SCOPE-SQL: Structured Semantic Credit for Reinforcement Learning of Interactive SQL Repair Agent

Anonymous ACL submission

Abstract

Text-to-SQL deployment is moving from one-shot query generation to interactive repair of executable but semantically incorrect SQL. In this setting, a model must revise an erroneous query under public tool feedback while preserving correct input structures. Existing reinforcement-learning signals are ill matched: outcome rewards reveal only whether the final SQL passes, whereas unconstrained process supervision may reward steps not grounded in SQL semantics. We propose **SCOPE-SQL**, a reinforcement-learning framework that treats SQL structure as an auditable credit space for interactive repair. It maps candidate queries and public feedback into F0–F6 modules—schema/runtime, table/join, projection, predicate, aggregation/grouping, order/limit, and nesting—and assigns action-level credit for repairing target modules, preserving stable modules, gathering evidence, and submitting when ready. Persistent low-credit regions then drive semantic refresh, generating validator-filtered frontier repair tasks. On BIRD-CRITIC-SQLite, SCOPE-SQL with semantic refresh reaches 48.8% success using a Qwen2.5-Coder-7B backbone, improving over current sota RL baselines by 4.4 and 4.2 points. Ablations attribute 4.4 and 1.8 points to process credit and semantic refresh. Trajectory analysis further shows better repair behavior: Repair+Preserve paths rise from 38.8% to 77.5%, while damage-and-recover paths fall from 43.8% to 1.2%. These results suggest that SQL-structured credit is an interpretable training signal for both policy learning and data expansion in interactive SQL repair.

1 Introduction

Text-to-SQL systems are turning relational databases into natural-language interfaces, driven by large benchmarks and strong parsing models (Zhu et al., 2025b; Liu et al., 2025a; Li et al., 2024). Real deployments, however, often expose a harder debugging setting: a query may execute successfully and return plausible results, yet still deviate from user intent because of semantic drift (Gong et al., 2025; Liu et al., 2025b; Luo et al., 2025a). Such failures motivate a shift from one-shot Text-to-SQL generation toward interactive SQL repair, where an agent iteratively revises erroneous SQL by incorporating tool feedback (Li et al., 2025e; Huo et al., 2026).

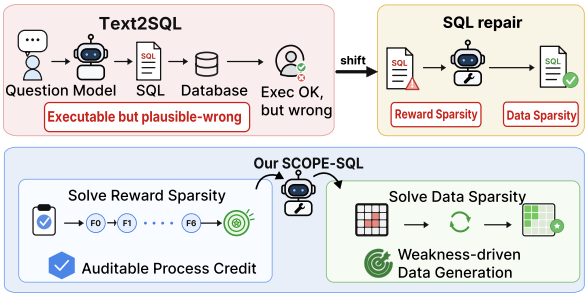

Figure 1: **From Text-to-SQL to SWE-SQL repair.** Executable SQL can still be plausibly wrong, so SWE-SQL shifts the problem from generating a query from scratch to repairing an erroneous SQL program under public evidence. SCOPE-SQL addresses the resulting reward sparsity with structured semantic credit from public tool (`check_sql`) states, and addresses data sparsity by refreshing tasks from low-credit semantic regions.

Prior work has shown that reinforcement learning is an effective approach for training multi-turn tool-interactive agents (Gong et al., 2025; Li et al., 2025e). However, when applied to SQL repair, reinforcement learning faces two key bottlenecks:

(1) Reward sparsity. An outcome reward only says whether the final SQL passes; it cannot attribute which step repaired a drifted structure, preserved a correct join, obtained useful evidence, or damaged a stable component (Zhang et al., 2025b; Yao et al., 2025; Weng et al., 2025). Existing process-reward work for Text-to-SQL provides fine-grained supervision, but when the process signal is not tied to reliable task states, it can distort

training (Zhang et al., 2025b; Weng et al., 2025). SQL repair therefore needs process feedback that is both fine-grained and grounded in reliable task states.

(2) Data sparsity. High-quality interactive repair trajectories are costly to collect and hard to scale to long-tail structural failures (Yu et al., 2018; Li et al., 2023b, 2025e). Yet the deeper bottleneck is not merely data volume: existing training pipelines lack a structure-aware way to localize where the agent persistently fails.This makes data expansion coarse and inefficient, rather than targeted toward the agent's actual weaknesses.

Our key insight is that SQL's modular structure, such as joins, predicates, and aggregations, provides a natural coordinate system for both credit assignment and data generation. We summarize F0–F6 modular—schema/runtime, table/join, projection, predicate, aggregation/grouping, order/limit, and nesting. Each repair action can be analyzed by how it improves or damages specific structural modules; in turn, persistent failures over these modules reveal where additional training data should be generated. We therefore introduce *semantic credit*: the computable contribution of an interaction action to a structured SQL semantic target under public evidence, such as schema information, execution feedback, public tool states, and interaction history. By decomposing outcome correctness into process feedback, semantic credit provides dense supervision for agent training. At the same time, persistently low-credit structural regions can be used to guide data curricula, allowing the same coordinate system to support both training signals and data generation.

Motivated by this observation, we propose SCOPE-SQL (**S**tructured **C**redit **O**ptimization with **P**ublic **E**vidence), a structured semantic-credit framework for interactive SQL repair that assigns credit using agent-accessible public evidence. As shown in Figure 1, SCOPE-SQL decomposes SQL into semantic modules and tracks each action's localized impact on "target" modules (needing repair) vs. "protected" modules (already correct), providing fine-grained process credit. It further uses persistently low-credit regions to generate targeted training data, closing the loop between credit signals and data expansion.

We evaluate SCOPE-SQL on challenging real-world SQL issue repair on BIRD-CRITIC-SQLite (Li et al., 2025e). SCOPE-SQL with a 7B backbone achieves 48.8% success rate, outperforming same-scale and larger baselines by 4+ points, while shifting repair behavior from chaotic damage-and-recover patterns to precise targeted edits. Ablations show that process credit and data expansion contribute **4.4** and **1.8** percentage-point gains, respectively. These results suggest that structured semantic credit can serve as a shared interface between reinforcement learning signals and curriculum-driven data expansion for interactive SQL repair.

Overall, our main contributions are summarized as follows:

(1) Structured Semantic Credit. We formalize interactive SQL repair as structured semantic credit assignment under public evidence.

(2) Fine-grained Credit Assignment. We propose a semantic-credit training algorithm that combines outcome correctness with module-level process feedback for action-level optimization.

(3) Data Expansion. We introduce a semantic-credit-driven data refresh strategy that turns low-credit modules, failure clusters, and structured repair labels into verifiable frontier tasks.

2 Structured Semantic Credit in SQL Repair

We first formalize the public-evidence repair task, then define semantic credit over SQL structural states.

2.1 Task Definition

Each repair instance is $x = (q, S, D, y_0)$, where q is the user question, S the schema, D the database, and y_0 the issue SQL. At step t, the Repair Agent chooses

$$\mathcal{A} = \{\texttt{execute_sql}, \texttt{check_sql}, \texttt{submit_solution}\},$$
$$a_t \in \mathcal{A}, \qquad o_t = \mathrm{Env}(a_t, h_t),$$

from public history h_t, and finally submits $\hat{y}$ with objective $\mathrm{Eval}(\hat{y}) = 1$. The agent observes only public h_t and o_t; reference SQL, hidden tests, and reward internals are unavailable.

Unlike standard Text-to-SQL generation, repair follows recent SQL repair and debugging settings (Gong et al., 2025; Li et al., 2025e): it starts from a partially correct program and must jointly preserve stable structures while modifying drifted ones. Too few edits leave errors unresolved, whereas aggressive rewrites damage correct components. Thus, intermediate progress must be observable and attributable rather than hidden in the final SQL string.

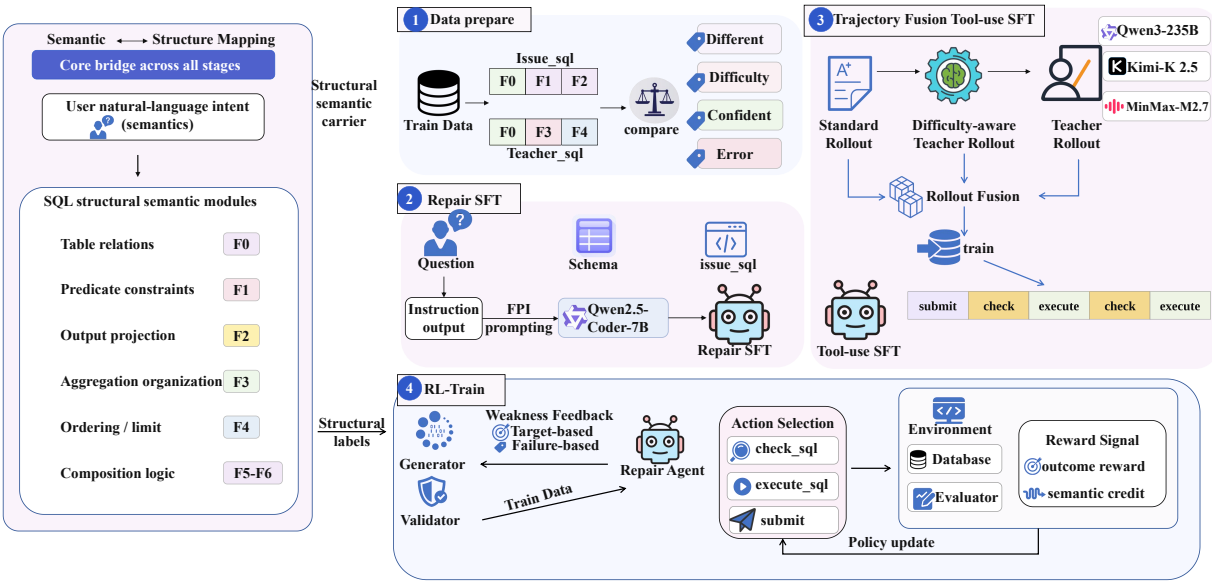

Figure 2: **Overview of SCOPE-SQL.** SCOPE-SQL tracks repair with structured semantic modules, assigns action-level credit from public evidence, and refreshes tasks from low-credit regions. The Repair Agent is the only trained and test-time interactive agent; the Generator and Validator are offline code-driven training components for weakness-driven refresh.

2.2 Structured Semantic Credit in SQL Repair

Before assigning training signals, SCOPE-SQL defines semantic progress as observable changes in public SQL-structure states, so each execution, diagnosis, or submission action receives credit according to its effect on task-bearing SQL modules.

SQL structure carries task semantics. Based on the semantic error taxonomy of NL2SQL-BUGs (Liu et al., 2025b), we use seven modules

$$\begin{aligned}\mathcal{F} = \{&F_0\!:\!\text{schema/runtime},\ F_1\!:\!\text{table/join},\\ &F_2\!:\!\text{projection},\ F_3\!:\!\text{predicate},\\ &F_4\!:\!\text{aggregation/group},\\ &F_5\!:\!\text{order/limit},\ F_6\!:\!\text{nested}\}.\end{aligned}$$

These modules turn "Is the SQL better?" into judgments over repaired, preserved, and regressed structures.

