7B (+2.63 pp over GRPO), improving selection quality across both backbones. Table 15 further supports the superiority of R^3-7B on the end-to-end EX performance.

A.3 Performance across τ

To evaluate how the threshold τ in Eq. (2) affects R^3-SQL's execution accuracy, Figure 4 shows execution accuracy as τ is varied. The curve peaks at very low τ and then declines as τ increases. At low τ, R^3-SQL applies the normalized $[0, 1]$ group-mean score only when the pairwise ranker strongly indicates that the group consists of incorrect candidates; otherwise the group is assigned the default score of 1. This confidence-gated strategy reduces bias from negative–negative pairings that can accidentally elevate incorrect candidate groups over correct ones, which is a practical source of false positives. Overall, the results favor tuning τ toward lower, more selective values so that listwise ranker is applied only when reliable, rather than always trusting fine-grained listwise scores.

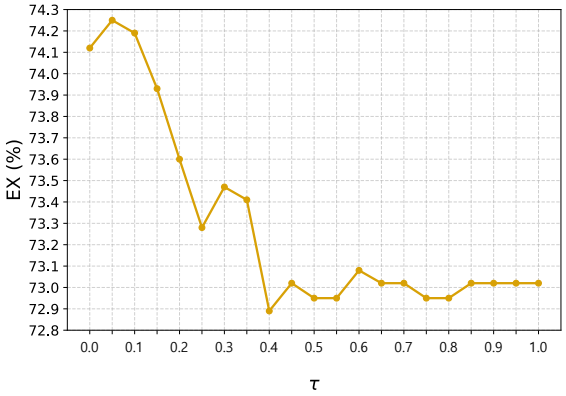

Figure 4: Execution accuracy (EX) of R^3-SQL on BIRD-dev across the threshold τ in Eq. (2). Each point ranks n=32 SQL candidates per query generated by Arctic-Text2SQL-R1-32B with nucleus sampling (T=0.8).

A.4 Generalization to SQL Generators

To verify the generator-agnostic robustness of R^3-SQL, we replicate the evaluation using OmniSQL-7B (Table 16). This smaller model creates a more challenging selection environment due to a lower yield of correct candidates. Despite the noisier candidate pool, R^3-SQL achieves 71.83 EX, outperforming the strongest baseline (Contextual-SQL, 70.80) by +1.03 pp. This confirms that our framework remains effective even with weaker underlying generators.

A.5 Generalization to Agentic Workflows

While our main experiments on 5 text2sql benchmarks already evaluate the core reasoning ability of SQL candidate selection, we conduct an additional evaluation on the Spider2.0 (Lei et al., 2025) to examine whether the benefit of R^3-SQL transfers to a more realistic setting. Following the evaluation protocol of Yao et al. (2025), we focus on the SQLite subset so that the comparison remains centered on the SQL generation/selection stage.

SQL Selection Method	Ranking	In-domain				Out-of-domain		
		BIRD-dev	Spider-test	Spider-DK	**Avg.**	EHRSQL	ScienceBenchmark	**Avg.**
R^3-SQL	Groupwise (Point + List) + FMV	**75.03**	**87.19**	**77.92**	**80.05**	46.30	66.82	56.56
w/o R^3-POINT-32B	Listwise + FMV	74.19	86.71	76.82	79.24	**46.97**	**67.28**	**57.13**

Table 12: Performance on in-domain (BIRD-dev, Spider-test, Spider-DK) and out-of-domain (EHRSQL, ScienceBenchmark) datasets w.r.t. the ranker's training distribution.

As shown in Table 17, R^3-SQL achieves the best EX score of 29.17, outperforming all baselines. Although this experiment is not a full end-to-end evaluation of an agentic system, it provides complementary evidence that our method remains effective under more realistic schema settings and can be naturally incorporated as a ranking and selection module within broader agentic Text-to-SQL pipelines.

A.6 Efficiency Evaluation

To complement EX, we additionally evaluate selection methods using Reward-based Valid Efficiency Score (R-VES)[4] on BIRD-dev, where R-VES extends execution-based evaluation by considering the efficiency of valid SQL queries.

Table 18 reports the EX and R-VES scores of different selection methods. R^3-SQL achieves the best performance on both metrics, obtaining 75.03 EX and 69.73 R-VES. In particular, it outperforms the strongest baseline, CHASE-SQL, by 1.69 pp in EX and 1.04 pp in R-VES. These results indicate that the advantage of R^3-SQL is not limited to execution correctness, but also extends to the efficiency of selected valid SQLs.

