BIRD-RL: Scaling Agentic Reinforcement Learning over Stateful Data-Centric Environments

Ge Qu[1], **Jinyang Li**[1], **Xiaohan Xu**[1], **Xiaolong Li**[1],
Bowen Qin[2], **Nan Huo**[1], **Ziwei Tang**[1], **Chenhao Ma**[3], **Reynold Cheng**[1]
[1] The University of Hong Kong [2] National University of Singapore
[3] The Chinese University of Hong Kong, Shenzhen
quge@connect.hku.hk

Abstract

Modern data-centric applications increasingly require language agents capable of operating over live, stateful database environments, where successful problem resolution depends not merely on initial SQL generation, but on reasoning about evolving database states, interacting with execution feedback, and repairing state-mutating operations. While agentic reinforcement learning (RL) has emerged as a promising paradigm for training small language models (SLMs), existing frameworks are predominantly tool-centric and struggle with data-centric environments. Specifically, applying tool-centric frameworks to stateful tasks like SQL debugging introduces severe spatial inefficiencies since proper environment isolation typically requires a separate database container per rollout. Naively reusing databases without tracking evolving states across multi-turn interactions causes trajectory contamination and cross-trajectory interference, which severely destabilizes policy updates. To overcome these bottlenecks, we introduce **BIRD-RL**, a novel four-stage agentic RL curriculum that alternately enhances foundational agent reasoning and trajectory-adaptive on-policy exploration to circumvent the instability inherently associated with direct end-to-end learning. Crucially, to support this curriculum, BIRD-RL is driven by a **Trajectory-scoped Persistent Execution Infrastructure**, which enables safe database replica reuse while preserving trajectory isolation. This design improves spatial RL efficiency by **12.8x**, successfully enabling scalable agentic RL for complex data-centric tasks without needs of Kubernettes. Empirical evaluations on the BIRD-CRITIC-SQLite benchmark demonstrate that our 7B and 14B models achieve Success Rates of 44.40% and 48.00%, respectively, showing competitive performance against strong proprietary model-based agents such as Claude-Opus-4.6 Agent (48.20%). Furthermore, BIRD-RL successfully trains unified SLMs capable of both advanced SQL debugging and generation, highlighting its broader potential for multi-objective optimization across heterogeneous, real-world data-centric tasks. For reproducibility, we release our code at https://github.com/bird-bench/BIRD-RL.

1 Introduction

Modern digital applications, from financial systems to web services, heavily rely on relational databases (RDBs) for robust data management [9, 16, 20, 49]. Structured Query Language (SQL), which functions as the definitive interface for these systems, is indispensable for data query and manipulation [5, 2]. Driven by the rapid development of large language models (LLMs) [45, 60], a broad spectrum of LLM-based SQL tasks has emerged. Text-to-SQL, which translates natural language questions into executable SQL queries, has become a representative beneficiary of this progress [24, 57, 35, 23]. While current models excel at direct SQL generation, effectively resolving

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

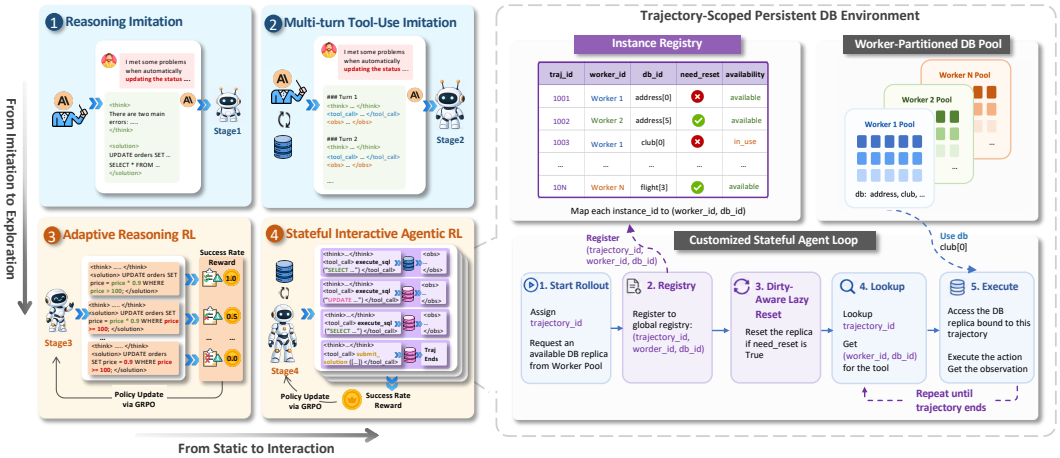


Figure 1: Overview of the BIRD-RL framework. **(Left)** The four-stage agentic RL curriculum transitioning from static reasoning imitation to interactive stateful exploration. **(Right)** The Trajectory-Scoped Persistent Execution Infrastructure, enabling scalable and isolated database replica reuse via a customized agent loop and worker-partitioned pool.

SQL issues remains a significant challenge with the evolving user demands in realistic scenarios [1, 30, 12, 51, 15]. Such SQL issue resolution often requires considerable manual effort, given the complexity of query semantics, intricate underlying schemas, and heterogeneous database operations. As a result, automating this process offers substantial practical value by improving efficiency and reducing reliance on specialized expertise [34, 62]. Automatic issue resolution inherently requires stronger agentic capabilities to conduct root cause location and repair through iterative interaction with the database environment [25, 54, 53]. Contemporary approaches to developing reliable debugging agents can be categorized into two primary paradigms: inference-time prompting [22, 53, 55] and post-training-based agent learning [25]. The former employs elaborate in-context orchestrations, in which LLMs iteratively refine intermediate reasoning decisions based on execution feedback. The latter paradigm strengthens small language models (SLMs) by internalizing these agentic capabilities directly into model parameters through supervised fine-tuning (SFT). Such agents are highly valuable in data-centric applications due to data privacy and security concerns [40]. This practical need motivates the focus of our work.

Despite its effectiveness, SFT often fails to promote robust intermediate reasoning and generalizable repair behavior due to its heavy reliance on demonstrated correction trajectories [7, 13]. Agentic reinforcement learning (RL), instead, has recently emerged as a promising paradigm for training interactive agents, enabling models to improve reasoning through autonomous exploration driven by environmental reward signals [60, 3, 61]. However, agentic RL for SQL issue resolution presents unique, domain-specific challenges. On the one hand, SQL issue resolution requires a broader action space for root-cause analysis in declarative SQL. Agents often need to verify hypotheses through stateful database operations, including data manipulation (`INSERT`, `UPDATE`, `DELETE`) and schema modification (`ALTER TABLE`) [9, 19], placing a strong demand on the underlying infrastructure to support persistent and isolated database interactions at scale. General tool-centric frameworks often manage environments via a container-per-rollout mechanism [42, 54, 27]. However, applying this to data-intensive database environments incurs severe spatial inefficiencies and prohibitive scaling bottlenecks during massive trajectory generation. On the other hand, SQL debugging is a complicated task that simultaneously demands schema-grounded reasoning and dynamic environmental interaction. Direct end-to-end learning may overwhelm the policy within an exponentially large search space, resulting in suboptimal debugging effectiveness [10].

To bridge this gap, we introduce BIRD-RL, a novel agentic RL curriculum for SQL issue solving. It decomposes agent learning into four progressive stages: reasoning imitation (Section 3.1), tool-use imitation (Section 3.2), adaptive reasoning reinforcement (Section 3.3), and interactive agentic reinforcement (Section 3.4). Such a framework decouples an otherwise densely intertwined skill set into atomic, manageable learning objectives, circumventing the instability inherently associated with direct end-to-end learning. To realize scalable training for the final stage, we further introduce a **Trajectory-Scoped Persistent Execution Infrastructure** comprising a worker-partitioned database

pool (Section 4.1) and a customized agent loop (Section 4.2), enabling isolated stateful interactions and high-concurrency efficiency via the orchestrated reuse of database replicas.

Building on BIRD-RL, we release the BIRD-TALON model family, including 7B and 14B versions. Furthermore, to demonstrate that our curriculum generalizes beyond single-objective optimization, we introduce BIRD-ZENO-7B, a unified multi-task model trained under a principled data-mixing ratio for heterogeneous SQL tasks targeting both debugging and query generation. As a resource contribution to the community, we also release SIX-GYM-SQLITE, an automatically constructed training environment that extends SIX-GYM [25] to the SQLite dialect to facilitate future research for SQL agents.

Empirical evaluations demonstrate that BIRD-TALON-7B and 14B achieve Success Rates (SR) of **44.40%** and **48.00%** on BIRD-CRITIC-SQLITE, respectively, performing competitively against strong proprietary model-based agents such as Claude-Opus-4.6 Agent (48.20%). Our trajectory-scoped infrastructure improves RL spatial efficiency by **12.8×** under the evaluated SQLite workload, enabling stable and scalable agentic training. Besides, the unified BIRD-ZENO model achieves an Execution Accuracy (EX) of **63.90%** on the BIRD development set, delivering generation performance comparable to dedicated text-to-SQL models specifically optimized with RL while maintaining its debugging proficiency. These findings establish BIRD-RL as a scalable framework for data-centric agent training, with its trajectory-scoped infrastructure paving the way for agentic RL in heavy, stateful database environments, while underscoring the potential of multi-objective training for heterogeneous real-world SQL tasks.

2 Preliminaries

2.1 Task Definition

Text-to-SQL Generation. Given a natural language question $\mathcal{Q}$, with its corresponding database schema $\mathcal{S}$, the goal of text-to-SQL is to generate the SQL query σ_{pred} that could answer the question $\mathcal{Q}$ accurately by:

$$\sigma_{pred} = f_\theta(\mathcal{Q}, \mathcal{S}), \tag{1}$$

where $f_\theta(\cdot)$ refers to the mapping function with parameters θ applied by the generator.

SQL Issue Resolution. Compared with SQL generation from scratch, SQL issue resolution is a more realistic and challenging task. The task takes as input a user-provided SQL query σ_{issue}, a natural language problem description $\mathcal{P}$ specifying the issue and intended behavior, and the database schema $\mathcal{S}$. The goal is to produce a revised query σ_{pred} that resolves the issue while preserving the user intent by:

$$\sigma_{pred} = f_\theta(\mathcal{P}, \mathcal{S}, \sigma_{issue}), \tag{2}$$

The target output σ_{pred} is expected to satisfy the underlying user intent inferred from the input triplet.

2.2 Tool-Centric Agent RL Loop

Given an SQL issue task $x = (\mathcal{P}, \mathcal{S}, \sigma_{\text{issue}})$, a standard tool-centric agent loop [42] models interaction as a history-conditioned decision process. The agent maintains a textual history h_t, initialized as $h_0 = x$, and at each step samples a structured tool invocation $a_t \in \mathcal{A}$ from policy π_θ:

$$a_t \sim \pi_\theta(\cdot \mid h_t), \quad o_t \sim f_k(\cdot \mid a_t, e_t), \quad h_{t+1} = h_t \oplus (a_t, o_t), \tag{3}$$

where f_k denotes the selected tool, o_t is the returned observation under environment state e_t, and $\oplus$ denotes history concatenation. The trajectory ends when the agent submits a final answer or reaches the maximum step budget.

This fundamentally tool-centric loop records tool feedback in the textual context while leaving the persistent environment external to the loop. For many agentic RL tasks [42, 54, 27], isolation can be approximated by assigning each rollout an independent sandbox. In data-centric tasks such as SQL issue resolution, however, this *container-per-rollout* design becomes prohibitively expensive, since each trajectory would require its own mutable replica of a potentially large database.

Reusing database replicas is therefore necessary for scalability, but remains unsafe under a purely tool-centric loop. Without explicit trajectory-to-environment binding, tool calls from the same rollout

may fail to access the intended evolving database state, while reused replicas may carry residual mutations from previous rollouts. Such state leakage contaminates execution feedback and induces cross-trajectory interference, thereby destabilizing on-policy RL. These limitations motivate the Trajectory-scoped Persistent Execution Infrastructure introduced in Section 4.

3 BIRD-RL: Efficient Agentic RL Curriculum for SQL Agent Training

Autonomous SQL issue resolution requires diagnostic reasoning, tool invocation, and feedback-guided refinement. Directly optimizing these capabilities end-to-end over a large interactive action space can lead to unstable or suboptimal policies [10]. To address this, BIRD-RL adopts a four-stage curriculum, as shown in Figure 1 (left), which contains Reasoning Imitation Section 3.1, Tool-Use Imitation (Section 3.2), Reasoning Reinforcement (Section 3.3), and Interactive Agentic Reinforcement (Section 3.4). This progressive design decomposes agent learning from static reasoning to stateful interaction, while the final stage is supported by our trajectory-scoped infrastructure for scalable and contamination-free RL.