SCOPE-SQL further distinguishes a query's *semantic profile* from its *repair target*. Let $\phi_f(y)$ denote SQL y's instantiation of module f. With reference SQL $y^\star$ used only offline for labels, tags, and evaluation targets, define

$$\begin{aligned}\mathcal{T} &= \{f \in \mathcal{F} : \phi_f(y_0) \neq \phi_f(y^\star)\},\\ \mathcal{P} &= \{f \in \mathcal{F} : \phi_f(y_0) = \phi_f(y^\star)\}.\end{aligned}$$

$\mathcal{T}$ contains target modules to change and $\mathcal{P}$ protected modules to preserve; $y^\star$ is never exposed to the Repair Agent or tool observations. Credit is constructive for improvements on $\mathcal{T}$ and negative for regressions on $\mathcal{P}$; mere structural presence does not imply a repair target.

To observe semantic progress during interaction, SCOPE-SQL uses an agent-visible public `check_sql` diagnostic tool. It maps the candidate SQL, schema evidence, public execution feedback, structural changes, and interaction history into public module states. Semantic credit comes from structural state changes between two consecutive steps. It evaluates each action along four dimensions: constructive improvement, stability preservation, evidence gain, and submission readiness. Final task success remains the objective. Semantic credit provides interpretable and calibratable process feedback for intermediate decisions; the public state schema and checker implementation are summarized in Appendix D, and reward calibration is given in Appendix E.

3 Methodology

3.1 Multi-stage Repair Agent Training

SCOPE-SQL first learns a direct SQL repair prior from canonical repair pairs, then converts this prior into an interactive tool-use policy. During supervised warm start, the model maps erroneous SQL to verified teacher repairs; during interaction, each instance provides a question, schema, issue SQL,

SQLite database, and public tool feedback.

Tool and information boundaries. Only public tool outputs enter the Repair Agent context: `execute_sql`, public `check_sql`, and `submit_solution`. Reward-side checker/manager outputs, Generator/Validator metadata, oracle labels, hidden tests, and final-evaluator diagnostics are trainer-, data-construction-, or evaluation-only signals and never appear in prompts, public tool observations, or test-time histories; the full access matrix is given in Appendix A.

Two-stage supervised warm start. SCOPE-SQL uses two supervised stages. The first trains a direct semantic repair prior from raw SQL pairs: given the question, schema, and issue SQL, the model outputs only the verified teacher SQL. This stage excludes tool calls, JSON actions, markdown explanations, and natural-language rationales, so the backbone first learns the mapping from erroneous SQL to corrected SQL.

The second stage converts the model into an interactive Repair Agent. We supervise verified trajectories whose observations are produced by the actual `execute_sql`, `check_sql`, and `submit_solution` tools. We retain only trajectories whose candidate SQL passes the evaluator, whose intermediate observations are real, and whose final submission is supported by observed evidence. This stage teaches the action protocol before RL begins.

We keep the stages separate because they learn different prerequisites for RL: the first builds a semantic repair prior over SQL edits, while the second binds that prior to valid `execute_sql`/`check_sql`/`submit_solution` trajectories.

RL stage. Starting from the supervised Repair Agent, SCOPE-SQL performs interactive RL in the same public tool environment. The policy samples multiple repair trajectories for each instance and learns from both final submission correctness and step-level semantic credit. The RL objective is described in Section 3.2; Appendix H gives the repair-agent and reward-side semantic-checker templates.

3.2 Semantic-Aware Policy Optimization

Given the public F0–F6 state defined in Section 2, SCOPE-SQL converts each transition $s_{i,j-1} \rightarrow s_{i,j}$ into process credit and mixes it with final outcome credit for policy optimization. Reward computation is not an observation channel: reward values, module weights, target/protected labels, and reward-side checker outputs are used only by the trainer.

For each sampled repair instance, the policy produces a set of interactive trajectories. We denote the i-th trajectory as

$$\tau_i = \{(o_{i,1}, a_{i,1}), \ldots, (o_{i,j}, a_{i,j})\}, \qquad (1)$$

where $o_{i,j}$ is the public tool observation and $a_{i,j}$ is the action at step j. After each action is executed, the current public state is updated; we denote the public state after step j in trajectory i as $s_{i,j}$, with $s_{i,0}$ denoting the initial state. Semantic process credit compares $s_{i,j-1}$ and $s_{i,j}$.

Reward design. Rewards consist of final outcome reward r_i^o and per-action process reward $r_{i,j}^p$.

$$r_{i,j}^p = \lambda_{\text{sem}} R_{i,j}^{\text{sem}} + \lambda_{\text{act}} R_{i,j}^{\text{act}}. \qquad (2)$$

$R_{i,j}^{\text{sem}}$ is a weighted signed F0–F6 transition score, positive for target-module improvement and negative for protected-module harm. The action term

$$R_{i,j}^{\text{act}} = \lambda_{\text{evi}} E_{i,j} + \lambda_{\text{sub}} U_{i,j} - \lambda_{\text{cost}} C_{i,j}. \qquad (3)$$

rewards evidence gain for `execute_sql`/`check_sql`, supported `submit_solution`, and penalizes unnecessary tool use; Appendix E gives thresholds, clipping, target/protected weights, and protocol penalties.

Outcome reward. The outcome reward r_i^o is obtained from the final submitted SQL of trajectory τ_i:

$$r_i^o = \text{Eval}(\text{sql}_{i,\text{sub}}). \qquad (4)$$

The final evaluator first extracts the SQL from `submit_solution`. If there is no valid submission, the SQL cannot be parsed, execution fails, or execution times out, the score is 0. Otherwise, the submitted SQL is executed on the target database and checked by the task test cases; the scalar score is the fraction of passed tests. In experiments, SR counts an instance as successful only when the final evaluator marks the submitted SQL as passing. The final evaluator is not an interactive tool and does not return diagnostic feedback to the policy.

Advantage estimation. Following group-relative policy optimization (Shao et al., 2024), SCOPE-SQL normalizes rewards at their natural granularity: outcome rewards over all sampled trajectories

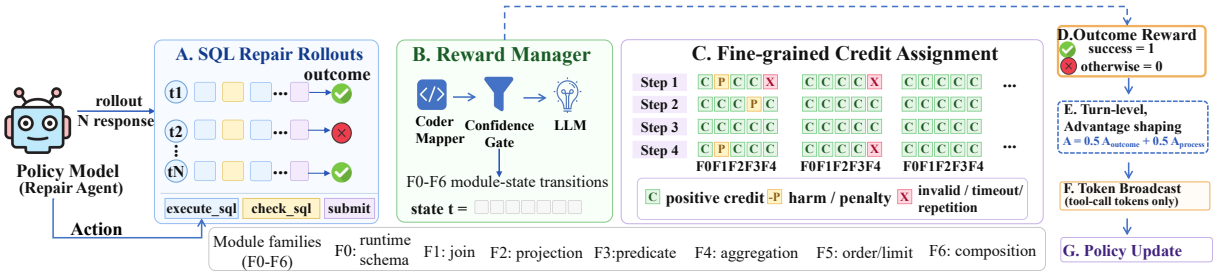

Figure 3: **Semantic-credit RL in SCOPE-SQL.** Grouped repair rollouts receive outcome credit and module-level process credit; the mixed advantage is applied to the tool-call tokens that produced each action. The reward-side semantic checker and reward manager are training-only components and are not interactive agents or test-time tools.

$\mathcal{G}$, and process rewards over the same-step valid subset $\mathcal{G}_j$:

$$A_i^{\text{out}} = \frac{r_i^o - \mu_{\mathcal{G}}(r^o)}{\sigma_{\mathcal{G}}(r^o) + \epsilon}, \qquad A_{i,j}^{\text{proc}} = \frac{r_{i,j}^p - \mu_{\mathcal{G}_j}(r_{\cdot,j}^p)}{\sigma_{\mathcal{G}_j}(r_{\cdot,j}^p) + \epsilon}. \tag{5}$$

Thus, outcome credit compares complete trajectories, whereas process credit compares same-step actions.

For the token span that realizes action $a_{i,j}$, the final optimization advantage is a weighted mixture:

$$A_{i,j} = \lambda A_i^{\text{out}} + (1-\lambda) A_{i,j}^{\text{proc}}. \tag{6}$$

Our main run uses $\lambda = 0.5$, giving equal weight to outcome correctness and semantic process credit. We keep outcome-only and conservative process-credit variants as ablations.

To make the process signal actionable for language-model policy optimization, we assign $A_{i,j}$ to the action tokens that produced tool call $a_{i,j}$. This maps semantic improvement and evidence gain back to the exact SQL-bearing action tokens, allowing the policy update to prefer edits that repair specific semantic modules rather than merely memorizing successful final strings.

3.3 Semantic-Aware Data Expansion

SCOPE-SQL expands the training pool through three semantic-aware steps: weakness diagnosis, issue SQL generation, and quality filtering. This component supplies validator-filtered generated repair tasks for RL and later refresh, but it is an offline code-driven data-construction component rather than multi-agent debate, adversarial generation, or a separate interactive policy stage.

Weakness diagnosis. SCOPE-SQL first diagnoses where additional repair tasks are useful. A weakness is a generatable and verifiable target cell where the current policy exposes a stable repair gap, described by its repair family, semantic group, mutation operator, difficulty band, error mode, and behavior failure mode. For pre-RL expansion, SCOPE-SQL locates such cells with Monte Carlo repair behavior from the current checkpoint. We estimate the policy's success probability on each eligible seed by sampling multiple repair trajectories:

$$p_{\text{mc}}(x) = \frac{1}{n} \sum_{k=1}^{n} \mathbf{1}[\text{Eval}(\text{sql}_k) = \text{pass}]. \tag{7}$$

A seed is eligible only when its teacher SQL is executable under execution guards and its issue SQL either executes to a wrong result or exposes a repairable parse, schema, or runtime failure. Teacher or gold SQL is used only for offline pool construction and validation, and is not visible to the Repair Agent during interaction. The Monte Carlo estimate identifies two useful buckets: *targeted-frontier* examples with intermediate success, $0.25 \le p_{\text{mc}}(x) \le 0.75$, and *failure-refinement* examples with $0.125 \le p_{\text{mc}}(x) \le 0.50$ or $p_{\text{mc}}(x) = 0$. In post-RL refresh, the same cell abstraction is populated from a weakness digest built from rollouts of the frozen selected checkpoint, including outcome failure, process reward, F0–F6 module credit, tool-use behavior, and Validator feedback. The rule controller converts these weaknesses into target semantic groups, difficulty bands, operator preferences, and generation quotas.