A.7 Robustness to Grouping Edge Cases

We analyzed potential failure modes of execution-based grouping: cases where all execution results are distinct (making grouping impossible) or all are empty (making grouping uninformative). Our empirical analysis on BIRD-dev reveals that the *"all-distinct"* scenario is virtually non-existent (0%), reflecting the inherent convergence of LLM outputs. Conversely, the *"all-empty"* scenario occurred in 1.43% (22/1,534) of queries. Critically, our framework does not fail in these instances; instead, the agentic resampling module detects the low confidence or absence of valid content and triggers the generation of new candidates to resolve the ambiguity.

[4] https://github.com/bird-bench/mini_dev

B Discussion

B.1 Robustness against Coincidental Correctness

A potential challenge in execution-based grouping is the occurrence of false positives, which are semantically incorrect SQLs that coincidentally yield the correct execution result. In our framework, these candidates are naturally grouped with the correct SQLs since they share the same execution output. While this introduces semantic noise within the target group, our scoring and selection mechanism is explicitly designed to filter out such *"lucky guesses."*

The final representative for each group is determined by the candidate with the highest pointwise score (Eq (4)), not selected randomly. Since the pointwise ranker evaluates the semantic alignment between the SQL and the question, it assigns lower scores to false positives compared to valid SQLs. Consequently, even if false positives exist within the correct group, the group anchor remains a semantically correct SQL, ensuring that the final decision is guided by the most reliable candidate rather than by noise.

B.2 Case Studies: Functional Inconsistency in Real BIRD Failures

Figure 5 presents two real BIRD-dev case studies, sorted by pointwise RM score. In both examples, multiple incorrect SQLs share the same execution result but receive different pointwise scores, revealing that pointwise RM is sensitive to token-level variations even when execution semantics are identical. As a result, the highest-scored candidate under the pointwise RM is incorrect in both cases. R^3-SQL resolves this issue by first grouping candidates by execution result and then applying the listwise RM over execution groups, which promotes the correct execution group to Top 1.

C Implementation Details

Candidate Sampling. We generate the initial pool of SQL candidates using Arctic-Text2SQL-

SQL Selection Method	Ranking	BIRD-dev	Spider-test	Spider-DK	EHRSQL	Science Benchmark	Avg.
Contextual-SQL							
w/ Contextual-RM-32B	Pointwise	73.01	86.54	75.11	41.01	63.36	67.81
w/ R^3-POINT-32B	Pointwise	73.14	86.36	75.50	41.41	63.13	67.91
R^3-SQL							
w/ Contextual-RM-32B	Groupwise (Point + List) + FMV	74.90	87.29	77.70	45.79	66.82	70.50
w/ R^3-POINT-32B	Groupwise (Point + List) + FMV	75.03	87.19	77.92	46.30	66.82	70.65

Table 13: EX comparison between Contextual-RM-32B and our optimized pointwise ranker, R^3-POINT-32B, across five benchmarks.

Method	Base Model	Binary Acc. (%)
GRPO + *consistency reward*	OmniSQL-7B	**78.86**
GRPO	OmniSQL-7B	76.08
SFT	OmniSQL-7B	67.66
GRPO + *consistency reward*	Qwen2.5-Coder-7B	**76.85**
GRPO	Qwen2.5-Coder-7B	74.22

Table 14: Binary selection accuracy of pairwise rankers on BIRD-dev. GRPO + *consistency reward* is our proposed ranker, while SFT represents the supervised selector used in approaches such as CHASE-SQL. Positive–negative pairs were generated using Qwen2.5-Coder-7B (T=0.8).

Method	Base Model	EX (%)
GRPO + *consistency reward*	OmniSQL-7B	**71.45**
GRPO	OmniSQL-7B	70.79
SFT	OmniSQL-7B	69.82
GRPO + *consistency reward*	Qwen2.5-Coder-7B	**71.19**
GRPO	Qwen2.5-Coder-7B	69.88

Table 15: EX of pairwise rankers on BIRD-dev. GRPO + *consistency reward* denotes our ranker, and SFT represents the supervised selector used in approaches such as CHASE-SQL. Positive–negative pairs were generated using Qwen2.5-Coder-7B (T=0.8).

R1-32B (Yao et al., 2025). Following its guideline,[5] we utilize the prompt template shown in Figure 6 and set the sampling temperature to 0.8. For each question, we generate n=32 candidate SQLs in the initial pool, and selectively resample m=1,024 candidates.