3.1 Stage 1: Reasoning Imitation

Training Objective. As the foundation of the curriculum, this initial stage aims to establish a strong reasoning prior for schema-grounded issue solving. Formulated as a single-turn sub-task, the target output for a given input task $x = (\mathcal{P}, \mathcal{S}, \sigma_{issue})$ is defined as $y = (t, \sigma)$, where t denotes the reasoning thought, and σ is the predicted query. The policy π_θ is trained via supervised fine-tuning (SFT). The structured diagnostic reasoning capability learned in Stage 1 effectively mitigates the naive exploration (e.g., `SELECT * FROM table`) in the ultimate interactive scenarios, resulting in more efficient execution and preventing unnecessary context expansion.

Training Data Construction. To reduce human annotation cost, we use `Claude-Sonnet-4.5` as the teacher model to construct reasoning-imitation data in the SIX-GYM-SQLITE environment. Details of the environment construction and the protocol for preventing training-evaluation contamination are provided in Appendix A. For each task $x = (\mathcal{P}, \mathcal{S}, \sigma_{\text{issue}})$, the teacher generates a debugging thought t with access to the corrected query σ^*, which serves as a reference for constructing the reasoning that diagnoses the SQL errors and motivates the target repair. We pair the generated thought with the corrected query as the supervision target $y = (t, \sigma^*)$. To ensure high-quality training data, we only include (t, σ^*) pairs where the corrected SQL σ^* can be regenerated from the debugging thought t and the regenerated solution passes all test cases in $\mathcal{C}$. This produces **4,982** instances for the Reasoning Imitation stage.

3.2 Stage 2: Multi-Turn Tool-Use Imitation

Training Objective. Building on the previous stage, Stage 2 performs trajectory fine-tuning with the SQL-Act scaffold [25], where executable SQL statements serve as fine-grained actions for database inspection, hypothesis testing, and root-cause localization. Each trajectory $\tau = \{(t_i, a_i, o_i)\}_{i=1}^{n}$ interleaves the agent thought t_i, SQL action a_i, and database observation o_i. We implement SQL actions with two tools: `execute_sql` for exploration and testing, and `submit_solution` for termination. These tools keep SQL as the core action while providing an explicit stopping signal. Tool descriptions are detailed in Figure 6.

Data Construction. Following the trajectory formulation defined above, the teacher model `Claude-Sonnet-4.5` is guided to generate a verified trajectory $\tilde{\tau}$ for each input task suite x. We further employ the f-plan boosting strategy [25] to augment successful candidate trajectories. We consider the query σ submitted via `submit_solution` as the predicted query of each trajectory and apply the identical verification protocol as Stage 1 to filter $(x, \tilde{\tau})$ pairs. We ultimately collected a training set of **4,502** trajectories.

3.3 Stage 3: Adaptive Reasoning Reinforcement

RL Training Objective. Although the first two stages provide reasoning and tool-use priors, they are learned from offline imitation and may induce rigid patterns that fail to adapt to new interaction

contexts, as evidenced in Section 5.5. To bridge imitation and on-policy agentic exploration, Stage 3 introduces a reasoning-focused RL phase that encourages adaptive reasoning before multi-turn RL.

We adopt GRPO [41] for both reinforcement stages of BIRD-RL. In Stage 3, following the single-turn output format $y = (t, \sigma)$, the policy samples a group of N candidate outputs $\{y_i = (t_i, \sigma_i)\}_{i=1}^{N}$ for each input x. Each solution σ_i is evaluated by the execution-based reward defined below, and the resulting group-relative advantages are used to optimize:

$$\mathcal{J}_{\mathrm{GRPO}}(\theta) = \mathbb{E}\left[\frac{1}{N}\sum_{i=1}^{N}\min\left(\rho_i A_i,\ \mathrm{clip}(\rho_i, 1-\epsilon, 1+\epsilon)A_i\right)\right] - \beta D_{\mathrm{KL}}(\pi_\theta \,\|\, \pi_{\mathrm{ref}}), \tag{4}$$

where $\rho_i = \frac{\pi_\theta(y_i|x)}{\pi_{\theta_{\mathrm{old}}}(y_i|x)}$ is the likelihood ratio, A_i is the advantage, and π_{ref} is initialized from the Stage 2 policy.

Reward Design. Since reasoning quality is difficult to verify directly, we supervise it through the functional correctness of the generated SQL. Given the test suite $\mathcal{C}$, the reward is defined as:

$$R = \frac{1}{|\mathcal{C}|}\sum_{c\in\mathcal{C}}\mathbb{I}\big(c(\sigma) = \texttt{True}\big), \tag{5}$$

where $\mathbb{I}(\cdot)$ is the indicator function. A solution receives credit for each test case it passes. The reward corresponds to the fraction of test cases passed by the generated SQL.

3.4 Stage 4: Stateful Interactive Agentic Reinforcement

Agentic RL Training Objective. As the culmination of the training curriculum, Stage 4 aims to enable autonomous multi-turn exploration with tool augmentation through agentic RL training. For each input task x, a group of N multi-turn trajectories $\{\tau_i\}_{i=1}^{N}$ is sampled. The final solution σ_i is determined by the `submit_solution` action when available, or otherwise by the last generated query. The optimization follows the GRPO objective in Eq. (4), formulated at the trajectory level. Specifically, the likelihood ratio is computed over the full trajectory τ_i instead of a single-turn output y_i, and the reference policy π_{ref} is updated to the policy obtained from Stage 3.

Reward Design. The objective of the final stage is to produce a functionally correct solution through iterative tool-augmented interaction. While prior works [37, 17] typically augment verifiable execution rewards with elaborately designed proxy rewards, we observe that these proxy rewards can induce reward hacking in SQL debugging, as demonstrated in Section 5.6. To this end, we adhere to the execution-based reward defined in Eq.(5) assigned exclusively to the final solution, providing an unbiased signal directly aligned with the stage goal. To encourage unconstrained exploration, we apply no process-level rewards. Eliminating the step-wise credit assignment prevents bias toward specific interactions and purely aligns optimization with the final functional objective. Detailed designs of the proxy-reward variants used for comparison are provided in Appendix B.2.

4 Trajectory-Scoped Persistent Execution Infrastructure

As discussed in Section 2.2, standard tool-centric loops record tool observations in the textual history but do not explicitly bind a rollout to a persistent execution environment. This design is insufficient for stateful SQL debugging, where each trajectory must retain the database effects of its earlier actions for subsequent hypothesis testing. To support scalable stateful RL with environment isolation, BIRD-RL introduces a Trajectory-scoped Persistent Execution Infrastructure, as shown in Figure 1 (right). It consists of two main modules: a worker-partitioned database pool (Section 4.1) for reusable storage and a customized stateful agent loop 4.2 for trajectory-level environment binding.

4.1 Worker-Partitioned Database Pool

The database pool maintains a pre-allocated set of reusable database replicas to avoid repeatedly instantiating a fresh database for every rollout. Given template databases $\mathcal{D} = \{d_1, \ldots, d_N\}$, the pool maintains K working replicas for each database d_i, denoted as $\{d_i^{(1)}, \ldots, d_i^{(K)}\}$. Since replica

availability is managed locally by distributed rollout workers, replicas are partitioned across W workers:

$$\{d_i^{(1)}, \ldots, d_i^{(K)}\} = \bigcup_{w=1}^{W} \mathcal{E}_i^{(w)}, \quad \mathcal{E}_i^{(w)} \cap \mathcal{E}_i^{(w')} = \emptyset \ (w \neq w'), \tag{6}$$

where $\mathcal{E}_i^{(w)}$ denotes the worker-exclusive replica subset assigned to worker w. Each replica in the pool maintains two states, which are `is_available` for assignment availability and `need_reset` for whether restoration is needed before reuse. This design enables efficient replica reuse while preventing cross-trajectory contamination.

4.2 Customized Stateful Agent Loop

Standard tool-centric loops [42] usually dispatch each tool invocation as a request-scoped call: the execution backend receives the current tool input and returns an observation, while the request identifier only represents this individual call rather than the entire rollout. Such request-scope abstraction fails to preserve state continuity within a trajectory, which is essential for SQL debugging with state-mutating operations. We therefore make the environment management trajectory-scoped with four designs: **1) Stable Trajectory Identifier Binding:** at the start of each rollout, we assign a stable trajectory identifier, and all subsequent `execute_sql` calls within the same rollout use this identifier to retrieve the same database replica, ensuring intra-trajectory state persistence. **2) Lightweight Instance Registry:** we maintain a lightweight registry that maps each trajectory identifier to its allocated database replica, e.g., (`trajectory_id`, `worker_id`, `db_id`). This registry serves as the assignment memory between the agent loop and the database pool, routing each tool call to the intended persistent environment. **3) Deferred State Release:** the assigned replica remains bound to the trajectory until termination, rather than being released after each tool call. It is normally released upon the terminal `submit_solution` action, while an asynchronous timeout mechanism reclaims replicas from abnormal rollouts to prevent resource leakage. **4) Dirty-Aware Lazy Reset:** when a released replica has been modified, it is marked with `need_reset=True` and returned to the pool without resetting at trajectory termination. The reset is deferred until the replica is allocated to a new trajectory, which avoids termination-phase lock contention while ensuring that every new rollout starts from a clean database state. To further clarify the workflow of the customized agent loop shown in Figure 1, we present its pseudo-code in Algorithm 1.

5 Experiments

5.1 Experiment Settings

Datasets and Metrics. We evaluate BIRD-RL on heterogeneous SQL tasks covering both SQL debugging and SQL generation. For SQL debugging, we evaluate on BIRD-CRITIC-SQLITE [25], which contains 500 SQLite issues across query-like, management, and personalization categories, and BIRD-CRITIC-MULTI [25], which contains 570 issues across PostgreSQL, MySQL, SQL Server, and Oracle. For SQL generation, we evaluate on the BIRD-SQL [24] development set with 1,534 examples. We report Success Rate (SR) for SQL debugging and Execution Accuracy (EX) for SQL generation following the official metrics of each benchmark.

Implementation and Baselines. We implement BIRD-RL on Qwen2.5-Coder-7B and Qwen2.5-Coder-14B, producing the debugging-specialized BIRD-TALON models. We also train BIRD-ZENO-7B as a unified model for both SQL debugging and generation through mixed-task training. All post-training stages are implemented with VERL [42] and run on 8×H100 GPUs. Our trajectory-scoped infrastructure is integrated as a plug-and-play patch compatible with general VERL-based rollout pipelines. For baseline comparison, we consider strong general-purpose LLMs, multi-turn SQL agents, and supervised debugging agents on SQL debugging, as well as recent RL-based text-to-SQL methods on SQL generation. Full dataset statistics, training details, inference settings, and compared-method descriptions are provided in Appendix A and C.

Table 1: Success Rate (SR %) of different models on BIRD-CRITIC-SQLITE, evaluated under greedy decoding and grouped by issue category. **Bold numbers indicate the highest score in each column**, and underlined numbers indicate the second highest. "Quer." refers to query-like issues, "Mana." refers to data-management issues, and "Pers." refers to personalized-function issues.

Type	Model	Quer.	Mana.	Pers.	Overall
General-Purpose Models	GPT-5.4-Pro	44.01	38.67	39.72	42.00
	Claude-Opus-4.6	50.35	44.00	39.01	46.20
	Claude-Sonnet-4.5	46.83	36.00	34.75	41.80
	Gemini-3.1-Pro	**53.52**	44.00	**41.84**	**48.80**
	GLM-4.7	48.24	40.00	33.33	42.80
	Kimi-K2.5	46.48	37.33	35.46	42.00
	MiniMax-M2-1	38.03	30.67	32.62	35.40
	Qwen3-Coder-480B	45.77	33.33	37.59	41.60
Multi-Turn Agents	Claude-Opus-4.6	**53.52**	42.67	40.43	48.20
	Claude-Sonnet-4.5	44.01	**54.67**	37.59	43.80
	Qwen3-Coder-480B	46.48	34.67	39.01	42.60
	BIRD-FIXER-7B	37.32	26.67	29.79	33.60
	BIRD-FIXER-14B	42.96	37.33	35.46	40.00
Ours	BIRD-Talon-7B	46.13	45.33	40.43	44.40
	BIRD-Talon-14B	51.06	50.67	40.43	48.00

Table 2: Detailed comparison of BIRD-TALON with other strong baselines on BIRD-CRITIC-SQLITE. The SQL-ACT agent employs SQLs as actions. Δ denotes the absolute improvement of BIRD-TALON compared with the base model.