Issue SQL generation. Because SQL encodes question semantics, SCOPE-SQL perturbs verified gold SQL rather than asking an LLM to invent new tasks:

$$y_{\text{issue}} = M_{f,m}(y^\star), \qquad f \in \mathcal{F},\ m \in \mathcal{M}_f. \tag{8}$$

The generator selects module f from repair-diff target cells and applies a deterministic mutation operator m, localizing the injected error and yielding a known repair direction from y_{issue} back to the verified target $y^\star$. Appendix G lists the mutation operators.

Quality filtering. Generated candidates enter the training pool only after strict filtering. The teacher SQL must remain a valid solution, while the generated issue SQL must expose a real repair target: it either executes to a behaviorally distinguishable wrong result or triggers a repairable parse, schema, or runtime failure. The Validator is an offline data-construction filter rather than an interactive tool. It rejects near-duplicates, incompatible mutation profiles, invalid SQL, unsafe statements, and cases without a real repair objective. Its accept/reject decisions and oracle-side validation metadata are not included in Repair Agent observations. Accepted candidates are then re-evaluated with generated-candidate Monte Carlo sampling and kept only if they remain within the targeted-frontier or failure-refinement difficulty region, pass execution-based validation, and satisfy bucket quotas. Thus, generated data consists of verified SQL-repair tasks whose difficulty is measured on the generated issue SQL itself; additional construction and filtering details are in Appendix F.

4 Experiments

4.1 Experimental Setup

Training details. We use Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) as the base model and train it with two full-parameter SFT warm-start stages: Stage 1 learns direct SQL repair from question/schema/issue-SQL pairs, and Stage 2 learns verified tool-use trajectories with real execute/check/submit observations. RL starts from the Stage 2 checkpoint and trains on validator-filtered generated repair tasks derived from the SIX-GYM-SQLite (Li et al., 2025e) training split; the main run uses rollout group size $N = 4$, outcome/process advantage mixing of $0.5/0.5$, and a fixed per-trajectory tool-call budget of at most three `execute_sql`, two `check_sql`, and one `submit_solution` calls, i.e., at most six valid tool-call turns. Appendix C summarizes the stage-wise configuration.

Evaluation and baselines. We evaluate on BIRD-CRITIC-SQLite (Li et al., 2025e), a SQLite version provided by the official BIRD-CRITIC release. We use greedy decoding and report success rate (SR) as the evaluation metric. We compare SCOPE-SQL with the following three types of baselines: (1) prompt engineering evaluates frozen API models, including GPT-5.4-PRO (OpenAI, 2026), Claude Opus 4.6, Kimi-K2.5, GLM-4.7, and o3-mini; (2) the supervised fine-tuning baseline is BIRD-Fixer from the SWE-SQL/BIRD-CRITIC release (Li et al., 2025e); and (3) reinforcement-learning-based fine-tuning is our primary comparison group: XiYan-SQL-CRITIC (Gao et al., 2025) performs single-turn repair with outcome-reward RL; BIRD-Talon and BIRD-Zeno are RL-trained models released by BIRD-CRITIC (Li et al., 2025e). The SR protocol and failure-credit rules are specified in Appendix A.

Method	Type	BIRD-CRITIC
Closed-source API baselines		
Claude Opus 4.6	Prompt Engineering	46.2
GPT-5.4-PRO		42.0
Kimi-K2.5		33.2
GLM-4.7		35.2
o3-mini		39.2
Qwen2.5-Coder-14B-Instruct		
BIRD-Fixer	SFT Training	40.0
BIRD-Talon	RL Training	48.0
Other open-source baselines		
XiYanSQL-QwenCoder-7B	RL Training	25.4
XiYanSQL-QwenCoder-14B		21.8
XiYanSQL-QwenCoder-32B		32.4
Qwen2.5-Coder-7B-Instruct		
Backbone	Prompt Engineering	25.8
BIRD-Fixer	SFT Training	33.6
BIRD-Talon	RL Training	44.4
BIRD-Zeno	RL Training	44.6
SCOPE-SQL (Ours)	RL Training	$47.0^{+2.4}$
SCOPE-SQL + Refresh(Ours)	RL Training	$\mathbf{48.8}^{+4.2}$

Table 1: Main results on BIRD-CRITIC-SQLite. We report success rate (SR, %) under greedy decoding as the primary metric. We use + for absolute accuracy increase from the previous SOTA method (BIRD-Zeno) on the same backbone. SCOPE-SQL denotes the algorithm using process reward, and +Refresh denotes the data expansion method.

4.2 Main Results

SCOPE-SQL Consistently Outperforms Baselines. As shown in Table 1, our method achieves state-of-the-art performance across different baselines. On the same Qwen2.5-Coder-7B-Instruct backbone, SCOPE-SQL + Refresh improves over the previous SOTA algorithm, BIRD-Zeno, by 4.2 accuracy points. It also outperforms the larger BIRD-Talon-14B model (+0.8 pp) and even the frontier commercial model Claude Opus 4.6 (+2.6

Method	SR	Δ Acc. Drop	
		pp	Relative (%)
SFT stage ablation			
Full	39.1	–	–
w/o stage1	28.9	-10.2	↓26.0%
w/o stage2	36.2	-2.9	↓7.4%
RL stage ablation			
Full	**48.8**	–	–
w/o process reward	44.4	-4.4	↓9.0%
w/o refresh	47.0	-1.8	↓3.7%

Table 2: Ablation study results on the BIRD-CRITIC-SQLite query subset. We report success rate (SR, %) and accuracy drops relative to the corresponding full setting.

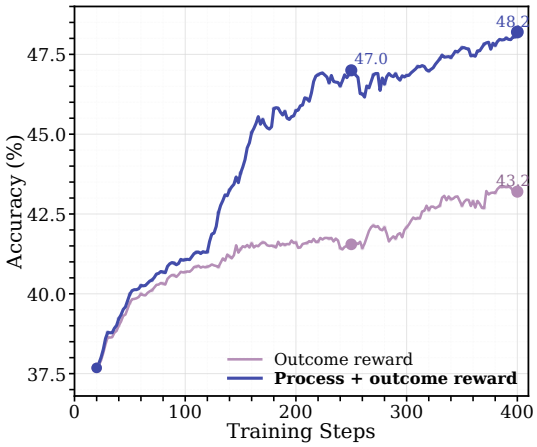

Figure 4: **Process reward learning curve.** On BIRD-CRITIC-SQLite, both curves use the same Stage 2 initialization and report SR (%) across RL checkpoints, comparing outcome-only reward with mixed process+outcome reward.

pp). Overall, our method obtains an 89.1% relative improvement over the backbone model.

4.3 Mechanism Ablations

We perform controlled ablations over the training pipeline. The study isolates two-stage SFT, process-reward mixing, and alternating refresh training while keeping the same backbone, data split, tool budget, and evaluation protocol.

Table 2 shows that both supervised warm start and semantic-credit RL contribute to the final performance. In the SFT stage, removing the direct repair prior causes the largest degradation, reducing SR from 39.1 to 28.9, while removing the tool-use SFT stage drops SR to 36.2. This indicates that Stage 1 provides the basic SQL repair capability and Stage 2 converts it into a reliable interactive policy. In the RL stage, removing process reward reduces SR from 48.8 to 44.4, a 4.4-point drop, showing that action-level semantic credit is critical beyond terminal outcome feedback. Removing refresh keeps the process-reward learner but drops SR to 47.0, indicating that semantic weakness-driven data expansion provides a complementary 1.8-point gain.

Process Reward Analysis. The ablation results already show that removing process reward causes a large performance drop. We further study how this signal affects the RL training process in Figure 4. Starting from the same Stage 2 initialization, the mixed process+outcome reward curve stays consistently above the outcome-only curve across checkpoints and reaches a higher final SR. This indicates that semantic process credit does not merely improve the final selected checkpoint; it also provides a more informative optimization signal throughout training. The result highlights two points: (1) module-level semantic progress is well aligned with final repair success, and (2) assigning credit to useful intermediate actions reduces the sparsity of terminal rewards, allowing the repair policy to learn more stable tool-use behavior.

Generator Evolution Analysis. Figure 5 analyzes how semantic refresh changes the generator over multiple training rounds. Both settings start from the same Stage 2 SFT checkpoint. Without refresh, continuing RL on the fixed data pool improves gradually to 47.0 SR at 250 steps. With semantic refresh, SCOPE-SQL aggregates low-credit modules from rollouts, generates validator-filtered repair tasks for those weakness cells, and uses the refreshed data in the next round. This produces consistently higher SR at every later checkpoint: 41.6, 44.4, 46.4, 47.8, and finally 48.8. These results indicate that semantic modules provide actionable signals for generator evolution.

Repair Path Analysis. We further analyze whether SCOPE-SQL changes how the model repairs SQL, rather than only whether the final SQL passes. Final evaluation is path-blind: two agents may both submit correct SQL but follow very different repair processes. On 80 matched BIRD-CRITIC instances where both SCOPE-SQL and BIRD-Talon-7B eventually pass, SCOPE-SQL reaches the correct SQL through the Repair+Preserve path much more often (77.5% vs. 38.8%). It also reduces Repair+Damage trajectories (15.0% vs. 50.0%) and damage-and-recover behavior (1.2% vs. 43.8%). These results show that semantic credit changes the dynamics of suc-

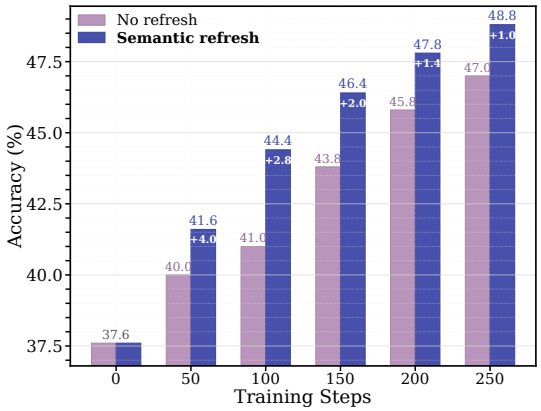

Figure 5: **Semantic refresh across rounds.** On BIRD-CRITIC-SQLite, bars report SR (%) after each training round, comparing no-refresh training against semantic refresh. Blue labels indicate the incremental gain over the previous semantic-refresh round.

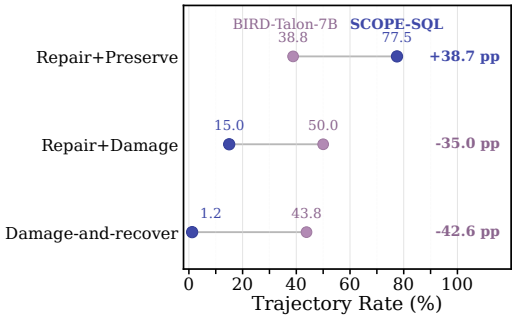

Figure 6: **Repair paths among successful matched cases.** On 80 matched BIRD-CRITIC cases, SCOPE-SQL has higher Repair+Preserve and lower Repair+Damage/damage-and-recover rates than BIRD-Talon-7B.

cessful repair: SCOPE-SQL more often fixes target structures while perturbing fewer already-correct structures in the issue SQL. Appendix I further analyzes whether outcome-failed rollouts still contain useful partial repair.