Agentic Resampling. For the agentic resampling, agent f must determine whether the initial candidate pool contains a correct answer. Specifically, the agent observes the question, database schema, candidate SQL queries, and their execution results with the prompt shown in Figures 8 and 9. To effectively follow this instruction, training data is required, so we generate candidates from BIRD-train under same candidate generation setting as used at evaluation time. We then construct examples where the pool does or does not contain the correct answer. We evaluate three approaches using Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) as the backbone: In-Context Learning (ICL)[6], Supervised Fine-Tuning (SFT)[7], and GRPO[8]. Then, ICL performs worse (74.44), while SFT and GRPO achieve identical performance (75.03). We adopt SFT, due to its efficiency. We use this LLM agent as the auditing agent f.

[5] Arctic-Text2SQL-R1's github

SQL Selection Method	BIRD-dev
CSC-SQL	67.40
Contextual-SQL	70.80
CHASE-SQL	68.38
XiYan-SQL	68.06
R^3-SQL	**71.83**

Table 16: EX comparison of different SQL selection methods on the BIRD-dev benchmark, where SQL candidates were generated by OmniSQL-7B.

Method	Spider2.0-SQLite
CSC-SQL	12.50
Contextual-SQL	16.67
CHASE-SQL	20.83
XiYan-SQL	16.67
R^3-SQL	**29.17**

Table 17: EX comparison of different selection methods on the Spider 2.0-SQLite, following Yao et al. (2025).

Rankers. To train our listwise ranker, R^3-7B, we use OmniSQL-7B (Li et al., 2025) as the backbone model, which is based on Qwen2.5-Coder-7B-Instruct. We construct training data by generating 32 SQL candidates from BIRD-train and Spider-train, then sampling one correct and one incorrect candidate per question to form positive-negative pairs. Since the resampled pool contains many SQL candidates with high pointwise ranker scores, we sample hard negatives from the Top 15 highest-

[6] ICL randomly samples few-shot from the training data.
[7] Trained with gold labels indicating answer presence.
[8] Reward of 1 for correct prediction, 0 otherwise.

Method	EX (%)	R-VES
CSC-SQL	71.58	67.29
Contextual-SQL	73.14	67.80
CHASE-SQL	73.34	68.69
XiYan-SQL	72.03	67.28
R^3-SQL	**75.03**	**69.73**

Table 18: Comparison of different selection methods on BIRD-dev using EX and Reward-based Valid Efficiency Score (R-VES). EX results are copied from Table 2. Higher is better for both metrics.

Benchmark	#Questions	#Domains	Domain Type
BIRD-dev	1,534	37+	Cross-domain
Spider-test	2,147	138	Cross-domain
Spider-DK	535	138	Cross-domain
EHRSQL	1,008	1	Clinical
ScienceBenchmark	299	3	Scientific

Table 19: Statistics of evaluation splits across five Text-to-SQL benchmarks.

scoring incorrect candidates during training. The prompt template for pairwise comparison is shown in Figure 7. The sequence length is set to 8K tokens. When prompts exceed this limit, we proportionally truncate the execution results from both candidates to minimize information loss.

Our pointwise ranker, R^3-POINT-32B, is initialized from Contextual-RM-32B.[9] According to Contextual-SQL (Agrawal and Nguyen, 2025), Contextual-RM-32B is trained with Qwen2.5-Coder-32B-Instruct as its backbone. We further optimize this model on BIRD-train by selecting hard negative SQL candidates sorted by R^3-7B and training with the same recipe as Contextual-SQL. As shown in Table 13, the additional training on BIRD-train show a slight gain on BIRD-dev, while no consistent trend is observed on other benchmarks.

Experimental Infrastructure. All training and evaluation experiments are conducted on a single node equipped with 8 NVIDIA H200 GPUs.

D Prompt Templates

We provide the complete prompt templates used throughout our framework. Figure 6 shows the initial SQL generation prompt with chain-of-thought reasoning. Figure 7 presents the pairwise comparison prompt for training and evaluating R^3-7B. Figures 8 and 9 describe the system and user prompts for the agent-guided generation module, which audits candidate pools and produces structured guidance for resampling.

[9] Contextual-RM-32B's huggingface

Case 1: BIRD-dev Question 196

Question. In the non-carcinogenic molecules, how many contain chlorine atoms?