Model	Base	SQL-ACT	BIRD-FIXER	BIRD-TALON	Δ (%)
Qwen-2.5-Coder-7B	26.20	29.80	33.60	**44.40**	+18.20
Qwen-2.5-Coder-14B	33.80	36.40	40.00	**48.00**	+14.20

5.2 Overall Performance

Results on SQL Debugging. Table 1 and Table 2 present the performance of BIRD-TALON comparing approaches with mainstream LLMs and advanced SQL debugging agents. We can observe that:

(1) **Considerable Challenge in Stateful SQL Issue Debugging:** Even the strongest proprietary model reaches only 48.80% SR, showing that SQL issue resolution over mutable databases is far from being solved. Category-level results further reveal that the difficulty mainly comes from state-dependent issues in which general-purpose models perform better on query-like cases, with 46.65% average SR, but drop to 38.00% and 36.79% on management and personalization issues, respectively. In addition, multi-turn agentic inference improves average overall SR by only 3.87%. This limited gain suggests that inference-time interaction alone is insufficient for robust stateful SQL debugging, motivating our agentic RL framework that enables adaptive reasoning through persistent database interaction.

(2) **Superior Performance of BIRD-TALON:** As shown in Table 2, BIRD-TALON brings consistent gains across both model scales, improving the Qwen2.5-Coder-7B and 14B backbones by **18.20%** and **14.20%**, respectively. We can also observe that directly applying the SQL-ACT scaffold produces only limited gains. While supervised trajectory imitation with BIRD-FIXER improves performance further, it still falls substantially behind BIRD-TALON. This suggests that imitation alone is insufficient to capture the decision dynamics required for multi-turn SQL debugging. Instead, BIRD-RL enables the agent to adapt its reasoning and tool use through reasoning and agentic RL. Notably, BIRD-TALON-14B achieves **48.00%** overall SR, approaching the comparable performance of the Claude-Opus-4.6 agent (48.20%) while employing a much smaller open model, which is especially valuable for privacy-sensitive SQL applications.

Table 3: Comparison of database lifecycle I/O during interactive rollout execution under trajectory-level isolation. "Emp. I/O" reports the measured online database lifecycle I/O from our implementation and excludes the one-time offline pool materialization cost.

Strategy	Training-time I/O	Emp. I/O (↓)	Cost Ratio
Per-rollout Container	$O(NR\bar{S})$	2.94 TiB	12.8×
Dynamic Reusable Pool	$O((B + q/k)R\bar{S})$	236 GiB	1.0×

5.3 Analysis of Database Reuse Efficiency.

To validate our proposed pool-based container isolation strategy, we conduct an efficiency analysis of database lifecycle I/O under trajectory-level isolation. Unlike peak memory and storage capacity, which can be managed through resource provisioning and pool sizing, repeated database materialization and restoration place sustained pressure on storage bandwidth. The resulting I/O saturation can cause rollout stalls or timeouts. Such infrastructure-induced failures can distort the reward signal and destabilize on-policy optimization if recorded as zero-reward trajectories. Reducing this recurring overhead is therefore important for feasible and reliable agentic RL over resource-intensive database environments.

As shown in Table 3, the training-time I/O column presents the theoretical online I/O complexity during rollout execution. The related parameter definition and detailed derivation of the complexity terms is provided in Appendix D.2. The naive per-rollout baseline scales with the total number of generated trajectories. In contrast, the dynamic reusable pool fundamentally amortizes the database instantiation cost via cross-rollout reusability. This strategic shift leaves the online I/O dominated mainly by resetting modified replicas to preserve clean trajectory-level isolation. Benefiting from the SFT stages in our framework, which effectively regularizes the agent's exploration behavior, only **9.25%** of the practical trajectories execute state-altering queries and actually trigger a database reset. As a result, our proposed method guarantees a superior I/O efficiency compared to the naive baseline.

However, dynamic online allocation is vulnerable to abrupt memory spikes under imbalanced database workloads, which can cause capacity overflows and destabilize rollout execution. We thus apply a pre-allocated database pool to mitigate this risk. Under the evaluated SQLite setting with batch size $B = 16$ and rollout $R = 16$, our design reduces the measured online lifecycle I/O by $\mathbf{12.8}\times$ compared to per-rollout containers. The trade-off is an offline pool cost of 69 GiB, which is practically negligible relative to the terabyte-level massive online I/O savings. Notably, since this offline pool scales strictly with the number of unique databases rather than the massive training volume $\boldsymbol{N}$, its storage overhead remains predictable and highly cost-effective even under large-scale trajectory-level isolation. As the first work to investigate pool-based environment isolation for data-centric tasks, our infrastructure enables stable and scalable stateful agentic RL training for SQL debugging, paving the way for more intelligent database environment management as future work.

5.4 Results on Heterogeneous SQL Tasks.

Overall Performance. Table 4 reports the performance of BIRD-ZENO, which is trained under the data-mixed BIRD-RL scaffold, on heterogeneous SQL tasks across SQL generation and issue resolution. BIRD-ZENO-7B maintains superior debugging capabilities while achieving results comparable to strong RL-based generation-specific specialists on the BIRD-SQL development set. To our knowledge, this study is the first exploration of unified training in the SQL domain, highlighting the potential for multi-objective optimization within a single framework across real-world data-centric tasks.

Table 4: Comparison of BIRD-ZENO with strong RL-based generation-specialized methods. "Gen." = "Generation" and "Debug." = "Debugging".

Model	Gen.	Debug.
Think2SQL-7B	56.1	24.0
SQL-R1-7B	58.9	26.6
ReEX-SQL-7B	63.4	-
Reasoning-SQL-7B	64.0	-
Arctic-T2S-R1-7B	**68.9**	29.8
SQL-Trail-7B	60.1	-
BIRD-Zeno-7B (Ours)	63.9	**44.6**

Data Mixture Ratio Discussion. The data mixture for multi-task training is a complex trade-off between task-specific performance and cross-task interference [31]. To estimate a suitable mixture

Table 5: Case study illustrating the effectiveness of Stage 3, with the full trajectory shown in Figure 4.

Setting	Reasoning Trace	Action Trace
w/o Stage 3	*"... the problematic query joins budget and expense ... To get the correct total budget per event, I need to aggregate budget amounts at the event level using MAX or MIN, since each event has one budget ..."*	`SELECT ... SUM(expense.cost), MAX(budget.amount) AS total_budget ... GROUP BY budget.link_to ...`
w/ Stage 3 (Ours)	*"... there are multiple budget lines ..., and each budget line has multiple expense records ... The solution is to pre-aggregate expenses per budget line first, then join with budget ..."*	`SELECT SUM(sub_exp.total_cost), SUM(budget.amount) ... FROM (SELECT link_to_budget, SUM(cost) ... GROUP BY link_to_budget) sub_exp ...`

ratio for BIRD-ZENO, we conduct a controlled study with Qwen2.5-Coder-7B under three debugging-generation ratios. Specifically, we maintain a fixed total training set of 4,000 instances and evaluate the following settings: *debug-heavy (3:1)*, *balanced (1:1)*, and *generation-heavy (1:3)*. For efficiency, SQL generation is evaluated on BIRD-MINI [4], a mini subset of the BIRD development set.

As illustrated in Figure 2, varying the mixture ratio reveals a clear trade-off between SQL generation and issue resolution. We speculate that this asymmetry arises because the two data sources play different roles within the shared execution-grounded reward space. Generation data primarily establishes schema grounding and compositional SQL priors, whereas debugging data shapes exploration behavior for effective information acquisition from databases. In light of this trade-off, we adopt the debug-heavy ratio of 3:1 as the final mixture for BIRD-ZENO, which yields the strongest issue-resolution performance while retaining competitive SQL generation capability.

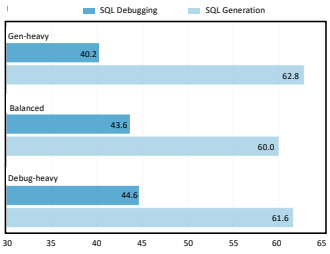


Figure 2: Effect of data mixture ratio.

5.5 Case Study: Effectiveness of the Adaptive Reasoning Reinforcement Stage

Table 5 presents a concrete case illustrating the effectiveness of our Adaptive Reasoning Reinforcement Stage (Stage 3), with the full trajectories provided in Appendix D.1 (Figure 4). In this case, both agents correctly identify the root cause as aggregation errors in the earlier step, demonstrating the effectiveness of our first two training stages. However, their solutions later diverge: 1) **w/o Stage 3:** The agent fails to incorporate feedback and instead produces ***independent*** step-wise erroneous reasoning, which is a typical side effect of SFT. 2) **w/ Stage 3:** RL enables the agent to leverage the context ***coherently*** to apply the knowledge acquired during SFT, ultimately resulting in the correct reasoning and actions. This case underscores a critical shift from memorized debugging patterns toward adaptive, evidence-grounded reasoning, proving the necessity of Stage 3.

5.6 Ablation Study

Table 6 presents our ablation study, with fine-grained breakdowns across issue categories. The removal of the training stages shown in (a)-(c) leads to obvious performance drops, indicating that the overall gain arises from the progressive curriculum. This pattern underscores the core design principle of BIRD-RL: imitation first anchors the policy in structured diagnostic reasoning and coherent tool use, while RL subsequently transforms into a more autonomous policy capable of feedback-driven exploration beyond the demonstrated trajectories. Notably, the effect of excluding the agentic RL stage (Table 6 (a)) is more pronounced on management (↓9.33) and personalization issues (↓7.10) than on query-like cases (↓0.33), suggesting that agentic RL is particularly beneficial for resolutions that require stateful hypothesis testing and iterative revision.

Table 6 (d)-(f) compares the purely execution-based reward employed in the agentic RL stage with three variants incorporating auxiliary proxy reward signals. Specifically, variant (d) applies rule-based partial-correctness signals to the submitted solution, whereas variants (e) and (f) additionally introduce trajectory-level supervision, scoring both the intermediate steps and the solution via an LLM judge and heuristic rubric criteria, respectively. Detailed reward designs are provided in

Table 6: Ablation study on the training curriculum and reward design of BIRD-RL.

TYPE	METHOD	QUER.	MANA.	PERS.	OVERALL
	BIRD-Talon-7B	46.13	45.33	40.43	**44.40**
Training Strategy	(a) *Up to Reasoning RL (Stages 1-3)*	45.77	36.00	33.33	**40.60** (↓3.80)
	(b) *Up to Tool-Use Imitation (Stages 1-2)*	45.07	33.33	30.50	**39.20** (↓5.20)
	(c) *Reasoning Imitation Only (Stage 1)*	41.55	28.00	26.24	**35.20** (↓9.20)
Reward Design	(d) *w/ Partial Correctness*	45.77	41.33	37.59	**42.80** (↓1.60)
	(e) *w/ LLM-as-a-Judge*	45.77	38.67	34.04	**41.40** (↓3.00)
	(f) *w/ Heuristic Rubric*	45.42	40.00	36.17	**42.00** (↓2.40)

Appendix B.2. The superior performance of the sparse, purely execution-based reward indicates that the reward fidelity outweighs density in SQL debugging tasks. Given the extreme error sensitivity of the task, proxy rewards are prone to reward hacking, steering the policy toward superficially plausible syntax or favorable trajectories instead of genuine functional resolution. This suggests that heuristic proxy rewards, while widely used, often require substantial expert effort to construct. To this end, BIRD-RL adopts the neat but effective execution-based design while remaining extensible to future sophisticated reward functions to further improve the upper bound of RL in data-centric tasks.

6 Related Work

LLMs for Text-to-SQL. In recent years, the application of LLMs has improved the performance of text-to-SQL to another level of intelligence [59, 52, 38, 18]. Early research primarily leveraged the in-context learning capabilities of LLMs to build text-to-SQL systems through sophisticated prompt engineering [11, 35, 11, 39, 36]. Driven by the advent of language agents, recent text-to-SQL research has pivoted toward multi-agent architectures [47, 50, 6]. A critical component emerging from these architectures is the automatic error correction mechanism. Although recent agents increasingly incorporate automatic correction loops to improve response accuracy [43, 29, 36], the distinct challenge of debugging the full spectrum of SQL operations for real-world database administration remains largely unexplored. Our work is expressly designed to bridge this gap.

RL for Reasoning. Reinforcement learning (RL) has recently emerged as a promising paradigm for improving the reasoning capabilities of LLMs, enabling models to autonomously learn effective policies with the guidance of reward signals [14, 32]. It has shown impressive success on reasoning-intensive tasks such as mathematical solving [41], code generation [26, 44], and visual reasoning [48, 46, 58]. In light of these advancements, recent research in the text-to-SQL domain has started to leverage RL-based frameworks to steer the query synthesis process [33, 56, 28, 8, 37, 17]. Our work aims to extend this line of research beyond static one-shot SQL synthesis to dynamic, stateful SQL issue resolution, where the agent must learn to diagnose and revise candidate repairs through multi-step interaction.