5 Related Work

Text-to-SQL, SQL repair, and agents. Text-to-SQL has advanced through benchmarks, schema-aware parsing, constrained decoding, skeleton parsing, and LLM-based decomposition or agents (Li et al., 2025b,a; Scholak et al., 2021; Li et al., 2023a,b; Pourreza and Rafiei, 2023; Wang et al., 2025; Gao et al., 2025; Pourreza et al., 2024). SQL repair work studies executable-but-wrong SQL, clause faults, test-suite equivalence, and user issues in executable environments (Zhong et al., 2020; Liu et al., 2025b; Gong et al., 2025; Li et al., 2025e; Huo et al., 2026). Tool-use work models action-observation loops and tool decisions (Yao et al., 2023; Schick et al., 2023; Qin et al., 2023), and SQL agents use tools for decomposition, schema selection, execution feedback, refinement, and search (Pourreza and Rafiei, 2023; Wang et al., 2025; Pourreza et al., 2024; Yang et al., 2026; Gao et al., 2025). SCOPE-SQL keeps the repair setting but supervises public SQL state transitions rather than only final answers.

SQL reinforcement learning and data refresh. Text-to-SQL RL uses execution rewards, reward functions, process reward models, graph matching, and stepwise CTE rewards (Ma et al., 2025; Zhang et al., 2025b; Weng et al., 2025; Yao et al., 2025). Prior work shows that outcome rewards are stable but sparse, while intermediate rewards are useful but brittle (Zhang et al., 2025b; Yao et al., 2025; Weng et al., 2025). Data-centric training uses generation, filtering, tagging, and adaptive task construction (Wang et al., 2023; Lu et al., 2024; Guo et al., 2025), and SQL debugging benefits from verified and plan-guided trajectories (Li et al., 2025e). SCOPE-SQL grounds process credit in F0–F6 structural transitions and uses low-credit modules and failure clusters for validator-filtered refresh.

6 Conclusion

We presented SCOPE-SQL, a structured semantic credit framework for interactive SQL repair under public evidence. By mapping candidate SQLs and tool feedback into F0–F6 structural-semantic states, SCOPE-SQL turns sparse outcome feedback into action-level credit for repairing target modules, preserving stable structures, and acquiring useful evidence. The same credit signal also drives validator-filtered data refresh from persistent low-credit regions. Experiments on BIRD-CRITIC-SQLite show that SCOPE-SQL improves over strong same-backbone RL baselines, and trajectory-level analysis indicates that it changes not only whether a repair succeeds, but also how the repair process reaches the final SQL. These results suggest that structured SQL semantics provide a practical coordinate system for interpretable policy learning and data expansion in interactive SQL repair.

7 Limitations

SCOPE-SQL has two main limitations.

Distribution gap in semantic refresh. Semantic Refresh creates repair tasks by deterministically mutating verified gold SQL and filtering them with a Validator. This yields controllable and verifiable issues, but may not fully capture errors from ambiguous user intent, evolving business logic, hidden data assumptions, or real debugging habits. Future work can incorporate anonymized query logs and human-reviewed issues to better match real-world repair distributions.

Evaluation scope and interaction horizon. We evaluate SCOPE-SQL mainly on SQLite-based SQL repair benchmarks derived from real-world SQL issues, with a fixed tool-call budget, rather than on repository-level software repair or production database maintenance. Production databases, such as PostgreSQL or Oracle, involve dialect differences, optimizer behavior, access control, transactions, and state-changing operations. Some repairs may also require longer interactions or user clarification. Extending SCOPE-SQL to these settings will require adapting the checker, Validator, mutation operators, tool protocol, and evaluation criteria.

References

Chengliang Chai, Nan Tang, Ju Fan, and Yuyu Luo. 2023. Demystifying artificial intelligence for data preparation. In *SIGMOD Conference Companion*, pages 13–20. ACM.

Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, Jinyang Gao, Liyu Mou, and Yu Li. 2025. A preview of XiYan-SQL: A multi-generator ensemble framework for text-to-SQL. *arXiv preprint arXiv:2411.08599*.

Yue Gong, Chuan Lei, Xiao Qin, Kapil Vaidya, Balakrishnan Narayanaswamy, and Tim Kraska. 2025. SQLens: An end-to-end framework for error detection and correction in text-to-sql. *arXiv preprint arXiv:2506.04494*.

Jiacheng Guo, Ling Yang, Peter Chen, Qixin Xiao, Yinjie Wang, Xinzhe Juan, Jiahao Qiu, Ke Shen, and Mengdi Wang. 2025. GenEnv: Difficulty-aligned co-evolution between LLM agents and environment simulators. *arXiv preprint arXiv:2512.19682*.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Li Zhang, Lingyao Zhang, Min Yang, Mingchen Zhuge, Taicheng Guo, Tuo Zhou, Wei Tao, Robert Tang, Xiangtao Lu, and 9 others. 2025. Data interpreter: An LLM agent for data science. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, volume ACL 2025 of *Findings of ACL*, pages 19796–19821. Association for Computational Linguistics.

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

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, and 5 others. 2026. BIRD-INTERACT: Re-imagining text-to-SQL evaluation for large language models via lens of dynamic interactions. *arXiv preprint arXiv:2510.05318*.

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

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The dawn of natural language to SQL: are we fully ready? [experiment, analysis & benchmark]. *Proc. VLDB Endow.*, 17(11):3318–3331.

Boyan Li, Yiran Peng, Yupeng Xie, Sirong Lu, Yizhang Zhu, Xing Mu, Xinyu Liu, and Yuyu Luo. 2026. Deepeye: A steerable self-driving data agent system. In *Companion of the 2026 International Conference on Management of Data*, SIGMOD Companion '26, Bengaluru, India. ACM.

Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025b. Alpha-sql: Zero-shot text-to-sql using monte carlo tree search. In *ICML*. OpenReview.net.

Changlun Li, Yao SHI, Chen Wang, Qiqi Duan, Runke RUAN, Weijie Huang, Haonan Long, Lijun Huang, Nan Tang, and Yuyu Luo. 2025c. Time travel is cheating: Going live with deepfund for real-time fund investment benchmarking. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Changlun Li, Chenyu Yang, Yuyu Luo, Ju Fan, and Nan Tang. 2025d. Weak-to-strong prompts with lightweight-to-powerful llms for high-accuracy, low-cost, and explainable data transformation. *Proc. VLDB Endow.*, 18(8):2371–2384.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023a. RESDSQL: Decoupling schema linking and skeleton parsing for text-to-SQL. *arXiv preprint arXiv:2302.05965*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023b. Can LLM already serve as a database interface? a big bench for large-scale database grounded text-to-SQLs. *arXiv preprint arXiv:2305.03111*.

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. 2025e. SWE-SQL: Illuminating LLM pathways to solve user SQL issues in real-world applications. *arXiv preprint arXiv:2506.18951*.

Xiaotian Lin, Yanlin Qi, Yizhang Zhu, Themis Palpanas, Chengliang Chai, Nan Tang, and Yuyu Luo. 2026. Lead: Iterative data selection for efficient llm instruction tuning. *Proc. VLDB Endow.*, 19(3):426–439.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025a. A survey of text-to-sql in the era of llms: Where are we, and where are we going? *IEEE Trans. Knowl. Data Eng.*, 37(10):5735–5754.

Xinyu Liu, Shuyu Shen, Boyan Li, Nan Tang, and Yuyu Luo. 2025b. NL2SQL-BUGs: A benchmark for detecting semantic errors in NL2SQL translation. *arXiv preprint arXiv:2503.11984*.

Keming Lu, Hongyi Yuan, Zheng Yuan, Runji Lin, Junyang Lin, Chuanqi Tan, Chang Zhou, and Jingren Zhou. 2024. #instag: Instruction tagging for analyzing supervised fine-tuning of large language models. In *International Conference on Learning Representations*.

Tianqi Luo, Chuhan Huang, Leixian Shen, Boyan Li, Shuyu Shen, Wei Zeng, Nan Tang, and Yuyu Luo. 2025a. nvbench 2.0: Resolving ambiguity in text-to-visualization through stepwise reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Yuyu Luo, Guoliang Li, Ju Fan, Chengliang Chai, and Nan Tang. 2025b. Natural language to SQL: state of the art and open problems. *Proc. VLDB Endow.*, 18(12):5466–5471.

Yuyu Luo, Xuedi Qin, Chengliang Chai, Nan Tang, Guoliang Li, and Wenbo Li. 2022a. Steerable self-driving data visualization. *IEEE Trans. Knowl. Data Eng.*, 34(1):475–490.

Yuyu Luo, Xuedi Qin, Nan Tang, and Guoliang Li. 2018. Deepeye: Towards automatic data visualization. In *ICDE*, pages 101–112. IEEE Computer Society.

Yuyu Luo, Nan Tang, Guoliang Li, Chengliang Chai, Wenbo Li, and Xuedi Qin. 2021. Synthesizing natural language to visualization (NL2VIS) benchmarks from NL2SQL benchmarks. In *SIGMOD Conference*, pages 1235–1247. ACM.

Yuyu Luo, Nan Tang, Guoliang Li, Jiawei Tang, Chengliang Chai, and Xuedi Qin. 2022b. Natural language to visualization by neural machine translation. *IEEE Trans. Vis. Comput. Graph.*, 28(1):217–226.

Yuyu Luo, Yihui Zhou, Nan Tang, Guoliang Li, Chengliang Chai, and Leixian Shen. 2023. Learned data-aware image representations of line charts for similarity search. *Proc. ACM Manag. Data*, 1(1):88:1–88:29.

Songlin Lyu, Lujie Ban, Zihang Wu, Tianqi Luo, Jirong Liu, Chenhao Ma, Yuyu Luo, Nan Tang, Shipeng Qi, Heng Lin, Yongchao Liu, and Chuntao Hong. 2026. Text2gql-bench: A text to graph query language benchmark [experiment, analysis benchmark]. *Preprint*, arXiv:2602.11745.

Peixian Ma, Xialie Zhuang, Chengjin Xu, Xuhui Jiang, Ran Chen, and Jian Guo. 2025. SQL-R1: Training natural language to SQL reasoning model by reinforcement learning. *arXiv preprint arXiv:2504.08600*.

OpenAI. 2026. GPT-5.4 Pro. `https://platform.openai.com/docs/models/gpt-5.4-pro`. OpenAI API model documentation. Accessed: 2026-05-25.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. 2024. CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. *arXiv preprint arXiv:2410.01943*.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. *arXiv preprint arXiv:2304.11015*.

Xuedi Qin, Yuyu Luo, Nan Tang, and Guoliang Li. 2020. Making data visualization more efficient and effective: a survey. *VLDB J.*, 29(1):93–117.

Yujia Qin, Shengding Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *arXiv preprint arXiv:2307.16789*.