SQL1 (*Incorrect*; pointwise RM score: -3.8625)
```
SELECT COUNT(T2.molecule_id)
FROM molecule AS T1
JOIN atom AS T2 ON T1.molecule_id = T2.molecule_id
WHERE T1.label = '-' AND T2.element = 'cl';
```
Execution result: `[[201]]`

SQL2 (*Correct*; pointwise RM score: -4.8750)
```
SELECT COUNT(DISTINCT m.molecule_id)
FROM molecule AS m
JOIN atom AS a ON m.molecule_id = a.molecule_id
WHERE m.label = '-' AND a.element = 'cl';
```
Execution result: `[[71]]`

SQL3 (*Incorrect*; pointwise RM score: -5.0250)
```
SELECT COUNT(*)
FROM atom
WHERE molecule_id IN (
   SELECT molecule_id
   FROM molecule
   WHERE label = '-'
) AND element = 'cl';
```
Execution result: `[[201]]`

Observation. SQL1 and SQL3 count chlorine atoms rather than distinct molecules containing chlorine. Pointwise RM ranks **SQL1** $>$ **SQL2** $>$ **SQL3**, although SQL1 and SQL3 belong to the same incorrect execution group (`[[201]]`). R^3-SQL instead ranks the correct execution group (**SQL2**, `[[71]]`) as **Top 1** and the incorrect execution group (**SQL1, SQL3**, `[[201]]`) as **Top 2**.

Case 2: BIRD-dev Question 1527

Question. Which gas station has the highest amount of revenue?

SQL1 (*Incorrect*; pointwise RM score: -5.3625)
```
SELECT GasStationID
FROM transactions_1k
GROUP BY GasStationID
ORDER BY SUM(Amount * Price) DESC
LIMIT 1;
```
Execution result: `[[3250]]`

SQL2 (*Correct*; pointwise RM score: -5.8125)
```
SELECT GasStationID
FROM transactions_1k
GROUP BY GasStationID
ORDER BY SUM(Amount) DESC
LIMIT 1;
```
Execution result: `[[4347]]`

SQL3 (*Incorrect*; pointwise RM score: -6.1125)
```
SELECT T2.GasStationID
FROM transactions_1k AS T1
INNER JOIN gasstations AS T2 ON T1.GasStationID = T2.GasStationID
GROUP BY T2.GasStationID
ORDER BY SUM(T1.Price * T1.Amount) DESC
LIMIT 1;
```
Execution result: `[[3250]]`

Observation. SQL1 and SQL3 use `SUM(Amount * Price)`, while the correct formulation under the dataset semantics is `SUM(Amount)`. Pointwise RM ranks **SQL1** $>$ **SQL2** $>$ **SQL3**, although SQL1 and SQL3 belong to the same incorrect execution group (`[[3250]]`). R^3-SQL instead ranks the correct execution group (**SQL2**, `[[4347]]`) as **Top 1** and the incorrect execution group (**SQL1, SQL3**, `[[3250]]`) as **Top 2**.

Figure 5: Real BIRD-dev case studies illustrating functional inconsistency in pointwise ranking, sorted by pointwise RM score (higher is better).

Initial Pool Generation Prompt

System:
You are a data science expert. Below, you are provided with a database schema and a natural language question. Your task is to understand the schema and generate a valid SQL query to answer the question.

User:
Database Engine:
SQLite

Database Schema:
`{Database Schema}`
This schema describes the database's structure, including tables, columns, primary keys, foreign keys, and any relevant relationships or constraints.

Question:
`{evidence + question}`

Instructions:
- Make sure you only output the information that is asked in the question. If the question asks for a specific column, make sure to only include that column in the SELECT clause, nothing more.
- The generated query should return all of the information asked in the question without any missing or extra information.
- Before generating the final SQL query, please think through the steps of how to write the query.

Output Format:
Please provide a detailed chain-of-thought reasoning process and include your thought process within `<thinking>` tags. Your final answer should be enclosed within `<answer>` tags. Ensure that your SQL query follows the correct syntax and is formatted as follows:

```sql
-- Your SQL query here
```

Example format:
<thinking>
Step-by-step reasoning, including self-reflection and corrections if necessary. [Limited by 4K tokens]
</thinking>

<answer>
Summary of the thought process leading to the final SQL query. [Limited by 1K tokens]

```sql
Correct SQL query here
```
</answer>

Figure 6: Prompt for the Initial Pool Generation stage, where the LLM generates SQL candidates with chain-of-thought reasoning.