7 Conclusion

In this work, we introduce BIRD-RL, a four-stage agentic RL curriculum for SQL issue resolution. To enable scalable training in data-centric environments, we equip BIRD-RL with a Trajectory-scoped Persistent Execution Infrastructure that supports isolated stateful interactions through orchestrated database-replica reuse. Built on this framework, we release the BIRD-TALON model family for SQL debugging. We further introduce BIRD-ZENO, a unified multi-task model trained with a principled data-mixing strategy, to examine the generality of our curriculum beyond single-objective optimization. Together, these contributions establish BIRD-RL as a scalable framework for data-centric agent training, paving the way for agentic RL in heavy, stateful database environments and highlighting the potential of multi-objective training for real-world SQL applications.

8 Acknowledgments

We thank the anonymous reviewers and committees for their helpful comments, suggestions and organizations. Reynold Cheng, Ge Qu, Jinyang Li, Xiaohan Xu, Xiaolong Li, Nan Huo, and Ziwei Tang are supported by the Research Grant Council of Hong Kong (RGC Project HKU 17202325) and the University of Hong Kong (Project 2609260682). Ge Qu and Jinyang Li are supported by HKU Presidential PhD Scholar Programme. Ge Qu is also funded by Hong Kong PhD Fellowship Scheme.

References

[1] Alireza Ahadi, Vahid Behbood, Arto Vihavainen, Julia Prior, and Raymond Lister. Students' syntactic mistakes in writing seven different types of sql queries and its application to predicting students' success. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 401–406, 2016.

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394. ACM, 2015.

[3] Hao Bai, Yifei Zhou, Jiayi Pan, Mert Cemri, Alane Suhr, Sergey Levine, and Aviral Kumar. DigiRL: Training in-the-wild device-control agents with autonomous reinforcement learning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[4] BIRD Bench Team. BIRD-SQL Mini-Dev. `https://github.com/bird-bench/mini_dev`, 2024.

[5] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In *Proceedings of the 1974 ACM SIGMOD Workshop on Data Description, Access and Control*, pages 249–264. ACM, 1974.

[6] Saumya Chaturvedi, Aman Chadha, and Laurent Bindschaedler. SQL-of-thought: Multi-agentic text-to-SQL with guided error correction. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025.

[7] Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V Le, Sergey Levine, and Yi Ma. SFT memorizes, RL generalizes: A comparative study of foundation model post-training. In *Forty-second International Conference on Machine Learning*, 2025.

[8] Yaxun Dai, Wenxuan Xie, Xialie Zhuang, Tianyu Yang, Yiying Yang, Haiqin Yang, Yuhang Zhao, Pingfu Chao, and Wenhao Jiang. Reex-sql: Reasoning with execution-aware reinforcement learning for text-to-sql. *arXiv preprint arXiv:2505.12768*, 2025.

[9] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley / Pearson, 8th edition, 2003.

[10] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces, 2016.

[11] 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. *arXiv preprint arXiv:2308.15363*, 2023.

[12] Sneha Gathani, Peter Lim, and Leilani Battle. Debugging database queries: A survey of tools, techniques, and users. In *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, pages 1–16. ACM, 2020.

[13] Arnav Gudibande, Eric Wallace, Charlie Victor Snell, Xinyang Geng, Hao Liu, Pieter Abbeel, Sergey Levine, and Dawn Song. The false promise of imitating proprietary language models. In *The Twelfth International Conference on Learning Representations*, 2024.

[14] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

[15] Sabaat Haroon, Chris Brown, and Muhammad Ali Gulzar. Desql: Interactive debugging of SQL in data-intensive scalable computing. *Proc. ACM Softw. Eng.*, 1(FSE):767–788, 2024.

[16] Joseph M Hellerstein and Michael Stonebraker. *Readings in database systems*. MIT press, 2005.

[17] Harper Hua, Zhen Han, Zhengyuan Shen, Jeremy Lee, Patrick Guan, Qi Zhu, Sullam Jeoung, Yueyan Chen, Yunfei Bai, Shuai Wang, Vassilis Ioannidis, and Huzefa Rangwala. Sql-trail: Multi-turn reinforcement learning with interleaved feedback for text-to-sql, 2026.

[18] Nan Huo, Jinyang Li, Bowen Qin, Ge Qu, Xiaolong Li, Xiaodong Li, Chenhao Ma, and Reynold Cheng. Micro-act: Mitigate knowledge conflict in question answering via actionable self-reasoning. *arXiv preprint arXiv:2506.05278*, 2025.

[19] Nan Huo, Xiaohan Xu, Jinyang Li, Per Jacobsson, Shipei Lin, Bowen Qin, Binyuan Hui, Xiaolong Li, Ge Qu, Shuzheng Si, Linheng Han, Edward Alexander, Xintong Zhu, Rui Qin, Ruihan Yu, Yiyao Jin, Feige Zhou, Weihao Zhong, Yun Chen, Hongyu Liu, Chenhao Ma, Fatma Ozcan, Yannis Papakonstantinou, and Reynold Cheng. BIRD-INTERACT: Re-imagining text-to-SQL evaluation via lens of dynamic interactions. In *The Fourteenth International Conference on Learning Representations*, 2026.

[20] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. Enterprise data analysis and visualization: An interview study. *IEEE Trans. Vis. Comput. Graph.*, 18(12): 2917–2926, 2012.

[21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

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

[23] 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. *Proceedings of the ACM on Management of Data*, 2(3):1–28, 2024.

[24] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

[25] Jinyang Li, Xiaolong Li, Ge Qu, Per Jacobsson, Bowen Qin, Binyuan Hui, Shuzheng Si, Nan Huo, Xiaohan Xu, Yue Zhang, Ziwei Tang, Yuanshuai Li, Florensia Widjaja, Xintong Zhu, Feige Zhou, Yongfeng Huang, Yannis Papakonstantinou, Fatma Ozcan, Chenhao Ma, and Reynold Cheng. SWE-SQL: Illuminating LLM pathways to solve user SQL issues in real-world applications. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026.

[26] Jiate Liu, Yiqin Zhu, Kaiwen Xiao, QIANG FU, Xiao Han, Yang Wei, and Deheng Ye. RLTF: Reinforcement learning from unit test feedback. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856.

[27] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie

Tang. Agentbench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*, 2024.

[28] Peixian MA, Xialie Zhuang, Chengjin Xu, Xuhui Jiang, Ran Chen, and Jian Guo. SQL-r1: Training natural language to SQL reasoning model by reinforcement learning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026.

[29] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[30] Floris Miedema, Janet Spacco, and Raymond Lister. Patterns of SQL mistakes among novice programmers: An exploratory study. In *Proc. 26th ACM Conf. on Innovation and Technology in Computer Science Education (ITiCSE)*, pages 55–61. ACM, 2021. doi: 10.1145/3456565.3456622.

[31] David Mueller, Mark Dredze, and Nicholas Andrews. Multi-task transfer matters during instruction-tuning. In *Findings of the Association for Computational Linguistics: ACL 2024*, page 14880–14891. Association for Computational Linguistics, 2024.

[32] 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*, 35:27730–27744, 2022.

[33] Simone Papicchio, Simone Rossi, Luca Cagliero, and Paolo Papotti. Think2sql: Reinforce llm reasoning capabilities for text2sql. *arXiv preprint arXiv:2504.15077*, 2025.

[34] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. Self-driving database management systems. In *CIDR*, volume 4, page 1, 2017.

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

[36] 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 *The Thirteenth International Conference on Learning Representations*, 2025.

[37] Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wan, Hailong Li, Azalia Mirhoseini, Amin Saberi, and Sercan O Arik. Reasoning-SQL: Reinforcement learning with SQL tailored partial rewards for reasoning-enhanced text-to-SQL. In *Second Conference on Language Modeling*, 2025.

[38] Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, Fei Huang, and Yongbin Li. A survey on text-to-sql parsing: Concepts, methods, and future directions. In *arXiv:2208.13629*, 2022.

[39] Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-SQL generation. In *Findings of the Association for Computational Linguistics ACL 2024*. Association for Computational Linguistics, August 2024.

[40] 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 *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025.

[41] 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.

[42] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, 2025.

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

[44] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856.

[45] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

[46] Zhiqing Sun, Sheng Shen, Shengcao Cao, Haotian Liu, Chunyuan Li, Yikang Shen, Chuang Gan, Liangyan Gui, Yu-Xiong Wang, Yiming Yang, et al. Aligning large multimodal models with factually augmented rlhf. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13088–13110, 2024.

[47] 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.

[48] Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

[49] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 996–1005. IEEE Computer Society, 2010.

[50] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, et al. Mac-sql: A multi-agent collaborative framework for text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557, 2025.

[51] Zuozhi Wang, Avinash Kumar, Shengquan Ni, and Chen Li. Demonstration of interactive runtime debugging of distributed dataflows in texera. *Proc. VLDB Endow.*, 13(12):2953–2956, 2020.

[52] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA): 1–26, 2017.

[53] John Yang, Akshara Prabhakar, Karthik R Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

[54] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[55] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

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

[57] 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 *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, page 3911–3921, 2018.

[58] Tianyu Yu, Haoye Zhang, Qiming Li, Qixin Xu, Yuan Yao, Da Chen, Xiaoman Lu, Ganqu Cui, Yunkai Dang, Taiwen He, et al. Rlaif-v: Open-source ai feedback leads to super gpt-4v trustworthiness. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 19985–19995, 2025.

[59] John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, pages 1050–1055, 1996.

[60] Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

[61] Hao Zhang, Mingjie Liu, Shaokun Zhang, Songyang Han, Jian Hu, Zhenghui Jin, Yuchi Zhang, Shizhe Diao, Ximing Lu, Binfeng Xu, Zhiding Yu, Jan Kautz, and Yi Dong. Prorl agent: Rollout-as-a-service for rl training of multi-turn llm agents, 2026.

[62] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. D-bot: Database diagnosis system using large language models. *Proc. VLDB Endow.*, 17(10):2514–2527, June 2024. ISSN 2150-8097.

A SIX-GYM-SQLITE: Training Environment Construction

Training LLMs as autonomous agents for complex tasks fundamentally requires an integrated environment that combines high-quality task suites, executable environments, and rigorous test verification. To efficiently support the massive trial-and-error exploration inherent in optimizing these agents via RL, with server-based database systems such as PostgreSQL and MySQL, the lightweight, in-process nature of SQLite eliminates the client-server overhead of engines like PostgreSQL, enabling the high-throughput parallel rollouts essential for large-scale SQL agent training. While the existing SQL debugging benchmark SWE-SQL has introduced SIX-GYM environment for PostgreSQL, it lacks a dedicated training environment specifically constructed for SQLite. To address this gap, we contribute SIX-GYM-SQLite as a practical resource extension of the prior work, employing it as the underlying sandbox to automatically synthesize the stage-specific training data for BIRD-RL.

A.1 SIX-GYM-SQLite

Following the SQL-Rewind methodology introduced in SWE-SQL, we construct SIX-GYM-SQLite by adapting the original SIX-GYM training environment to SQLite. The resulting task suite combines instances adapted from the original SIX-GYM task suites with additional instances synthesized from newly collected Stack Overflow posts. We retain 5,000 high-quality instances based on answer quality, SQL complexity, and taxonomic coverage.

To prevent training–evaluation contamination, the SQLite adaptation preserves the database- and source-level separation established by the SWE-SQL construction protocol. Specifically, SIX-GYM-SQLite and BIRD-CRITIC-SQLite retain the database schemas of their respective counterparts in SWE-SQL, which uses disjoint database domains and instances for training and evaluation. At the source level, the inherited construction protocol excludes Stack Overflow posts used in BIRD-CRITIC from the training source pool by post ID prior to synthesis. During subsequent supervision generation for Stages 1 and 2, the teacher model is provided only with training instances and their reference solutions, without access to evaluation instances.

Furthermore, we exhaustively compared all $500 \times 5{,}000$ train–test pairs using a dual-view TF-IDF audit over normalized issue descriptions and buggy SQL. A pair was flagged as a high-confidence near-duplicate only when the same training instance exceeded 0.85 text similarity and 0.90 SQL similarity. Only 3 of 500 tasks (0.6%) are highly similar to training tasks, and all three are permission-privilege-related tasks, a category that is inherently shared, since privilege management (e.g., `GRANT/REVOKE` and access-control patterns) is core database knowledge that neither training nor evaluation can meaningfully exclude. Excluding these three flagged evaluation tasks leaves our main conclusions unchanged.