Jianhao Ruan, Zhihao Xu, Yiran Peng, Fashen Ren, Zhaoyang Yu, Xinbing Liang, Jinyu Xiang, Yongru Chen, Bang Liu, Chenglin Wu, Yuyu Luo, and Jiayi Zhang. 2026. Aorchestra: Automating sub-agent creation for agentic orchestration. *Preprint*, arXiv:2602.03786.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Jingze Shi, Zhangyang Peng, Yizhang Zhu, Yifan Wu, Guang Liu, and Yuyu Luo. 2026. Omnimoe: An efficient moe by orchestrating atomic experts at scale. *Preprint*, arXiv:2602.05711.

Yao Shi, Boyan Li, Yuyu Luo, Lei Chen, and Nan Tang. 2025. Augmenting realistic charts with virtual overlays. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems, CHI 2025, YokohamaJapan, 26 April 2025- 1 May 2025*, pages 619:1–619:23. ACM.

Nan Tang, Chenyu Yang, Ju Fan, Lei Cao, Yuyu Luo, and Alon Y. Halevy. 2024. Verifai: Verified generative AI. In *CIDR*. www.cidrdb.org.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. MAC-SQL: A multi-agent collaborative framework for text-to-SQL. *arXiv preprint arXiv:2312.11242*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*.

Han Weng, Puzhen Wu, Longjie Cui, Yi Zhan, Boyi Liu, Yuanfeng Song, Dun Zeng, Yingxiang Yang, Qianru Zhang, Dong Huang, Xiaoming Yin, Yang Sun, and Xing Chen. 2025. Graph-Reward-SQL: Execution-free reinforcement learning for text-to-SQL via graph matching and stepwise reward. *arXiv preprint arXiv:2505.12380*.

Yifan Wu, Yiran Peng, Yiyu Chen, Jianhao Ruan, Zijie Zhuang, Cheng Yang, Jiayi Zhang, Man Chen, Yenchi Tseng, Zhaoyang Yu, Liang Chen, Yuyao Zhai, Bang Liu, Chenglin Wu, and Yuyu Luo. 2026. Autowebworld: Synthesizing infinite verifiable web environments via finite state machines. *Preprint*, arXiv:2602.14296.

Yifan Wu, Lutao Yan, Leixian Shen, Yunhai Wang, Nan Tang, and Yuyu Luo. 2024. Chartinsights: Evaluating multimodal large language models for low-level chart question answering. In *EMNLP (Findings)*, pages 12174–12200. Association for Computational Linguistics.

Jinyu Xiang, Jiayi Zhang, Zhaoyang Yu, Xinbing Liang, Fengwei Teng, Jinhao Tu, Fashen Ren, Xiangru Tang, Sirui Hong, Chenglin Wu, and Yuyu Luo. 2025. Self-supervised prompt optimization. In *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, pages 9017–9041. Association for Computational Linguistics.

Yupeng Xie, Zhiyang Zhang, Yifan Wu, Sirong Lu, Jiayi Zhang, Zhaoyang Yu, Jinlin Wang, Sirui Hong, Bang Liu, Chenglin Wu, and Yuyu Luo. 2025. Visjudgebench: Aesthetics and quality assessment of visualizations. *CoRR*, abs/2510.22373.

Chenyu Yang, Yuyu Luo, Chuanxuan Cui, Ju Fan, Chengliang Chai, and Nan Tang. 2025. Data imputation with limited data redundancy using data lakes. *Proc. VLDB Endow.*, 18(10):3354–3367.

Haolin Yang, Jipeng Zhang, Zhitao He, Alexander Zhou, and Yi R. Fung. 2026. MARS-SQL: A multi-agent reinforcement learning framework for text-to-SQL. *arXiv preprint arXiv:2511.01008*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*.

Zhewei Yao, Guoheng Sun, Lukasz Borchmann, Gaurav Nuti, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. 2025. Arctic-Text2SQL-R1: Simple rewards, strong reasoning in text-to-sql. *arXiv preprint arXiv:2505.20315*.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, and 5 others. 2019a. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 1962–1979.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921.

Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019b. SParC: Cross-domain semantic parsing in context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4511–4523.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025a. Aflow: Automating agentic workflow generation. In *ICLR*. OpenReview.net.

Yuxin Zhang, Meihao Fan, Ju Fan, Mingyang Yi, Yuyu Luo, Jian Tan, and Guoliang Li. 2025b. Reward-SQL: Boosting text-to-sql via stepwise reasoning and process-supervised rewards. *arXiv preprint arXiv:2505.04671*.

Yuxin Zhang, Meihao Fan, Ju Fan, Mingyang Yi, Yuyu Luo, Jian Tan, and Guoliang Li. 2025c. Reward-sql: Boosting text-to-sql via stepwise reasoning and process-supervised rewards. *Preprint*, arXiv:2505.04671.

Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic evaluation for text-to-SQL with distilled test suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pages 396–411.

Yizhang Zhu, Runzhi JIANG, Boyan Li, Nan Tang, and Yuyu Luo. 2025a. EllieSQL: Cost-efficient text-to-SQL with complexity-aware routing. In *Second Conference on Language Modeling*.

Yizhang Zhu, Liangwei Wang, Chenyu Yang, Xiaotian Lin, Boyan Li, Wei Zhou, Xinyu Liu, Zhangyang Peng, Tianqi Luo, Yu Li, Chengliang Chai, Chong Chen, Shimin Di, Ju Fan, Ji Sun, Nan Tang, Fugee Tsung, Jiannan Wang, Chenglin Wu, and 6 others. 2025b. A survey of data agents: Emerging paradigm or overstated hype? *CoRR*, abs/2510.23587.

A Evaluation Protocol and Information Access

Benchmark. We evaluate on BIRD-CRITIC-SQLite (Li et al., 2025e), the official SQLite release of BIRD-CRITIC/SWE-SQL. The release contains 500 human-verified user issues from real-world SQLite database applications, with task categories covering query, management, and personalization issues. Each public instance provides an `instance_id`, `db_id`, natural-language `query`, buggy `issue_sql`, SQLite `dialect` and version metadata, task `category`, and evaluator setup fields such as `preprocess_sql` and `clean_up_sql`. The SQLite database files are organized by `db_id`. Following the official release, gold solution SQL and test cases are not part of the public dataset. In our interaction protocol, the Repair Agent receives the question, schema/database context, issue SQL, and public tool observations; evaluator setup fields, hidden tests, gold SQL, and final-evaluator diagnostics remain outside policy observations.

Success-rate protocol. SCOPE-SQL reports success rate (SR) on BIRD-CRITIC-SQLite. A turn denotes one assistant tool-call action followed by its public tool observation. In the reported setting, each trajectory ends when the agent calls `submit_solution` or exhausts the per-action tool-call budget: at most three `execute_sql`, two `check_sql`, and one `submit_solution` call (six valid tool-call turns in total). The evaluator extracts the submitted SQL, enforces a single SQLite statement, runs the task-specific evaluator, and marks the instance successful only when the submitted SQL passes all required checks. Missing submissions, invalid tool-call JSON, non-parseable SQL, execution errors, and timeouts receive failure credit. Public tool observations guide the agent during interaction, but the final evaluator and hidden tests are not exposed to the policy.

Information access and leakage prevention. Table 1 summarizes which components can access oracle-side information. Oracle information may be used for offline data construction, reward computation, or final evaluation, but it is never serialized into model-facing prompts, public tool observations, or test-time interaction histories.

B More Related Works

B1. Text-to-SQL, query-language benchmarks, and SQL agents. Recent surveys and benchmarks summarize the LLM-era development of Text-to-SQL and its open problems, including robustness, ambiguity, and benchmark coverage (Liu et al., 2025a; Luo et al., 2025b; Li et al., 2024). New SQL systems further explore search, software-engineering-style pipelines, and cost-aware routing for query generation (Li et al., 2025b,a; Zhu et al., 2025a). Related benchmarks also broaden natural-language querying beyond standard single-turn SQL, including contextual and conversational SQL, graph query languages, and visualization-oriented query tasks (Wang et al., 2020; Yu et al., 2019b,a; Lyu et al., 2026; Luo et al., 2025a). Process-supervised SQL training is closely related to our use of intermediate credit, but SCOPE-SQL grounds this credit in public structural-state transitions during repair (Zhang et al., 2025c).

B2. Data agents, workflow optimization, and verified environments. General-purpose data agents combine planning, tool execution, and task-specific feedback for data science workflows (Hong et al., 2025; Zhu et al., 2025b). Agentic workflow and orchestration methods automate workflow construction, sub-agent creation, or web-environment generation (Zhang et al., 2025a; Ruan et al., 2026; Wu et al., 2026). Other work studies verified generative AI, live financial evaluation, prompt optimization, and efficient expert orchestration (Tang et al., 2024; Li et al., 2025c; Xiang et al., 2025; Shi et al., 2026). These works motivate reliable action-observation loops, while SCOPE-SQL focuses on SQL repair actions and public semantic credit.

B3. Data-centric training and data preparation. Data-centric methods improve LLM training by selecting useful data, using weak-to-strong prompting, and automating data transformation or preparation (Lin et al., 2026; Li et al., 2025d; Chai et al., 2023). Database work also studies data imputation with limited redundancy from data lakes (Yang et al., 2025). SCOPE-SQL shares the data-centric view, but uses low-credit SQL structural modules as a compact weakness signal for validator-filtered repair-task refresh.

B4. Natural-language interfaces for visualization and analysis. Natural-language data interfaces also appear in visualization and chart analysis. Prior work surveys data visualization systems, builds automatic or steerable visualization agents, and translates natural language into visualization specifications (Qin et al., 2020; Luo et al., 2018; Li

Component	Phase	Inputs	Outputs	Gold / teacher?	Hidden tests?	Policy visible?
Repair Agent	SFT/RL/test	Problem input and public history	Tool calls, final SQL	No	No	Yes
`execute_sql`	SFT/RL/test	Candidate SQL, public DB	Public execution result/error	No	No	Yes
Public `check_sql`	SFT/RL/test	Candidate SQL, schema, issue SQL, public history	F0–F6 states, evidence, readiness	No	No	Yes
Reward-side semantic checker / manager	RL training	Logged public states, action history, offline labels when available	Process rewards, advantages	Offline labels only	No as observation	No
Generator / Validator	Offline data construction	Gold/teacher SQL, mutation metadata, validation checks	Accepted repair tasks	Yes, offline only	Offline only	No
Final evaluator	RL outcome / benchmark	Submitted SQL, task evaluator	Outcome score / SR	Evaluator-internal	Yes	No

Table 1: **Information access matrix.** Oracle-side information can be used for offline construction, reward computation, or final scoring, but never appears in Repair Agent prompts, public tool observations, or test-time interaction histories.

et al., 2026; Luo et al., 2022a,b). Benchmark and evaluation work synthesizes NL2VIS tasks from NL2SQL, studies chart representations and overlays, and evaluates chart or visualization quality (Luo et al., 2021, 2023; Shi et al., 2025; Wu et al., 2024; Xie et al., 2025). These lines are adjacent to our setting because both rely on structured semantics, but SCOPE-SQL targets executable SQL repair rather than visualization generation.