Prompt Template for Training and Evaluating Pairwise Reward Model

System:
You are a SQL expert. When given a SQL question along with two proposed solutions candidates A and B, your task is to evaluate the options based on clarity, efficiency, and adherence to best practices. Provide your answer strictly as either "A" or "B".

User:
Instruction:
Given the DB info and question, there are two candidate queries. There is correct one and incorrect one.
- First, think why one candidate is better than the other by comparing the two candidate answers and analyzing the differences of the query and the result.
- Then, based on your analysis, the original question, and the provided database info, select the better candidate query.
- Do not generate a new SQL query; focus solely on comparing the two given candidates.
****

Database Schema
`{Database Schema}`
****

Question:
`{Question}`
****

Candidate A
`{SQL Query A}`
Execution result
`{Execution Result A}`
****

Candidate B
`{SQL Query B}`
Execution result
`{Execution Result B}`
****

Output Format:
Please provide a detailed chain-of-thought reasoning process and include your thought process within `<think>` tags. Your final answer should be enclosed within `<answer>` tags.
****

Example format:
`<think>` Step-by-step reasoning, including self-reflection and corrections if necessary. [Limited by 4K tokens] `</think>`
`<answer>` Only write “A” or “B” depending on which is the correct answer. Do not include any other text. `</answer>`
Assistant:
Let me solve this step by step.
`<think>`

Figure 7: Prompt template for pairwise reward model comparing two SQL candidates.

System Prompt in Agentic SQL Candidate Generation

You are a SQL Candidate Gatekeeper.
Your job is to decide whether there is AT LEAST ONE LIKELY-CORRECT SQL among the given candidates.

Use the following criteria to judge whether a candidate is "likely-correct":
1) Intent match: entities, filters, metrics, order, and top-k behavior align with the user query.
2) Schema validity: the query uses correct tables/columns, required joins are present, and aggregations are legal.
3) Execution sanity: the exec_preview has a plausible shape/values for the query (no obvious contradictions).
4) No major red flags: units/ratios are handled reasonably, limit/order are coherent, and there are no clearly spurious tables or conditions.

You should make a balanced judgment:
- Mark `likely_has_correct=true` if at least one candidate appears reasonably correct according to the above criteria.
- Minor ambiguities are acceptable as long as the query and SQL are broadly aligned and there are no obvious fatal issues.

Special handling for the first candidate (sorted pool):
- Always inspect the first candidate carefully first.
- If multiple candidates are likely-correct, prefer the first candidate as `best_cand_idx` when it also appears likely-correct.
- If the first candidate is clearly incorrect, then consider other candidates for `best_cand_idx`.

You MUST output JSON with a SINGLE top-level key `"decision"`:

```
{
  "decision": {
    "likely_has_correct": true/false,
    "confidence": 0.0~1.0,
    "reason_tags": ["MISMATCH_INTENT","MISSING_JOIN", ...],
    "support": {
      "best_cand_idx": <int or null>,
      "notes": "≤200 chars optional"
    }
  }
}
```

Strict output rules:
- Do NOT output any other top-level keys (NO `"sampling"`, NO `"guidance"`, NO `"compat_drop_mode"`).
- Keep all fields in `"decision"` present; if unknown, use `null` or `[]` rather than omitting.
- `cand_idx` refers to ids provided with candidates (not array positions); use the given `cand_idx` integers.
- Output MUST be valid JSON, with double quotes on all keys and string values.

Figure 8: Full system prompt for the LLM agent f for resampling decision.

User Prompt in Agentic SQL Candidate Generation

You are given a Text2SQL problem.

User query
`{user_query}`

DB dialect
`{db_dialect}`

Schema (summary or DDL)
`{db_schema}`

Candidate SQLs (with id and execution preview)
Each item provides (cand_idx, SQL, and exec_preview: a few rows from a dry-run; `None` means error/missing).
The first candidate in the sorted pool is the primary candidate; pay particular attention to whether it is likely-correct.

`{items_block}`

Matching checklist (for your internal reasoning)
- Entities/filters/metrics/order/top-k extracted from user query must be reflected in the SQL.
- If metric and filter come from different tables, valid explicit JOIN is required.
- Ratios should reasonably avoid wrong integer division (e.g., using casting when appropriate).
- Avoid undocumented mappings; rely only on provided schema/metadata.

Return JSON ONLY with a single top-level key `"decision"`, following the schema in the system prompt.

Figure 9: Full input prompt for the LLM agent f for resampling decision, populated with the natural language question, schema, and candidate SQL pool.