Table 7 summarizes the overall data characteristics, including task categories, token lengths, evaluation scripts, and database coverage. Table 8 further breaks down query, issue SQL, solution SQL, and test-case lengths by task category. Figure 3 visualizes the distribution of SQL usage categories and database domains, showing that SIX-GYM-SQLite covers diverse SQL operations and application contexts.

B Methodology Details

B.1 Workflow Orchestration of the Customized Stateful Agent Loop

Algorithm 1 presents the pseudo-code of the customized stateful agent loop, illustrating how trajectory-level database binding, dirty-aware lazy reset, and replica release are orchestrated during rollout execution.

B.2 Auxiliary Proxy Reward Designs

In this section, we detail the auxiliary proxy-reward variants used in the ablation study. These variants augment the final execution-based reward with additional signals intended to provide denser supervision during agentic RL. Specifically, we consider three representative designs: partial-correctness rewards, LLM-as-a-judge trajectory scoring, and heuristic rubric-based rewards. All these rewards

Table 7: Data characteristics of SIX-GYM-SQLite. Length statistics are reported as mean/max token counts using the `cl100k_base` tokenizer.

Statistic	SIX-GYM-SQLite
Total Issues	5,000
# of query-like issues	3,094
# of management issues	1,087
# of personalization issues	819
user query length (mean/max)	162.55/917
issue SQL length (mean/max)	85.50/898
solution SQL length (mean/max)	97.66/1,976
test-case length (mean/max)	221.38/1,601
# distinct test cases	2,477
# preprocess SQLs	3,512
# clean_up SQLs	969
# multi-SQL issue instances	454
# database domains	13

Table 8: SIX-GYM-SQLite statistics grouped by task category. Length statistics are reported as mean/max token counts.

Category	Count	User Query	Issue SQL	Solution SQL	Test Cases
Query	3,094	183.93/917	86.76/751	96.22/1,976	143.74/1,601
Management	1,087	120.20/535	73.51/802	83.84/802	446.16/1,085
Personalization	819	138.01/421	96.63/898	121.40/1,041	216.33/1,418
Overall	5,000	162.55/917	85.50/898	97.66/1,976	221.38/1,601

are used as auxiliary shaping signals on top of the execution-based reward. They are activated only for trajectories whose execution test pass rate is zero, so that any trajectory passing at least one test case is still primarily optimized by the execution-based reward.

Partial Correctness Reward. For zero-pass trajectories, we provide two lightweight partial signals: one for syntactic correctness and one for schema linking. The first signal rewards SQL that is syntactically valid and executable, while the second measures whether the generated SQL refers to schema elements consistent with the gold SQL.

Formally, the additional partial reward is defined as:

$$\boldsymbol{R_{\text{partial}} = 0.1 \times \mathbb{I}_{\text{exec}} + 0.1 \times F_1(s_p, s_g)} \tag{7}$$

where $\mathbb{I}_{\text{exec}}$ indicates whether the submitted SQL is syntactically valid and executable. s_p and s_g denote the sets of schema items appearing in the predicted and gold SQL queries, respectively. The schema-linking score is computed as:

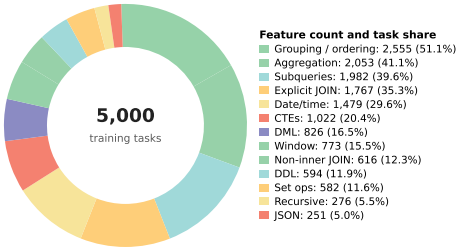
$$\boldsymbol{F_1(s_p, s_g) = \frac{2 \times P(s_p, s_g) \times R(s_p, s_g)}{P(s_p, s_g) + R(s_p, s_g)}}, \tag{8}$$

with

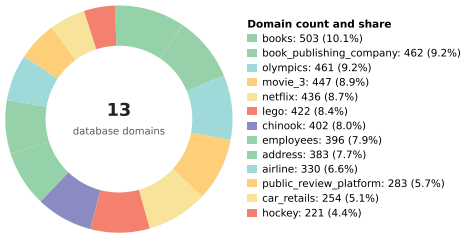
$$\boldsymbol{P(s_p, s_g) = \frac{|s_p \cap s_g|}{|s_p|}, \quad R(s_p, s_g) = \frac{|s_p \cap s_g|}{|s_g|}}. \tag{9}$$

LLM-as-a-judge trajectory scoring The LLM-as-a-Judge reward uses strong proprietary LLMs as the judge model to score both the debugging trajectory and the final SQL solution using the prompt shown in Figure 13 and 14. The judge assigns a trajectory quality score $q_{\text{traj}} \in [0, 5]$ and a solution quality score $q_{\text{sol}} \in [0, 10]$, with the maximum total score denoted as $Q_{\max} = 15$. Since these trajectories have already failed execution tests, the auxiliary final score is computed by:

$$\boldsymbol{R_{\text{judge}} = 0.2 \times \frac{q_{\text{traj}} + q_{\text{sol}}}{Q_{max}}}. \tag{10}$$



(a) SQL usage categories. Categories are non-exclusive; percentages indicate the share of tasks containing each feature.



(b) Database domains. SIX-GYM-SQLite spans 5,000 tasks across 13 SQLite database domains.

Figure 3: Statistics of SIX-GYM-SQLite training data, covering SQL feature usage and database-domain distribution.

Heuristic Rubric-Based Reward. The heuristic rubric-based reward consists of two criteria: 1) whether the submitted SQL executes successfully, and 2) whether the model identifies known bugs in its reasoning. The bug-identification criterion is computed by matching the model's reasoning against a pre-extracted bug checklist with an LLM verifier. The reward is then computed as

$$R_{\mathbf{rubric}} = 0.1 \times \mathbb{I}_{\mathbf{exec}} + 0.1 \times \frac{N_{\mathbf{bug}}^{\mathbf{id}}}{N_{\mathbf{bug}}}. \tag{11}$$

Here, $\mathbb{I}_{\mathbf{exec}}$ indicates whether the submitted SQL executes without syntax errors, $N_{\mathbf{bug}}$ is the number of known bugs in the checklist, and $N_{\mathbf{bug}}^{\mathbf{id}}$ is the number of bugs identified by the model. The prompts used for bug extraction and verification in this process are presented in Figure 15 and 16.

C Experiment Details

This section summarizes the implementation details, and compared methods used in our experiments.

C.1 Alias of LLMs

The following aliases are used for the models in this work:

- **GPT-5.4-Pro**: `gpt-5.4-pro`
- **Claude-Opus-4.6**: `claude-opus-4.6`
- **Claude-Sonnet-4.5**: `claude-sonnet-4-5-20250929`
- **Gemini-3.1-Pro**: `gemini-3.1-pro-preview`
- **GLM-4.7**: `glm-4.7`
- **Kimi-K2.5**: `kimi-k2.5`
- **MiniMax-M2-1**: `minimax-m2-1`
- **Qwen3-Coder-480B**: `Qwen3-Coder-480B`

All open-source models are downloaded from Hugging Face[1]:

- **Qwen-Coder**: `Qwen2.5-Coder-7B-Instruct`, `Qwen2.5-Coder-14B-Instruct`
- **SQL generation baselines**: `Think2SQL-7B`, `SQL-R1-7B`, `Arctic-Text2SQL-R1-7B`

[1] `https://huggingface.co/`

Algorithm 1 Workflow Orchestration Pseudo Code

Require: Task instance $\boldsymbol{x}$, target database template $\boldsymbol{d}$, worker-partitioned DB pool $\mathcal{P}$
Ensure: Final submitted solution or timeout termination

Generate a unique trajectory identifier $\mathbf{traj_id}$
Request an available replica $\boldsymbol{r}$ of database $\boldsymbol{d}$ from the local worker pool $\mathcal{P}$
if no replica is immediately available **then**
    Wait in the allocation queue until a replica is released or timeout occurs
end if
if allocation timeout occurs **then**
    return timeout failure
end if
Set $\boldsymbol{r}$.`is_available` ← `False`
Register $(\mathbf{traj_id}, \mathbf{worker_id}, \mathbf{db_id})$ in the stateful agent loop
if $\boldsymbol{r}$.`need_reset` = `True` **then**
    Restore $\boldsymbol{r}$ from the clean database template
    Set $\boldsymbol{r}$.`need_reset` ← `False`
end if
while the trajectory has not terminated **do**
    The agent generates an action $\boldsymbol{a}$
    if $\boldsymbol{a}$ is `execute_sql` **then**
        Lookup $(\mathbf{worker_id}, \mathbf{db_id})$ by $\mathbf{traj_id}$ in the stateful agent loop
        Execute the SQL action on the trajectory-bound replica $\boldsymbol{r}$
        Return the database observation to the agent
        if the SQL action mutates the database state **then**
            Set $\boldsymbol{r}$.`need_reset` ← `True`
        end if
    else if $\boldsymbol{a}$ is `submit_solution` **then**
        Record the submitted solution
        Break
    end if
end while
Release $\boldsymbol{r}$ back to the local worker pool $\mathcal{P}$
Set $\boldsymbol{r}$.`is_available` ← `True`
return submitted solution or timeout result

C.2 Model Implementation Details

We post-train all our models using the VERL framework [42] (v0.7.0.dev0)[2] and vLLM [21] (v0.11.0)[3]. All experiments are conducted on 8×H100 GPUs with 80GB memory. Our training pipeline has four stages. The first two stages are supervised fine-tuning: Stage 1 trains single-turn SQL reasoning for 3 epochs with batch size 16, maximum length 8192, and learning rate 1e-5; Stage 2 trains multi-turn tool-use trajectories for 3 epochs with batch size 16, maximum length 16000, and learning rate 1e-5. The last two stages use GRPO [41]. Stage 3 performs single-turn reasoning RL with rollout size 16, batch size 16, PPO mini-batch size 16, learning rate 1e-6, KL coefficient 0.01, maximum prompt length 15000, and maximum response length 5000. Stage 4 performs multi-turn agentic RL with asynchronous vLLM rollouts up to 5 assistant turns. Unless otherwise specified, it follows the same RL hyperparameters as Stage 3, except that the rollout size is set to 4. The specific training hours for each backbone model are shown in Table 9. During inference, we use vLLM with greedy decoding (temperature 0) and a maximum input length of 15000 tokens. We report experimental results as the average of five repeated trials.

For proprietary-model inference, we use the official API providers, including OpenAI (`https://openai.com/`), Anthropic (`https://www.anthropic.com/`), and Google (`https://gemini.google.com/`). All open-source models are downloaded from Hug-

[2] `https://verl.readthedocs.io/en/latest/index.html`
[3] `https://docs.vllm.ai/en/latest`

Table 9: GPU hours spent to train each backbone model.

Model	GPU Hours
BIRD-Talon-7B	93.38
BIRD-Zeno-7B	80.36
BIRD-Talon-14B	152.75

ging Face[4]; the Qwen-Coder backbones refer to `Qwen2.5-Coder-7B-Instruct` and `Qwen2.5-Coder-14B-Instruct`.

C.3 Compared Methods

This section provides additional details of the compared methods reported in Tables 1, 2, and 4.

SQL debugging baselines. The General-Purpose Models group evaluates frontier proprietary models and strong open-weight models under a single-turn prompt-based SQL repair setting. Given the user issue description, database schema, and buggy SQL query, the model is directly prompted to produce the corrected SQL solution without interacting with the database during inference. This setting follows the baseline SQL debugging prompt shown in Figure 10. We include GPT-5.4-Pro, Claude-Opus-4.6, Claude-Sonnet-4.5, Gemini-3.1-Pro, GLM-4.7, Kimi-K2.5, MiniMax-M2-1, and Qwen3-Coder-480B to establish broad baselines on BIRD-CRITIC-SQLITE.

The Multi-Turn Agents group evaluates models with database interaction through the SQL-ACT scaffold. Following the ReAct paradigm [55], SQL-ACT interleaves reasoning, SQL actions, and database observations over multiple turns. At each step, the model may execute an arbitrary SQL statement against the database environment and use the returned observation to refine its diagnosis. The trajectory continues until the model submits a final repaired SQL solution. Compared with tool-based agents whose action space is restricted to predefined operations such as schema inspection or sample retrieval, SQL-ACT treats executable SQL itself as the action space, enabling more flexible debugging strategies. The system and user prompts used by this scaffold are shown in Figures 6 and 7; the same scaffold is used for BIRD-TALON inference. For BIRD-FIXER-7B and BIRD-FIXER-14B, we follow the supervised agent fine-tuning pipeline from the work [25] to reproduce the training trajectories and fine-tune the models Qwen2.5-Coder-7B-Instruct and Qwen2.5-Coder-14B-Instruct on SIX-GYM-SQLITE (Appendix A).