C Training and Hyperparameter Details

Appendix Table 2 summarizes the training configuration used by SCOPE-SQL.

Model and checkpoint aliases. We use Qwen2.5-Coder-7B-Instruct as the base Repair Agent. Stage 1 denotes direct SQL-repair SFT from question/schema/issue-SQL pairs; Stage 2 denotes verified tool-use SFT initialized from the Stage 1 dev-best checkpoint; semantic-credit RL is initialized from the selected Stage 2 checkpoint (`global_step_640`). In result tables, BIRD-Fixer, BIRD-Talon, and BIRD-Zeno follow the names released with BIRD-CRITIC, while SCOPE-SQL + Refresh denotes the semantic-credit RL model after weakness-driven data refresh.

Resource reporting. We report the stable settings that define the training protocol rather than hardware-dependent bookkeeping such as wall-clock time or every optimizer bookkeeping parameter. API models are used only as frozen prompting baselines or trajectory teachers where stated, not as trainable components of SCOPE-SQL. All stages use SQL safety guards, and final checkpoint selection is performed on held-out development tasks.

D Public `check_sql` Tool and State Schema

The public `check_sql` tool is a code-based diagnostic tool in our experiments. It receives only public inputs: the question, schema, issue SQL, candidate SQL, and public interaction history. It parses the candidate SQL, applies SQLite safety guards, computes a structural diff against the issue SQL, and summarizes any matching public `execute_sql` evidence from the history. These signals are mapped into F0–F6 module states, confidence scores, evidence coverage, unresolved families, and submit-readiness fields. The public tool does not use an LLM, does not access reference SQL, teacher SQL, hidden tests, final-evaluator outputs, reward values, or target repair labels, and never emits repaired SQL or patch suggestions. Table 3 summarizes the returned state fields; Figures 4–5 show a full public observation object.

Public checker versus reward-side checker. The public `check_sql` tool is visible to the Repair Agent during training and evaluation and returns only public diagnostic states. The reward-side semantic checker/manager is training-only: it consumes logged public states and offline target/protected labels when available to compute process rewards, but its rewards, labels, and internal outputs are never serialized into prompts or public observations.

E Process Reward Implementation and Calibration

The reward manager operates outside the model-facing observation channel. It consumes logged public checker states and offline target/protected labels when available, but reward values, reward-side checker outputs, and token-level advantages

Stage	Data and initialization	Optimization	Protocol / reward settings
Stage 1 RawSQL SFT	4,500 train and 500 dev rows from the SIX-GYM-SQLite split; initialized from Qwen2.5-Coder-7B-Instruct; the dev-best checkpoint is selected from retained checkpoints.	Full-parameter FSDP SFT with AdamW; learning rate 1×10^{-5}, effective batch 16, 2 epochs, and maximum length 24k tokens.	Direct repair only: the target is the corrected SQL. Tool calls, JSON actions, markdown fences, and natural-language rationales are excluded.
Stage 2 Tool-use SFT	3,000 accepted verified trajectories: 1,200 oracle-solution replay, 1,500 strong-teacher rollouts, and 300 f-plan-boosted trajectories. Teachers are qwen3-235b-a22b-instruct-2507, Kimi-K2.5, and MiniMax-M2.7. Training starts from the Stage 1 dev-best checkpoint.	Full-parameter SFT with next-action prefix expansion; learning rate 5×10^{-6}, global batch 32–64, and maximum length 8192. Observation tokens are kept in context but masked from the loss.	The model is supervised only on assistant tool-call tokens. Accepted trajectories must contain real `execute_sql`, `check_sql`, and `submit_solution` observations and pass the final evaluator.
Semantic-credit RL	Initialized from the Stage 2 verified tool-use checkpoint (`global_step_640`); the RL pool uses 2,500 validator-filtered generated repair tasks, split 50/50 between targeted-frontier and failure-refinement cells.	Full-parameter token-level policy optimization; rollout group size $N = 4$, learning rate 1×10^{-6}, KL coefficient 0.01, sampling temperature 1.0, maximum prompt length 15k, and maximum response length 4096.	The tool-call budget is at most three `execute_sql`, two `check_sql`, and one `submit_solution` calls, so one trajectory contains at most six valid tool-call turns and may terminate earlier on submission. The policy observes the public checker only; reward-side semantic checking is used only during training. Outcome and process advantages are group-normalized separately and mixed as $0.5A^{\text{out}} + 0.5A^{\text{proc}}$.

Table 2: **Training and hyperparameter details.** Stage 1 builds a direct SQL-repair prior, Stage 2 teaches verified tool use, and semantic-credit RL optimizes from the Stage 2 checkpoint.

Field	Role	Typical values	Use in SCOPE-SQL
`module_states`	F0–F6 public semantic diagnostics	`ok`, `stable`, `suspected_bad`, `not_applicable`, with confidence and evidence	Defines semantic progress and protected-structure harm.
`result_shape`	Public execution shape	column names, column count, sampled row count, latency	Provides evidence for projection, predicate, aggregation, and ordering decisions.
`diff_summary`	Structural change summary	changed clauses, edit ratio, removed anchors	Helps estimate whether an edit is surgical or likely to damage stable structures.
`evidence_coverage`	Whether the current SQL has public support	score in $[0, 1]$	Rewards useful `execute_sql` and `check_sql` calls.
`submission_value`	Submit-readiness signal	`likely_continue`, `ready_to_submit`, score	Rewards well-supported submission and discourages premature submission.
`unresolved_families`	Remaining semantic warnings	family, confidence, public reason	Identifies target modules for later repair actions and weakness aggregation.
`guard fields`	Visibility and safety flags	safety guard, no gold SQL, no hidden tests	Maintains the public-evidence setting during interaction.

Table 3: Public `check_sql` state fields used for diagnosis and process credit.

are used only by the trainer and are never inserted into Repair Agent prompts or tool observations.

The implementation instantiates semantic reward with module-level credit from consecutive public checker states. Positive credit is assigned when a semantic module moves from unresolved or uncertain toward improved, stable, or correct; negative credit is assigned when a previously stable module is damaged. Target modules receive larger positive weight, while protected-module harm receives larger negative weight. The implementation clips both per-step process reward and total trajectory process reward.

Guarding against spurious process credit. Process credit is auxiliary to final task success: it is clipped, mixed with outcome advantage, and penalizes protocol failures, unsafe actions, premature submission, and protected-module harm. Thus, intermediate rewards shape useful action spans without replacing the final evaluator as the objective.

Algorithm D.1: Module-level semantic credit. Given the previous public state s_{t-1}, current public state s_t, action a_t, target modules $\mathcal{T}$, protected modules $\mathcal{P}$, and action history h_t, initialize positive credit $C^+ = 0$ and harm credit $C^- = 0$. For each module $f \in \{\text{F0}, \ldots, \text{F6}\}$, skip f if either state marks it as `not_applicable`. Otherwise, read its previous score, current score, current confidence, and status transition. If f moves from `broken`, `suspected_bad`, or `unknown` toward `improved`, `stable`, or `ok`, or if its score improves by at least the improvement threshold with sufficient confidence, add positive credit weighted by whether $f \in \mathcal{T}$. If f moves from `ok`, `stable`, or

improved to suspected_bad or broken, or if its score drops by at least the harm threshold, add negative credit weighted by whether $f \in \mathcal{P}$. Cap positive and negative module credit per family and per step. Then add evidence credit for useful execute_sql/check_sql, submit-readiness credit for submit_solution, hard penalties for unsafe or timeout states, and the fixed action cost. The final local process reward is the clipped sum of these terms; trajectory-level process reward is the clipped sum over local rewards.

The action-quality terms are implemented with public checker fields and action history. Evidence gain is computed for execute_sql and check_sql from improvements in evidence coverage. For example, check_sql may not improve the candidate query directly, but it can still be valuable if it provides public evidence and helps the agent decide whether to keep repairing or submit. Submit readiness is computed for submit_solution from the checker readiness score and whether the history contains supporting execute/check evidence. Tool cost is a small fixed penalty for execute/check actions. Additional protocol penalties handle invalid tool calls, truncated JSON, missing tool calls, and trajectories that use tools but never submit.

For process credit, the implementation uses improvement and harm thresholds of 0.10, a minimum positive confidence of 0.45, target-family weight 1.00, protected-family harm weight 1.25, and non-target positive weight 0.40. Positive module credit is capped at 0.06 per family and 0.12 per step; negative harm is capped at -0.08 per family and -0.16 per step. Evidence credit lies in $[0.00, 0.03]$, submit-readiness credit in $[-0.05, 0.04]$, execute/check/submit action costs are $-0.005/-0.003/0.0$, and process reward is clipped to $[-0.20, 0.15]$ per step and $[-0.50, 0.50]$ per trajectory. Protocol penalties are -0.08 for missing, invalid, truncated, or multiple tool calls, -0.10 for no submission, and $-0.12/-0.08/-0.05/-0.03$ for timeout, syntax error, execution error, and test failure. These caps keep process credit auxiliary: final task success remains the objective, while process credit shapes which action tokens receive higher or lower advantage.

F Data Construction and Filtering

SCOPE-SQL uses three data sources with different roles. The direct-repair SFT stage uses the canonical SIX-GYM-SQLite split to learn SQL repair from question, schema, issue SQL, and teacher SQL pairs. The tool-use SFT stage keeps only verified trajectories whose public observations are produced by the real tools and whose final candidates pass the evaluator. The RL pool contains 2,500 validator-accepted generated SQL-repair tasks, balanced between targeted-frontier and failure-refinement cells.

Generated tasks are accepted only after strict filtering. The teacher SQL must be executable and pass the evaluator. The issue SQL must expose a real repair target: it either executes to a behaviorally distinguishable wrong result or triggers a repairable parse, schema, or runtime failure. The Validator rejects near-duplicates, incompatible mutation profiles, invalid SQL, unsafe statements, and cases that do not expose a real repair objective. Accepted generated tasks are then re-evaluated with generated-candidate Monte Carlo sampling, so their difficulty is measured on the generated issue SQL itself rather than inherited from the source seed.

Data scale summary. Stage 1 uses 4,500 train / 500 dev direct-repair rows, Stage 2 uses 3,000 accepted verified tool-use trajectories, and the RL pool contains 2,500 validator-accepted generated repair tasks. We keep the appendix compact and do not expand every internal bucket because bucket assignments serve generator scheduling rather than evaluation metrics.

G Issue SQL Mutation Operators

Appendix Table 4 reorganizes the deterministic mutation operators by the F0–F6 structured semantic modules. Core operators are used by the main profile-constrained code generator, while targeted top-up operators are additional variants used to increase frontier coverage before generated-candidate Monte Carlo filtering.

H Prompt Templates

Figure 1 shows the prompt template used by the Repair Agent. During semantic-credit RL, the process reward does not ask any model to produce a repair. Instead, the training-only reward-side semantic checker receives public repair context and classifies the candidate SQL into F0–F6 semantic module states for reward computation. Figure 2 shows the input schema used by this reward-side semantic checker; this schema is not part of the agent-visible tool environment.

Semantic module	Code group(s)	Mutation operator(s)	Semantic perturbation
F0_schema_runtime	schema_runtime runtime_schema	same_runtime_error_variant (core)	Insert a missing-column reference after SELECT, preserving a schema/runtime failure mode.
F1_table_join	table_join relation_condition	join_type_flip (core) join_condition_to_true (top-up)	Change join cardinality by flipping join type, or make the first join predicate vacuous with ON 1 = 1.
F2_projection	projection result_semantics	projection_star (core) projection_add_distinct (top-up) projection_drop_distinct (top-up) projection_first_only (top-up) projection_last_only (top-up) projection_drop_last (top-up) projection_add_extra_literal (top-up)	Change selected attributes, distinctness, projection arity, or add an extra literal output column.
F3_predicate	predicate relation_condition	predicate_flip (core) predicate_drop (core) predicate_add_false_guard (top-up) predicate_add_true_disjunct (top-up) predicate_remove_where (top-up) predicate_like_to_not_like (top-up) predicate_not_like_to_like (top-up) predicate_in_to_not_in (top-up) predicate_not_in_to_in (top-up) predicate_between_to_not_between (top-up) predicate_not_between_to_between (top-up) numeric_literal_shift_-1 (top-up) numeric_literal_shift_+1 (top-up)	Flip comparison or set-membership semantics, remove or weaken filters, add always-false/always-true guards, or perturb predicate literals.
F4_aggregation_group	aggregation_group result_semantics	aggregate_function_swap (core) group_by_drop (core) aggregate_count_to_count_distinct (top-up) aggregate_count_distinct_to_count (top-up) aggregate_sum_to_avg (top-up) aggregate_avg_to_sum (top-up) aggregate_min_to_max (top-up) aggregate_max_to_min (top-up)	Change aggregation function, distinct aggregation, or grouping granularity.
F5_order_limit	order_limit result_semantics	order_direction_flip (core) limit_shift (core) order_by_drop (top-up) order_by_add_first_desc (top-up) limit_add_1 (top-up) limit_add_2 (top-up) limit_add_10 (top-up) limit_set_1 (top-up) limit_set_2 (top-up) limit_set_10 (top-up) limit_drop (top-up)	Change ranking direction, remove/add ordering, or add, overwrite, shift, or drop row limits.
F6_nested_composition	nested_composition complex_rewrite	subquery_condition_drop (core)	Remove a condition from nested SQLs, perturbing compositional constraints across query blocks.

Table 4: Issue-SQL mutation operators organized by F0–F6 structured semantic modules. The first column gives the structured semantic module perturbed by the generated issue SQL. The second column lists the corresponding implementation groups used by the code-driven generator controller. The third column lists the mutation operators; *core* denotes operators from the main deterministic generator catalog, and *top-up* denotes additional deterministic variants used to increase frontier coverage before final filtering. The last column states the semantic effect that the repair agent is expected to diagnose and undo.

I Constructive Failure Conversion Analysis

Outcome reward is progress-blind for failed trajectories, so we analyze 11,493 outcome-failed SCOPE-SQL RL rollouts with semantic-credit traces. Constructive failures make target-module progress without protected-module harm; destructive failures damage protected modules; stagnant failures show little structural change; and protocol failures violate the interaction protocol. For each failed rollout at step k, conversion means that the same task obtains a successful rollout after at least 20 or 40 later rollout steps.

J Case Studies and Public Observation Examples

We provide representative repair trajectories and public observation objects to show how semantic credit is grounded in tool-visible evidence. Each case reports the issue SQL, final submitted SQL, execution feedback, public `check_sql` module states, step-level semantic-credit signals, and final evaluator result. These examples complement the aggregate mechanism analysis in Appendix I; they are illustrative case studies rather than a full failure taxonomy.

Appendix Table 6 shows a complete SCOPE-SQL case example. The instance asks the agent to repair a SQLite string-comparison failure: the issue query filters `au_lname = 'white'` and

Prompt Template for SCOPE-SQL RL

You are SCOPE-SQL Repair Agent.
Repair the problematic SQLite SQL using public tool feedback only.
Tools are provided by the environment: `execute_sql(sql, role)` runs one public SQL statement, using `role="probe"` for evidence and `role="draft"` for candidate repairs; `check_sql(sql, role)` summarizes public F0–F6 module states for a candidate, without revealing hidden tests, gold SQL, or corrected SQL; `submit_solution(sql or sql_list)` submits the final SQL and ends the task.
Output rules: the entire assistant message must be exactly one `<tool_call>...</tool_call>`; use valid JSON inside the tags; do not output markdown fences, explanations, `<think>`, multiple tool calls, or text outside the tool call. Example: `<tool_call>{"name":"execute_sql","arguments":{"sql":"SELECT ...","role":"draft"}}</tool_call>`.
Repair rules: use CHECKSQL after meaningful repairs; use EXECUTESQL probes when schema, join, predicate, aggregation, ordering, or result shape is uncertain; preserve correct parts of the issue SQL unless public evidence supports changing them; treat checker module states and schema evidence as public diagnostics, not as a final-answer generator; never issue DML/DDL or multiple SQL statements; do not mention hidden tests, gold SQL, reward, or unavailable oracle information.
User message template. The user message contains {query}, {schema}, and {issue_sql}:
`## Problem Description`
{query}
`## Database Schema`
{schema}
`## Problematic SQL`
{issue_sql}
Repair the problematic SQL using the public tools. Generate only the next turn: `<tool_call>{"name":"...","arguments":{...}}</tool_call>`. Do not output markdown, explanation, or `<think>`. At most {max_turns} turns are allowed; if previous tool observations are present in the conversation, continue from them.

Figure 1: Prompt template used for SCOPE-SQL RL agents.

returns no rows because SQLite's default equality comparison is case-sensitive for this column. Appendix Table 7 adds a longer seven-turn SCOPE-SQL case example, where the agent repairs a date predicate over timestamp-valued invoice records. Both case examples use only public tool observations; the rollout metadata records `student_visible_no_teacher_sql=true`, `student_visible_no_hidden_tests=true`, and `student_visible_no_f_plan=true`; the accepted candidates pass final evaluation. For readability, the Observation fields in the case tables are abbreviated summaries of the public tool responses; Figures 4 and 5 show the unabridged public observation JSON object from Turn 6 of Appendix Table 7, with inline field comments.

Reward-side Semantic Checker Input Template

The checker command receives one JSON object on standard input:

```
{
"question": {question},
"schema": {schema},
"issue_sql": {issue_sql},
"current_sql": {current_sql},
"history": [
{
"action_type": "execute_sql | check_sql | submit_solution",
"role": "probe | draft | final",
"sql": {previous_sql},
"check": {previous_public_checker_state}
}
],
"task_type": "sql_repair",
"rules": [
"Return JSON only.",
"Classify F0-F6 module states.",
"Do not output corrected SQL, patch SQL, final SQL, or runnable SQL snippets."
]
}
```

The fields {question}, {schema}, {issue_sql}, and {current_sql} are public inputs to the repair instance. The `history` field contains only previous public tool actions and their public checker states. The reward-side checker output is consumed only as structured diagnostic state for process reward and is not returned to the Repair Agent; gold SQL, hidden tests, final reward, and target repairs are not included in this input.

Figure 2: Input template used by the training-only reward-side semantic checker. This component is used only during RL reward computation and is not part of the agent-visible tool environment.

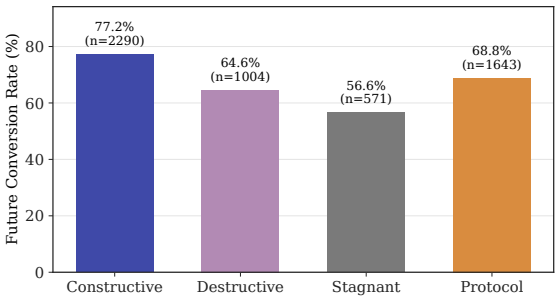

Figure 3: **Future conversion of outcome failures.** On SCOPE-SQL RL training rollouts, constructive failures have the highest probability of later converting to a successful rollout on the same task after at least 20 rollout steps.

Failure type	Share	$\Delta 20$	$\Delta 40$
Constructive	43.5%	77.2% (1767/2290)	77.1% (1576/2044)
Destructive	18.3%	64.6% (649/1004)	64.9% (576/888)
Protocol	27.4%	68.8% (1130/1643)	68.6% (1005/1464)
Stagnant	10.8%	56.6% (323/571)	56.0% (284/507)

Table 5: **Constructive failure conversion.** Share is computed over all 11,493 outcome-failed SCOPE-SQL RL rollouts. $\Delta 20$ and $\Delta 40$ report later success on the same task after at least 20 or 40 rollout steps.

Turn	Conversations	Public state
	SCOPE-SQL Case Example: Case-insensitive author lookup	
Turn1 Diagnosing the failing draft	**Task Define:** Retrieve author first and last names for last name "White" in a SQLite book-publishing database while making the predicate robust to user-input case. Source: SCOPE-SQL case example. **Issue SQL:** `SELECT "au_lname", "au_fname" FROM "authors" WHERE "au_lname" = 'white'` **Repaired SQL alias:** `SELECT "au_lname", "au_fname" FROM "authors" WHERE "au_lname" = 'white' COLLATE NOCASE` **Assistant:** `<tool_call>` name=`execute_sql`; arguments: sql=`issue_sql`, role=`draft`; `</tool_call>` **Observation:** tool=`execute_sql`; stage=`executable`; public error is empty; result shape has `columns=2` and `sample_rows=0`; module states include `F0=ok`, `F2=suspected_bad`, `F3=suspected_bad`, and `F5=stable`.	rows=0 F3: suspected public only
Turn2 Checking the original candidate	**Assistant:** `<tool_call>` name=`check_sql`; arguments: sql=`issue_sql`, role=`draft`; `</tool_call>` **Observation:** stage=`executable`; matching execute is seen; evidence coverage is `0.9`; result shape remains `columns=2`, `sample_rows=0`; submission value is `likely_continue` with score `0.45`; suggested next evidence is to repair the issue SQL; unresolved families are predicate and projection.	coverage=0.9 submit=0.45 continue
Turn3 Testing the repaired predicate	**Assistant:** `<tool_call>` name=`execute_sql`; arguments: sql=`repaired_sql`, role=`draft`; `</tool_call>` **Observation:** tool=`execute_sql`; stage=`executable`; public error is empty; result shape has `columns=2` and `sample_rows=1`; module states include `F0=ok`, `F2=suspected_bad`, `F3=stable`, and `F5=stable`.	rows=1 F3: stable improved
Turn4 Confirming submit readiness	**Assistant:** `<tool_call>` name=`check_sql`; arguments: sql=`repaired_sql`, role=`draft`; `</tool_call>` **Observation:** stage=`executable`; matching execute is seen; evidence coverage is `0.9`; result shape is `columns=2`, `sample_rows=1`; global readiness is `0.8678` with uncertainty `0.04`; submission value is `ready_to_submit`; suggested next evidence is to submit if no known requirement is still untested.	coverage=0.9 ready=0.8678 submit
Turn5 Submitting the final repair	**Assistant:** `<tool_call>` name=`submit_solution`; arguments: sql=`repaired_sql`; `</tool_call>` **Observation:** `submitted`. Final audit marks the rollout as accepted, checker bucket `ready_to_submit`, final pass `true`, and score `1.0`.	accepted final pass score=1.0

Table 6: SCOPE-SQL tool-use trajectory used as a case example.