In Table 1, we instantiate the multi-turn agent setting with Claude-Opus-4.6, Claude-Sonnet-4.5, and Qwen3-Coder-480B. In Table 2, we additionally report SQL-ACT results by directly placing the corresponding Qwen2.5-Coder base model into the same interactive scaffold, without BIRD-RL training.

SQL generation baselines. We compare BIRD-ZENO with recent RL-based text-to-SQL methods that optimize SQL generation through execution feedback, partial rewards, or multi-turn interaction. Think2SQL [33] studies reinforcement learning with verifiable rewards for text-to-SQL reasoning It introduces execution-guided dense rewards to provide more granular feedback than binary execution correctness. SQL-R1 [28] trains natural-language-to-SQL reasoning models with a cold-start SFT phase followed by reinforcement learning on complex examples. Its reward design is tailored to NL2SQL generation and uses database feedback to encourage exploration beyond fixed supervised generation patterns. ReEx-SQL [8] introduces execution-aware reasoning for text-to-SQL by interleaving exploratory SQL execution with model reasoning during generation. It further uses execution-informed rewards and decoding strategies to help the model revise reasoning based on database feedback. Reasoning-SQL [37] applies GRPO with SQL-tailored partial rewards to mitigate the sparsity of binary execution rewards. Its reward components include signals such as schema linking, syntax validity, similarity, and AI feedback to encourage reasoning-enhanced SQL generation. Arctic-T2S-R1 [56] emphasizes simple end-task rewards for text-to-SQL RL, relying mainly on execution correctness and syntax validity instead of complex intermediate supervision. SQL-Trail [17] frames text-to-SQL as a multi-turn RL problem with interleaved database feedback.

[4] `https://huggingface.co/`

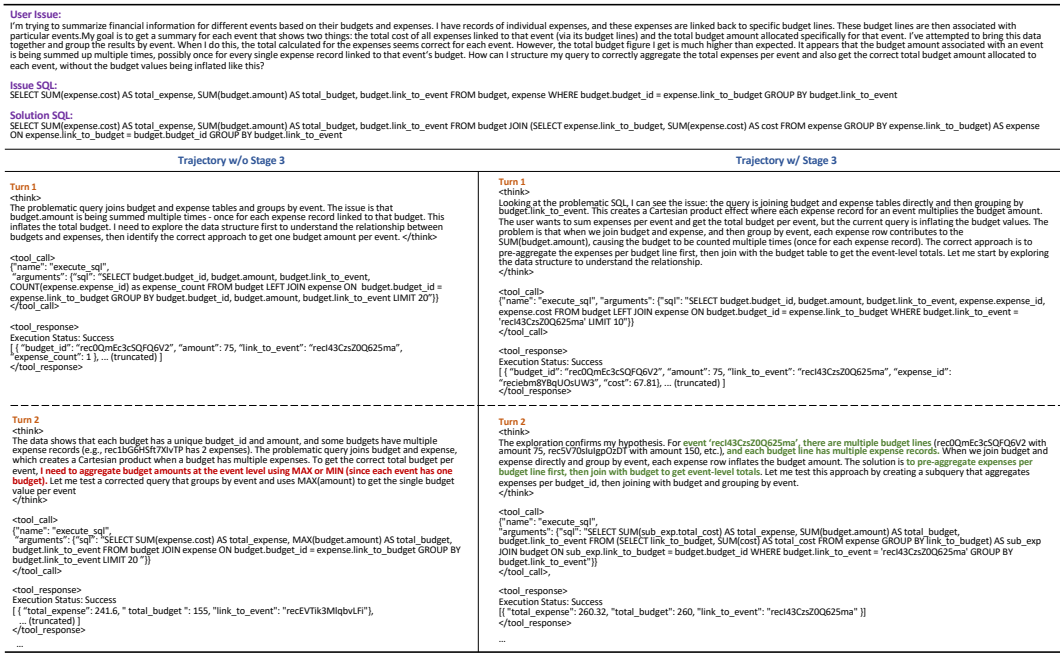


Figure 4: Case study comparing debugging trajectories with and without Stage 3 adaptive reasoning reinforcement.

It uses iterative interaction, adaptive turn allocation, and rewards that balance SQL correctness with efficient exploration.

D More Analysis

D.1 Case Study for Stage 3

Figure 4 shows a representative SQL debugging case where both agents identify the join-induced budget inflation issue, but diverge in the repair strategy. Without Stage 3, the agent incorrectly assumes that each event has only one budget and uses `MAX(budget.amount)`, leading to an erroneous fix. With Stage 3, the agent correctly infers that an event may contain multiple budget lines and verifies a repair that pre-aggregates expenses per budget line before joining with the budget table. This illustrates how Stage 3 improves adaptive, evidence-grounded reasoning beyond the imitation-learned trajectory patterns.

D.2 Comparison of Training-Time Database Lifecycle I/O

Table 3 formalizes the database lifecycle I/O overhead required to maintain trajectory-level isolation during interactive rollout execution. The naive per-rollout container baseline instantiates a fresh database for every generated trajectory. Consequently, this approach incurs an I/O cost proportional to the total number of rollout trajectories, resulting in the I/O complexity of $\boldsymbol{O}(\boldsymbol{NR}\bar{\boldsymbol{S}})$, where $\boldsymbol{N}$ is the number of training tasks, $\boldsymbol{R}$ is the number of rollout trajectories per task, and $\bar{\boldsymbol{S}}$ is the average database size.

To circumvent this exhaustive overhead, we propose a **dynamic reusable database pool** that fundamentally alters the database lifecycle for data-centric tasks. During interactive rollout execution, the pool continuously tracks the operation state, including the availability and modification status, of each database replica. A trajectory first tries to acquire an available clean replica for its target database. Under an ideal dynamic allocation scheme, a new replica is instantiated online strictly when the concurrent demand exceeds the capacity of existing available replicas. Upon trajectory completion, the leased replica is immediately released back to the pool, avoiding any naive re-instantiation. Crucially, to handle state-changing queries, the pool employs a *dirty-aware lazy reset* policy. Once a

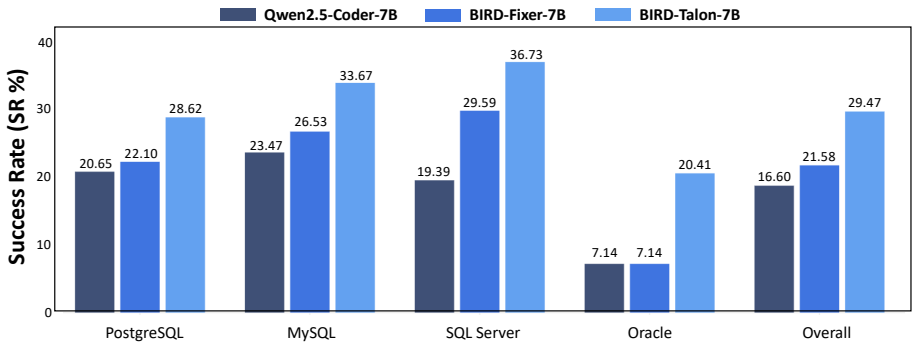


Figure 5: The success rate of BIRD-TALON on unseen dialects shows its generalization ability.

replica is modified, it is not eagerly restored upon release. Instead, the reset operation is deferred until the replica is leased again for a new trajectory.

To formalize the theoretical I/O reduction over the baseline, we decompose the dynamic pool's training-time overhead into two analytical components: 1) **Bounded Instantiation Overhead**: Assuming database requests follow a uniform access distribution across unique databases within a training batch of size $\boldsymbol{B}$, the dynamic pool size is upper-bounded by the maximum concurrency, $\boldsymbol{B \times R}$, limiting the replica creation I/O to $\boldsymbol{O(BR\bar{S})}$. In practical large-scale RL settings (e.g., optimizing a 14B parameter model), GPU memory constraints dictate a small, fixed batch size (typically $\boldsymbol{B = 16}$). Consequently, this guarantees that $\boldsymbol{B \ll N}$. 2) **Regularized Reset Overhead**: While the dynamic pool eliminates redundant materialization, it unavoidably introduces state-reset operations for modified replicas. We quantify such overhead using an example-effective count $\boldsymbol{q = Q/R}$, where $\boldsymbol{Q}$ denotes the total number of rollout trajectories that actually trigger the reset action during training across all $\boldsymbol{NR}$ rollouts. Let $\boldsymbol{k \geq 1}$ denote the empirical I/O reduction factor of an in-place reset relative to full copy-delete materialization. In database systems, targeted state restoration (e.g., table-level truncation or snapshot rollbacks) is considered cheaper than a full physical instantiation. Consequently, the specific value of $\boldsymbol{k}$ depends on the underlying reset optimization deployed. The reset complexity is thus bounded by $\boldsymbol{O(qR\bar{S}/k)}$. Notably, $\boldsymbol{q}$ is intrinsically manageable. The preceding Supervised Fine-Tuning (SFT) phase of our framework effectively regularizes the agent's exploration behavior, preventing arbitrary database mutations. Empirically, only about $\boldsymbol{9.25\%}$ of the practical rollouts execute state-altering exploitation in our experiment. Combining both components, the total lifecycle overhead of the dynamic pool evaluates to $\boldsymbol{O((B + q/k)R\bar{S})}$, where $\boldsymbol{B + q/k \ll N}$. This establishes that the dynamic pool mechanism achieves vastly superior spatial and I/O efficiency compared to the naive per-rollout baseline.

While the bounded creation overhead holds under a steady-state assumption, real-world RL exploration frequently encounters a highly imbalanced distribution. Concurrent rollouts can heavily skew towards hot databases, triggering sudden capacity overflows and allocation bottlenecks that destabilize the training pipeline. To systematically amortize this risk upfront, our actual implementation replaces dynamic online creation with a pre-materialized database pool. By shifting the database instantiation entirely offline, we decouple resource allocation from the rollout path, eliminating the risk of memory thrashing and I/O contention. Consequently, as rollouts acquire replicas via constant-time ($\boldsymbol{O(1)}$) pool lookups, the online training I/O is dominated by the predictable reset operations with complexity $\boldsymbol{O(qR\bar{S}/k)}$.

In our practical setting ($\boldsymbol{B = 16, R = 16}$), we pre-reserve $\boldsymbol{BR/2}$ replicas per database in the pool. Empirical observations confirm that this capacity is sufficient to avoid pool exhaustion. As shown in Table 3, our implementation reduces measured online lifecycle I/O by $\boldsymbol{12.8\times}$ compared to per-rollout containers. This comes with a one-time pre-materialized storage cost of 69 GiB as a trade-off, which is excluded from the online I/O reported in the table. Such a trade-off is practically negligible relative to the terabyte-level massive online I/O savings. Notably, since this offline pool scales with the number of unique databases rather than the training volume N, its storage cost remains predictable and amortizable, enabling stable large-scale trajectory-level isolation.

D.3 Generalization for Different SQL Dialects

Figure 5 illustrates BIRD-TALON's robust generalization on unseen dialects, although it is trained only on SQLite tasks within SIX-GYM-SQLite. This can be attributed to BIRD-RL strengthening the model with a more general logic of debugging reasoning through environmental interaction, rather than fitting SQLite-specific surface patterns, while preserving the model's pretrained knowledge of dialect variation. Consequently, BIRD-TALON exhibits strong cross-dialect reasoning without extra training, highlighting its potential in realistic database implementation scenarios.

D.4 Test-time Scaling Analysis

Although BIRD-TALON is evaluated under greedy decoding in the main results, we further study the potential of BIRD-TALON-14B under increased test-time sampling. We sample up to 16 independent trajectories using the same decoding configuration, temperature 0.7 and top-p 0.95, and report oracle pass@k in Table 10.

Table 10: Test-time scaling results of BIRD-TALON-14B on BIRD-CRITIC-SQLITE.

Model	pass@1	pass@4	pass@8	pass@16
BIRD-Talon-14B	46.40	57.40	60.60	65.00

As shown in Table 10, BIRD-TALON-14B benefits consistently from additional samples, indicating that test-time computation can expose correct solutions missed by a single rollout. This suggests that the learned agentic policy provides a useful candidate solution space for future verifier- or reranker-based scaling.

E Limitations and Future Work

This work only focuses on SQL tasks as a representative class of stateful data-centric environments. A natural future direction is to extend BIRD-RL to broader high-value data science tasks, such as data cleaning, transformation, workflow construction, and analytical decision-making, where agents must similarly reason over persistent states, execution feedback, even on top of large data.