Turn	Conversations	Public state
	SCOPE-SQL Case Example: Timestamp-aware invoice-date repair	
Turn1 Testing the issue predicate	**Task Define:** Retrieve `invoice` rows associated with January 1, 2021, including records whose timestamp falls in the late hours of December 31, 2020. Source: SCOPE-SQL case example. **Issue SQL:** `SELECT * FROM invoice WHERE invoice_date = '2021-01-01'` **Final SQL alias:** `SELECT * FROM invoice WHERE (date(invoice_date) = '2021-01-01' OR (invoice_date BETWEEN '2020-12-31 23:00:00' AND '2020-12-31 23:59:59'))` **Assistant:** `<tool_call>` name=`execute_sql`; arguments: sql=`issue_sql`, role=`probe`; `</tool_call>` **Observation:** stage=`executable`; public error is empty; result shape has `columns=9` and `sample_rows=0`; module states include `F0=ok`, `F2=stable`, and `F3=suspected_bad`.	rows=0 F3: suspected public only
Turn2 Checking the issue predicate	**Assistant:** `<tool_call>` name=`check_sql`; arguments: sql=`issue_sql`, role=`draft`; `</tool_call>` **Observation:** stage=`executable`; matching execute is seen; evidence coverage is `1.0`; result shape remains `columns=9`, `sample_rows=0`; submission value is `likely_continue` with score `0.45`; top unresolved family is `predicate`; suggested next evidence is to repair the issue SQL before considering submission.	coverage=1.0 submit=0.45 continue
Turn3 Testing a date-cast repair	**Assistant:** `<tool_call>` name=`execute_sql`; arguments: sql=`SELECT * FROM invoice WHERE date(invoice_date) = '2021-01-01'`, role=`draft`; `</tool_call>` **Observation:** stage=`executable`; public error is empty; result shape has `columns=9` and `sample_rows=1`; module states include `F0=ok`, `F2=suspected_bad`, and `F3=stable`.	rows=1 F3: stable partial repair
Turn4 Probing late previous-day records	**Assistant:** `<tool_call>` name=`execute_sql`; arguments: sql=`SELECT * FROM invoice WHERE invoice_date BETWEEN '2020-12-31 23:00:00' AND '2020-12-31 23:59:59' OR date(invoice_date) = '2021-01-01'`, role=`draft`; `</tool_call>` **Observation:** stage=`executable`; public error is empty; result shape has `columns=9` and `sample_rows=1`; module states keep `F0=ok` and `F3=stable`.	rows=1 F3: stable range tested
Turn5 Testing the final grouped predicate	**Assistant:** `<tool_call>` name=`execute_sql`; arguments: sql=`final_sql`, role=`draft`; `</tool_call>` **Observation:** stage=`executable`; public error is empty; result shape has `columns=9` and `sample_rows=1`; module states include `F0=ok`, `F2=suspected_bad`, and `F3=stable`.	rows=1 F3: stable final tested
Turn6 Confirming submit readiness	**Assistant:** `<tool_call>` name=`check_sql`; arguments: sql=`final_sql`, role=`draft`; `</tool_call>` **Observation:** stage=`executable`; matching execute is seen; evidence coverage is `1.0`; result shape is `columns=9`, `sample_rows=1`; global readiness is `0.8615`; submission value is `ready_to_submit`; suggested next evidence is to submit if no known requirement is still untested.	coverage=1.0 ready=0.8615 submit
Turn7 Submitting the final repair	**Assistant:** `<tool_call>` name=`submit_solution`; arguments: sql_list=[`final_sql`]; `</tool_call>` **Observation:** `submitted`. Final audit marks the rollout as accepted, checker bucket `ready_to_submit`, final pass `true`, and score `1.0`.	accepted final pass score=1.0

Table 7: Longer SCOPE-SQL tool-use trajectory used as a seven-turn case example.

Public Observation JSON With Field Comments

Turn: Turn6. Tool call: check_sql(sql=final_sql, role=draft).


```
{
    "allow_execute": false, # whether this checker call may execute SQL internally
  "check_available": true, # whether the checker returned a structured public observation
  "checker_profile": "sft_generation", # runtime profile used to build rollout observations
  "checker_profile_description": "SFT observation builder; aligned with CheckSQL", # human-readable checker profile description
  "checker_profile_effective_allow_execute": false, # effective execution permission after profile policy
  "checker_profile_execution_policy": "history_only", # checker uses only prior public tool history
  "checker_profile_payload_allow_execute": false, # execution permission recorded in the checker payload
  "checker_profile_timeout": 5.0, # checker-side timeout budget in seconds
  "checker_variant": "C1_code_only", # checker variant identifier
  "checker_version": "scope_sql_public_checker_v0.3_observation_guard_hard_timeout", # public checker implementation version
  "diff_summary": { # structural difference between issue SQL and candidate SQL
    "added_clause_count": 0, # number of newly added high-level SQL clauses
    "changed": true, # whether the candidate differs from the issue SQL
    "edit_ratio": 0.4167, # normalized edit magnitude
    "large_edit": true, # whether the edit magnitude crosses the large-edit threshold
    "removed_anchor_ratio": 0.0, # fraction of anchor structures removed from the issue SQL
    "removed_clause_count": 0, # number of high-level SQL clauses removed
    "summary": "clause structure preserved" # compact textual summary of structural changes
  },
    "disagreement_log": [], # internal checker disagreement records; empty here
  "evidence_coverage": 1.0, # fraction of expected public evidence covered by history
  "forbidden_information_used": false, # whether hidden tests, gold SQL, or oracle repair information was used
  "global": { # aggregate scores used by readiness and reward computation
    "evidence_coverage": 1.0, # global copy of public evidence coverage
    "preserve_risk": 0.35, # risk that the candidate changed previously correct behavior
    "submission_readiness": 0.86154, # aggregate readiness score for submitting
    "uncertainty": 0.0 # aggregate checker uncertainty
  },
    "matching_execute_seen": true, # whether this SQL was already observed by execute_sql
  "preserve_risk": {"bucket": "medium", "score": 0.35}, # discrete and numeric preserve-risk signal
  "public_error": "", # public runtime error string; empty means no public error
  "repair_mode_hint": "surgical", # checker hint about expected repair scope
  "result_shape": { # public output shape observed for the candidate
    "column_names": ["invoice_id", ..., "total"], # public output column names
    "columns": 9, # number of output columns
    "elapsed_ms": 8, # public execution latency in milliseconds
    "sample_rows": 1 # number of sample rows exposed
  },
    "schema_evidence": [], # additional public schema hints; empty here
  "stage": "executable", # candidate parses and executes publicly
  "...": "continued in Figure 5"
}
```


Figure 4: Complete public observation object from Turn 6 of the case example in Appendix Table 7, tool call `check_sql(sql=final_sql, role=draft)`. Part 1: top-level checker and execution fields.

Public Observation JSON With Field Comments (Continued)

Turn: Turn6. Tool call: `check_sql(sql=final_sql, role=draft)`.


```
{
    "module_states": { # F0–F6 public semantic module diagnostics
     "F0_schema_runtime": { # schema and runtime validity module
       "status": "ok", # module state label
       "confidence": 0.9, # checker confidence in this state
       "evidence": ["SQL parses and executes; columns=9 sample_rows=1"], # public evidence supporting the state
       "harm_risk": 0.0, # estimated risk for this module
       "score": 0.95 # numeric module score
     },
    "F1_table_join": {"status": "not_applicable", "confidence": 1.0, "evidence": [], "harm_risk": 0.0, "score": null}, # table/join module; no join
    repair target
     "F2_projection": { # projection and selected-column module
       "status": "suspected_bad", # module state label
       "confidence": 0.65, # checker confidence in this warning
       "evidence": ["unmatched identifier(s): between, date"], # public reason behind the warning
       "harm_risk": 0.65, # estimated projection harm risk
       "score": 0.35 # numeric projection score
     },
    "F3_predicate": { # predicate/filter module
       "status": "stable", # module state label
       "confidence": 0.65, # checker confidence in this state
       "evidence": ["predicate clauses present"], # public evidence for predicate structure
       "harm_risk": 0.05, # estimated predicate harm risk
       "score": 0.85 # numeric predicate score
     },
    "F4_aggregation_group": {"status": "not_applicable", "confidence": 1.0, "evidence": [], "harm_risk": 0.0, "score": null}, # aggrega-
    tion/grouping module; no aggregation repair target
     "F5_order_limit": {"status": "not_applicable", "confidence": 1.0, "evidence": [], "harm_risk": 0.0, "score": null}, # ordering/limit module;
    no order or limit target
     "F6_nested_composition": {"status": "not_applicable", "confidence": 1.0, "evidence": [], "harm_risk": 0.0, "score": null} # nested-SQL
    module; no nested repair target
   },
    "scores": { # numeric signals consumed by process reward and action-quality terms
     "evidence_coverage": 1.0, # public evidence coverage component
     "preserve_risk": 0.35, # risk penalty component
     "public_fix_score": 0.8226, # aggregate public-fix quality score
     "submission_value": 0.8615, # numeric value for submitting the current candidate
     "uncertainty": 0.0, # uncertainty component
     "unresolved_score": 0.4225 # aggregate unresolved-issue score
   },
    "submission_value": {"bucket": "ready_to_submit", "score": 0.86154}, # discrete and numeric submit-readiness signal
   "suggested_next_evidence": ["submit if no known requirement is still untested"], # list-form recommendation for next action
   "suggested_next_evidence_text": "submit if no known requirement is still untested", # text-form recommendation for next action
   "top_unresolved_family": "projection", # highest-priority unresolved semantic family
   "uncertainty": {"bucket": "low", "score": 0.0}, # discrete and numeric checker uncertainty
   "unresolved_families": [ # structured unresolved semantic warnings
     {"family": "projection", "confidence": 0.65, "public_reason": "unmatched identifier(s): between, date", "severity": 0.65} # one unresolved
    warning exposed to the agent
   ],
    "unresolved_score": 0.4225 # top-level unresolved-score copy
}
```


Figure 5: Complete public observation object from Turn 6 of the case example in Appendix Table 7, tool call `check_sql(sql=final_sql, role=draft)`. Part 2, continued: module states, scores, and next-action signals.