In addition, BIRD-RL currently adopts execution-based rewards to ensure faithful alignment with functional correctness. Future work may explore hybrid reward designs that combine execution signals with semantics-aware verification, while explicitly addressing reward hacking to preserve reliable optimization behavior in complex data-centric settings. This can further advance upper bound of our framework.

F Ethics Statement

In this work, although our method involves data synthesis, all data are derived from databases and resources that are publicly available, ensuring transparency and reproducibility. We take care to avoid sensitive or personally identifiable information, and the generated outputs are SQL queries rather than open-ended natural language, reducing the risk of biased or harmful textual content.

G Broader Impact

This work aims to advance scalable agentic RL for data-centric environments, where agents interact with live, stateful databases rather than static textual contexts. The proposed trajectory-scoped infrastructure may benefit future research on database agents by reducing the cost of isolated rollout execution and making RL training more practical for heavy database workloads. In the long term, such techniques could help build more capable SQL assistants that reduce manual debugging effort, improve developer productivity, and support privacy-sensitive applications through stronger open-source models that can be deployed locally.

At the same time, beyond improving SQL issue resolution, our method points toward a future paradigm of general-purpose SQL agents. By training over heterogeneous SQL tasks, BIRD-RL

makes it possible to move beyond narrowly specialized models toward unified agents that can perform query generation, issue diagnosis, execution-based verification, and interactive repair within a single framework. Such agents could serve as a more general interface for real-world database applications, where users often need end-to-end assistance across multiple stages of database interaction rather than isolated text-to-SQL generation.

H Prompts and Examples

This section presents the prompts used in our experiments and example trajectory. Figures 6–9 show the system and user prompts for SQL debugging and SQL generation tasks. Figure 10 presents the baseline SQL debugging prompt. Figures 11 and 12 show the prompts used for Stage 1 reasoning imitation data construction and verification. Figures 13–16 show the system and user prompts for getting heuristic rubric reward and LLM-as-judge reward. Figure 17 illustrates an example trajectory produced by BIRD-TALON when solving a SQL debugging task.

Inference prompt for SQL Debugging (system prompt)

You are an expert SQL query writer. Your task is to translate natural language questions into correct SQL queries.
You will be provided with:
1. A database schema with sample data
2. Column descriptions
3. A natural language question
4. Evidence/hints about domain-specific terms

Your Objective
Generate the correct SQL query by:
- Analyzing the question to understand what data is needed
- Using tools to explore the database schema and sample data
- Testing candidate queries to validate your approach
- Submitting the final correct SQL query

Available Tools
<tools>
```
[
  {
    "type": "function",
    "function": {
      "name": "execute_sql",
      "description": "Execute one SQL query for exploration/testing. Returns results but does not judge correctness.",
      "parameters": {
        "type": "object",
        "properties": {
          "sql": {"type": "string", "description": "A single SQL query."}
        },
        "required": ["sql"] }}},
  {
    "type": "function",
    "function": {
      "name": "submit_solution",
      "description": "Submit the final SQL solution and end the task.",
      "parameters": {
        "type": "object",
        "properties": {
          "sql_list": {
            "type": "array",
            "items": {"type": "string"},
            "description": "Complete final SQL statement(s)."
          }},
        "required": ["sql_list"] }}}
]
```
</tools>

How to Call Tools
To invoke a tool, the output must be in the following format:
1. Open the tool call tag: <tool_call>
2. Provide a valid JSON object with exactly two keys:
- "name": string - the name of the function to call ("execute_sql" or "submit_solution")
- "arguments": object - a JSON object containing the parameters for that tool
3. Close the tool call tag: </tool_call>
Format examples:
For testing/exploration:
<tool_call>{{"name": "execute_sql", "arguments": {{"sql": "SELECT * FROM table LIMIT 5"}}}}</tool_call>
For final submission:
<tool_call>{{"name": "submit_solution", "arguments": {{"sql_list": ["SELECT col FROM table WHERE condition"]}}}}</tool_call>

Critical Format Requirements
Your trajectory MUST follow this exact structure for EVERY turn:
1. **<think> tags**: Concise, high-level reasoning about what you're doing and why. Do NOT include SQL code in thoughts.
2. **Action**: After thinking, take ONE action by calling a tool:
- Use execute_sql for exploration and testing
- Use submit_solution for final submission (ends trajectory)
3. **Turn limit**: Maximum {max_turns} turns. Each turn: <think> + <tool_call>. **CRITICAL**: If you don't call submit_solution in the first {prev_turns} turns, the last turn MUST be submit_solution (forced submission).

Figure 6: The inference prompt used for SQL debugging task (system prompt).

Inference prompt for SQL Debugging (user prompt)

Problem Description
{query}

Database Schema
{schema}

Problematic SQL
The following SQL query contains errors and does not correctly satisfy the user's requirements:

```sql
{issue_sql}
```

Your task: Provide the next single turn of the debugging trajectory. This means:
1. One <thought> block explaining your reasoning for the next action
2. Followed by ONE <tool_call>: either execute_sql to explore/test OR submit_solution for the final answer

Generate only the next turn. Maximum {max_turns} turns allowed. Review the conversation history to determine your current turn. **If this is the final turn, you MUST call submit_solution.**

Note: If conversation history is provided below (previous thoughts, tool calls, and tool responses), continue from where the debugging left off. Analyze the previous results and determine the next logical action. If this is the first turn (no history), start by analyzing the problem and deciding on an initial exploration strategy.

Figure 7: The inference prompt used for SQL debugging task (user prompt).

Inference prompt for SQL Generation (system prompt)

You are an expert SQL query writer. Your task is to translate natural language questions into correct SQL queries.
You will be provided with:
1. A database schema with sample data
2. Column descriptions
3. A natural language question
4. Evidence/hints about domain-specific terms

Your Objective
Generate the correct SQL query by:
- Analyzing the question to understand what data is needed
- Using tools to explore the database schema and sample data
- Testing candidate queries to validate your approach
- Submitting the final correct SQL query

Available Tools

<tools>
[
 {
  "type": "function",
  "function": {
   "name": "execute_sql",
   "description": "Execute one SQL query for exploration/testing. Returns results but does not judge correctness.",
   "parameters": {
    "type": "object",
    "properties": {"sql": {"type": "string", "description": "A single SQL query."}},
    “required”: [“sql”] }}},
 {
  "type": "function",
  "function": {
   "name": "submit_solution",
   "description": "Submit the final SQL solution and end the task.",
   "parameters": {
    "type": "object",
    "properties": {
     “sql_list”: {“type”: “array”, “items”: {“type”: “string”}, "description": "Complete final SQL statement(s)."
     }},
    “required”: [“sql_list”] }}}
]
</tools>

How to Call Tools
To invoke a tool, the output must be in the following format:
1. Open the tool call tag: <tool_call>
2. Provide a valid JSON object with exactly two keys:
- "name": string - the name of the function to call ("execute_sql" or "submit_solution")
- "arguments": object - a JSON object containing the parameters for that tool
3. Close the tool call tag: </tool_call>
Format examples:
For testing/exploration:
<tool_call>{{"name": "execute_sql", "arguments": {{"sql": "SELECT * FROM table LIMIT 5"}}}}</tool_call>
For final submission:
<tool_call>{{"name": "submit_solution", "arguments": {{"sql_list": ["SELECT col FROM table WHERE condition"]}}}}</tool_call>

Critical Format Requirements
Your trajectory MUST follow this exact structure for EVERY turn:
1. **<think> tags**: Concise, high-level reasoning about what you're doing and why. Do NOT include SQL code in thoughts.
2. **Action**: After thinking, take ONE action by calling a tool:
- Use execute_sql for exploration and testing
- Use submit_solution for final submission (ends trajectory)
3. **Turn limit**: Maximum {max_turns} turns. Each turn: <think> + <tool_call>. **CRITICAL**: If you don't call submit_solution in the first {prev_turns} turns, the last turn MUST be submit_solution (forced submission).

Figure 8: The inference prompt used for SQL generation task (system prompt).

Inference prompt for SQL Generation (user prompt)

Database Schema
{schema}

Column Descriptions
{column_descriptions}

Question
{question}

Evidence
{evidence}

Your task: Generate the next single turn of the SQL generation trajectory. This means:
1. One <think> block with concise reasoning (no SQL code in thoughts)
2. Followed by ONE <tool_call>: either execute_sql to explore/test OR submit_solution for the final answer

Generate only the next turn. Maximum {max_turns} turns allowed. Review the conversation history to determine your current turn. **If this is the final turn, you MUST call submit_solution.**

Note: If conversation history is provided below, continue from where the exploration left off. Analyze previous results and determine the next logical action. If this is the first turn, start by analyzing the question and exploring the database.

Figure 9: The inference prompt used for SQL generation task (user prompt).

Baseline Prompt for resolving SQL issues with an LLM

You are a SQL assistant. Your task is to understand user issue and correct their problematic SQL given the database schema. Please wrap your corrected SQL with ```sql\n[Your Fixed SQL]\n``` tags in your response.

Database Schema:
{SCHEMA}

User issue:
{USER_ISSUE}

Problematic SQL:
{ISSUE_SQL}

Corrected SQL:

Figure 10: The baseline prompt used for SQL debugging tasks.

Prompts of Stage 1 (Reasoning Imitation) Data Construction (1/2)

Step 1. Prompt for eliciting reasoning thought from teacher

System Prompt:

You are an expert SQL debugger. Your task is to generate a realistic thinking process for debugging a problematic SQL query.

You will be given:
1. A problem description
2. A database schema
3. A problematic SQL query
4. The correct solution SQL (for reference only)

Your job is to write a thinking process AS IF you are debugging the problematic SQL from scratch, without knowing the solution beforehand. Your reasoning should naturally lead to discovering the issues and arriving at the correct solution.

Focus on the debugging journey, not on comparing two SQLs.

User Prompt:

Problem Description
{query}

Database Schema
{schema}

Problematic SQL
```sql
{issue_sql}
```

Correct Solution (for reference only)
```sql
{solution_sql}
```

Generate a detailed, high-level thinking process AS IF you are debugging the problematic SQL from scratch. Pretend you don't know the solution yet - show how you would analyze the problem and naturally arrive at the fix.

Requirements:
1. Do NOT simply compare the problematic SQL with the solution
2. Do NOT repeat or quote the SQL code in your reasoning
3. Show a realistic debugging thought process: analyze requirements -> identify issues -> reason about fixes
4. Explain the "why" behind each issue and how you would discover it
5. Your reasoning should naturally lead to the solution
6. Keep your reasoning between 300-500 words

Wrap your response in <thought> tags:

<thought>
[Your detailed debugging thought process here - 300-500 words]
</thought>

Figure 11: The prompt used in Stage 1 reasoning imitation data construction to elicit reasoning thought from teacher model.

Prompts of Stage 1 (Reasoning Imitation) Data Construction (2/2)

Step 2. Prompt for verifying the reasoning thought could regenerate the corrected SQL.

System Prompt:

You are an expert SQL debugger. Your task is to fix bugs in SQL queries.

You will be provided with:
1. A problem description
2. The database schema
3. A problematic SQL query
4. A debugging reasoning thoughtabout the problem(s) and how to fix it.
Based on the thought and your analysis, provide the correct SQL solution.

Structure your response using this format:

<solution>
[Provide the corrected SQL code here as a JSON list]
</solution>

Remember:
- Use insights from the debugging attempts to guide your solution
- Ensure the solution SQL is complete and executable
- Format the solution as a JSON list: ["SQL statement 1", "SQL statement 2", ...]
- Do NOT include markdown code fences inside the solution tag
- Provide ONLY the corrected SQL, without explanation

User Prompt:

Problem Description
{query}

Database Schema
{schema}

Problematic SQL
The following SQL code is problematic and does not satisfy the user's requirements:

```sql
{issue_sql}
```

Reasoning Thought
{thought}

Based on the above the reasoning thought, provide the correct SQL solution.

Figure 12: The prompt used in Stage 1 reasoning imitation data construction to verify whether teacher model could leverage the reasoning thought to regenerate the correct fixed SQL.

Prompt for using LLM-as-Judge to get reward (1/2)

System Prompt:

You are a STRICT SQL evaluation judge. You score SQL debugging agents on a numeric scale.
IMPORTANT: You must be HARSH and CRITICAL. Most failed solutions deserve LOW scores. High scores (8+/10 for solution, 4+/5 for trajectory) should be RARE — only for solutions that are nearly correct with minor issues.
Return only valid JSON with your scores.

User Prompt (1/2):

You are an expert SQL debugging evaluator. Score the agent's debugging process and final solution on numeric scales.

The agent debugs SQL by iteratively testing queries and analyzing results before submitting a final solution. The submitted solution was executed against automated test cases and **FAILED**. Your job is to score HOW GOOD the debugging process was and HOW CLOSE the solution is to being correct.

Problem Description:
{query}

Database Schema:
{schema}

Problematic SQL (Original):
```sql
{issue_sql}
```

Agent's Complete Debugging Trajectory:
{trajectory}

Agent's Final Submitted SQL:
```sql
{solution_sql}
```

Ground Truth Solution (Reference):
```sql
{ground_truth_sql}
```

Test Case Results (the solution FAILED these):
{test_case_results}

SCORING RUBRIC

PART A: TRAJECTORY QUALITY (0-5 integer)

Score how effectively the agent used its tools to debug, and whether the trajectory pushed toward the correct solution.

- **0**: No meaningful debugging. Agent submitted without using tools, or trajectory is empty/nonsensical.
- **1**: Minimal effort. Agent ran a query or two but showed no understanding of the bug. Random or aimless tool usage.
- **2**: Some diagnostic effort but ineffective. Agent identified symptoms but not root cause. Tool calls did not converge toward the ground truth.
- **3**: Reasonable debugging. Agent identified the general problem area and made purposeful tool calls, but missed key aspects or went partially off-track.
- **4**: Strong debugging. Agent correctly diagnosed the bug, used tools systematically to verify, and each step built on previous results toward the ground truth.
- **5**: Excellent debugging. Agent precisely identified the root cause, verified with targeted queries, and the trajectory clearly converged toward the correct solution. Only give this if the process was nearly flawless.

Figure 13: The prompt used to get LLM-as-judge reward (1/2).

Prompt for using LLM-as-Judge to get reward (2/2)

User Prompt (2/2):

```
### PART B: SOLUTION QUALITY (0-10 integer)

Score how close the submitted SQL is to the ground truth. Use the test case results to gauge the gap.

- **0**: Completely wrong. Solution is unrelated to the problem, uses wrong tables entirely, or is nonsensical SQL.
- **1-2**: Far from correct. Major structural differences from ground truth. Wrong approach entirely (e.g., missing critical JOINs, wrong aggregation strategy). Most test cases fail for fundamental reasons.
- **3-4**: Partially correct approach. Some correct elements (right tables, some correct JOINs) but significant logic errors remain. Solution addresses the problem but with major flaws.
- **5-6**: Moderate quality. Correct overall approach matching ground truth structure, but with notable errors in filtering, grouping, or conditions. Some test cases may be close to passing.
- **7-8**: Close to correct. Solution structure matches ground truth closely. Only minor differences remain (e.g., off-by-one in LIMIT, missing edge case handling, slight condition difference). Most logic is right.
- **9**: Nearly correct. Only trivial differences from ground truth (e.g., column alias, minor formatting). The solution captures the correct logic but fails on a technicality.
- **10**: Do NOT give this score. The solution failed test cases, so it cannot be perfect.

**STRICT SCORING GUIDELINES:**
- The solution FAILED automated tests. This means it produces WRONG results. Score accordingly.
- A score of 7+ should be RARE — only when the solution is structurally very similar to ground truth with minor differences.
- Compare the solution SQL against the ground truth SQL line by line. Every missing JOIN, wrong condition, or incorrect clause should reduce the score.
- Use the test case results as evidence: more failures = lower score. If all test cases failed, the solution is likely far from correct.

---

## OUTPUT FORMAT

Return ONLY valid JSON (no markdown, no code fences, no extra text):

{{
"trajectory_score": <integer 0-5>,
"trajectory_reasoning": "<1-2 sentences: what the agent did well/poorly in debugging>",
"solution_score": <integer 0-10>,
"solution_reasoning": "<1-2 sentences: key differences between submitted SQL and ground truth>"
}}

**CRITICAL:**
- trajectory_score MUST be an integer from 0 to 5
- solution_score MUST be an integer from 0 to 10
- Do NOT give solution_score of 10 (the solution failed tests)
- Be STRICT. Most solutions that fail tests deserve scores of 3-5, not 7+.

Now evaluate and return your JSON response.
```

Figure 14: The prompt used to get LLM-as-judge reward (2/2).

Prompt for bug checklist extraction for heuristic rubric criteria

System Prompt:

You are a senior SQL analyst. Your task is to compare a buggy SQL query against its corrected version and extract the specific bugs as a concise checklist.

Rules:
- Extract between 1 and 3 bug points. No more than 3.
- Each bug point must be ONE specific, concrete issue in a single sentence.
- Be precise: reference exact clauses, operators, column names, or logic errors.
- Do NOT describe symptoms (e.g., "returns wrong results"). Describe root causes.
- Do NOT suggest fixes. Only describe what is wrong.
- Return your answer inside <bug></bug> tags with NO text outside the tags.
- Inside the tags, return ONLY valid JSON — no markdown, no code fences, no explanation.

User Prompt:

Compare the buggy SQL against the correct SQL and extract the specific bugs.

Problem Description
{query}

Database Schema
{schema}

Buggy SQL (issue_sql)
```sql
{issue_sql}
```

Correct SQL (ground_truth_sql)
```sql
{ground_truth_sql}
```

Extract bugs existed in the task. Each bug is a single concise sentence identifying one specific root cause. Reference exact SQL elements (clause names, operators, column names, table names).

Wrap your answer in <bug></bug> tags. Inside the tags, return ONLY valid JSON:

<bug>
{{
"bugs": [
"<concise root cause, 1 sentence>",
"<concise root cause, 1 sentence>"
]
}}
</bug>

Examples of GOOD bug descriptions:
- "The HAVING clause uses strict > instead of >= which excludes exact matches."
- "Missing IS NOT NULL filter on the Enrollment column before aggregation."
- "The WHERE clause filters on IDENTIFIER = 'Job' but this column only contains 'Employee' values."
- "Using INNER JOIN instead of LEFT JOIN drops rows with no matching records in the right table."
- "The GROUP BY includes a derived column alias that is not valid in this SQL dialect."

Examples of BAD bug descriptions (too vague — do NOT write like this):
- "The query logic is wrong."
- "The results are incorrect."
- "The query needs to be rewritten."
- "There is an issue with the JOIN."

Now analyze the SQL pair and return your answer inside <bug></bug> tags.

Figure 15: The prompt for the heuristic rubric reward: instructions used to extract a bug checklist from the task context.

Prompt for bug verification for heuristic rubric criteria

System Prompt:

You are a precise text analyst. Your task is to determine whether an agent's reasoning demonstrates identification of specific SQL bugs.

Rules:
- Answer strictly based on what the agent wrote.
- true: the agent clearly understood and articulated this specific issue.
- Mentioning a related keyword alone is NOT enough. The agent must show understanding of the specific problem.
- false: the agent did not address this issue, or only vaguely referenced it without demonstrating understanding.
- Return your answer inside <verification></verification> tags with NO text outside the tags.
- Inside the tags, return ONLY valid JSON.

User Prompt:

Read the agent's reasoning below, then check whether the agent identified \
each of the listed bugs.

Agent's Reasoning
{think_blocks}

Bugs to Check ({num_bugs} total)
{bugs_block}

For each bug, answer true if the agent identified it, false if not.

Wrap your answer in <verification></verification> tags. Inside the tags, return ONLY valid JSON:

<verification>
{{
"answers": [true/false, ...]
}}
</verification>

The "answers" array must have exactly {num_bugs} elements, one per bug, \
in the same order. Do NOT write anything outside the <verification></verification> tags.

Figure 16: The prompt for the heuristic rubric reward: instructions used to verify whether the reasoning of agent correctly identifies the checklist bugs.

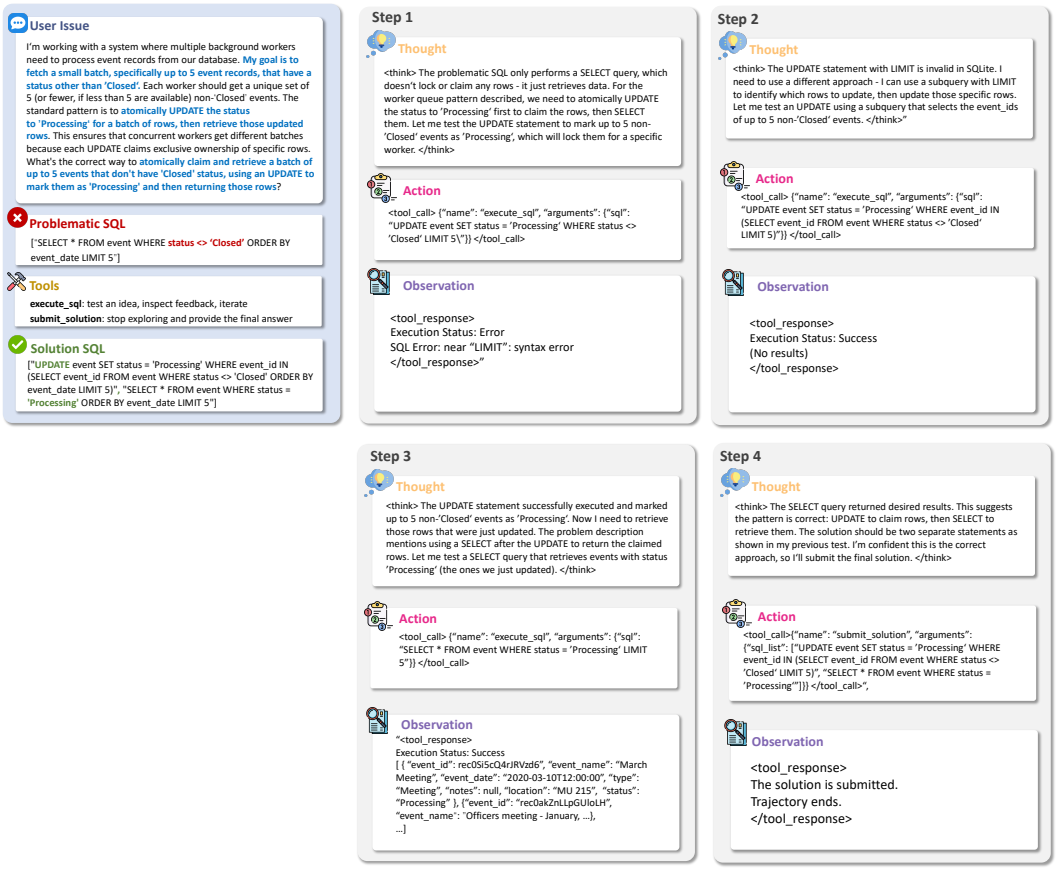


Figure 17: The example of trajectory from BIRD-TALON solving SQL debugging task.

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 abstract and introduction clearly state the paper's contributions and scope, aligning well with the methodology and experimental results.

   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: We discussed the limitations of the work in Appendix E.

   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: This paper does not include theoretical results, and therefore does not contain formal theorems or 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: In Appendix C.1 and C.2, we provide detailed descriptions of the alias of the LLMs used, the training details, infrastructure mechanism, and inference configurations. Additionally, in Appendix H, we include all the prompts used in the experiments. To support the reproducibility of the reported results, we provide access to our implementation through the code repository linked in the abstract.

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: The code and data for reproducing the main experimental results are available through the GitHub repository linked in the abstract.

   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: In Appendix C, we provide detailed information about the experimental setup, including model specifications, training details, and inference configurations.

   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: All the results shown in the paper are based on experiments run five times for each model and report the average results. This approach ensures that the results reflect consistent performance across multiple runs, accounting for any variability.

   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: [Yes]

   Justification: The specific compute resources needed for training and running our experiments are detailed in Appendix C.2.

   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 the paper adheres to the NeurIPS Code of Ethics as claimed in Appendix F. We have reviewed the guidelines and ensured that the work aligns with the ethical standards outlined, including considerations related to data privacy and model fairness.

   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: [Yes]

    Justification: We discuss the broader impacts in the Appendix G.

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: This research does not involve releasing models or data that possess a high risk of being misused.

    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: [Yes]

    Justification: The original creators of the datasets (e.g., BIRD-SQL, BIRD-CRITIC-MULTI, BIRD-CRITIC-SQLITE) and the base LLM weights (e.g., Qwen2.5-Coder) used in our experiments are properly cited and credited throughout the paper.

    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: We provide detailed documentation for our new assets.

    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: This research does not involve crowdsourcing or 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 research with human subjects, so IRB approval is not required.

    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: [N/A]

    Justification: No declaration is necessary, as the core methodology does not utilize LLMs as a significant, original, or non-standard component.

    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.