Progress-SQL: Improving Reinforcement Learning for Text-to-SQL via Progressive Rewards

Shihao Zhang, Xiaoman Wang, Yuan Liu, Yunshi Lan*, Weining Qian
Department of Data Science and Engineering, East China Normal University
{shzhang, xmwang, 52285903014}@stu.ecnu.edu.cn,
{yslan, wnqian}@dase.ecnu.edu.cn

Abstract

Reinforcement learning has recently shown promise in improving large language models for Text-to-SQL generation, yet existing methods typically optimize one-shot rewards defined over a single SQL state. Such rewards provide limited guidance for iterative SQL correction and are insufficient to capture the improvement of multi-turn SQL refinement. In this paper, we propose Progress-SQL, a multi-turn reinforcement learning framework with progressive rewards for Text-to-SQL. Our approach introduces an Oracle-guided Diagnostic Tree (ODT), which abstracts SQL queries into clause-level structural profiles and produces diagnostic feedback for next-turn refinement. To provide dense and robust reward signals, we combine ODT-based structural alignment with lexical alignment and define a progressive reward that measures the improvement from the initial SQL to the final SQL. We further incorporate a progression latency reward that favors earlier correctness and an execution status reward that encourages recovery from the invalid SQL. Experiments on BIRD, Spider, and Spider robustness variants demonstrate that our method consistently improves Text-to-SQL performance across both primary and robustness evaluations. Our code is released at https://github.com/YooYoo67/ProgressSQL.

1 Introduction

Large Language Models (LLMs) have significantly advanced Text-to-SQL parsing (Li et al., 2024, 2025). Recent Reinforcement Learning (RL) methods further improve these models by optimizing one-shot rewards (Pourreza et al., 2025b; Zhang et al., 2026; Weng et al., 2025; Ma et al., 2026), where the single turn rollout is measured by a one-shot reward. This reward is computed solely based on the execution result of the SQL. However, such a reward is often sparse, which provides limited guidance on SQL generation. This results in inefficient optimization for the exploration of correct SQLs, especially for complex SQLs involving joins and aggregation (Pourreza et al., 2025b).

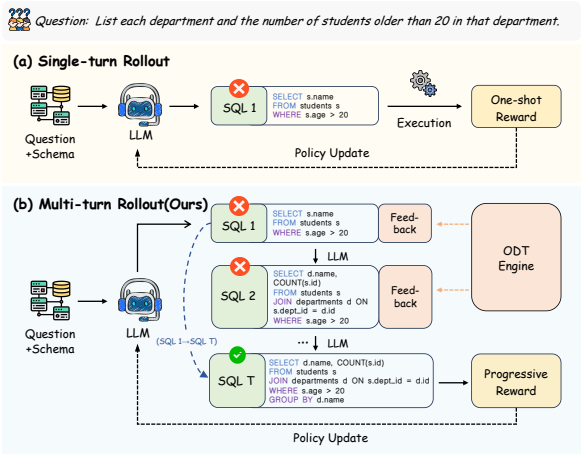


Figure 1: Comparison of reward paradigms. **(a) Single-turn Rollout**: the policy model generates a single SQL and receives a reward signal after execution. **(b) Multi-turn Rollout with Progressive Reward (Ours)**: the policy model iteratively refines its SQL over T turns guided by ODT engine. The progressive reward measures improvement from the first SQL to the final SQL.

A cutting-edge study, SkyRL-SQL (Liu et al.), introduced multi-turn rollout for the Text-to-SQL task, where multi-turn interaction is conducted between the LLMs and engines. The LLM generates a sequence of SQLs for a question. For each turn, the LLM obtains the execution result from the engine and takes it into consideration for the next-turn generation. Nevertheless, SkyRL-SQL collects the reward based on the last-turn generation, which cannot yet break away from the limitation of the one-shot reward. In other words, **one-shot reward is not enough to capture the dynamic behavior of multi-turn rollout**.

To address this limitation, we propose Progress-SQL, a multi-turn RL framework with progressive

*Corresponding author.

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 17675–17695
October 24-29, 2026 ©2026 Association for Computational Linguistics

rewards for Text-to-SQL, as shown in Figure 1. Specifically, we first introduce an Oracle-guided Diagnostic Tree (ODT), which abstracts SQL queries into clause-level structural profiles and generates diagnostic feedback for next-turn refinement. By comparing the predicted ODT with the gold ODT during training, the model can revise its SQL prediction according to the structured feedback. Unlike one-shot rewards, our progressive reward is defined over the SQL trajectory and measures whether the final SQL improves over the initial SQL in terms of structural and lexical alignment. Together with a progression latency reward and an execution status reward, the objective favors trajectories that improve effectively, reach correctness earlier, and recover from invalid SQL predictions. We evaluate our RL method on widely used Text-to-SQL benchmarks, including BIRD and Spider. Based on the 7B backbone, our method improves the base model by an average of 8.5% in execution accuracy across BIRD Dev, Spider Dev, and Spider Test, and by 6.3% in test-suite accuracy on Spider Dev. Compared with LLMs tuned by existing RL methods, our method achieves competitive or superior performance after fine-tuning.

Our contributions are summarized as follows:

- We propose Progress-SQL, a multi-turn RL framework for Text-to-SQL. By defining an ODT for clause-level SQL diagnosis, Progress-SQL collects fine-grained feedback for next-turn SQL generation.
- We design a progressive reward that explicitly measures the improvement from the initial SQL to the final SQL, complemented by early-correctness and execution-status rewards for efficient and robust refinement.
- Extensive experiments on multiple Text-to-SQL benchmarks demonstrate that our method consistently improves both execution accuracy and test-suite accuracy with different base models.

2 Related Work

2.1 Text-to-SQL with Large Language Models

Due to the outstanding performance of LLMs in various NLP tasks, we observe a methodology trend of Text-to-SQL systems from heuristic rules and deep learning to LLMs (Zelle and Mooney, 1996; Popescu et al., 2003; Li and Jagadish, 2014; Yu et al., 2018a; Wang et al., 2020; Tao et al., 2024). In the early stages, LLM-based Text-to-SQL systems relied heavily on In-Context Learning (ICL) and structured prompting strategies. DIN-SQL (Pourreza and Rafiei, 2023) utilizes prompting instruction to decompose complex queries into sub-problems, while DAIL-SQL (Gao et al., 2024) constructs efficient few-shot demonstrations via question skeleton matching. More recent pipelines such as CHASE-SQL (Pourreza et al., 2025a) and XiYan-SQL (Liu et al., 2026) further combine candidate generation and selection strategies to refresh the accuracy on competitive benchmarks. Alongside prompting, post-training has become a standard paradigm (Li et al., 2024, 2025), improving the Text-to-SQL performance for open-source models in curated training splits.

2.2 Reinforcement Learning for Text-to-SQL

Regarding the post-training procedure, recent studies have framed Text-to-SQL as a sequential decision-making problem optimized with reinforcement learning, where reward design is crucial to measuring SQL quality. Recent studies have explored RL for Text-to-SQL by exploring different rewards. The most direct signal is binary execution accuracy (EX), but it is highly sparse because rewards are obtained only when the generated SQL matches the gold execution result. To provide denser supervision, Reward-SQL (Zhang et al., 2026) introduces a Process Reward Model for step-wise reasoning supervision, while Reasoning-SQL (Pourreza et al., 2025b) designs SQL-specific partial rewards such as schema-linking accuracy, n-gram similarity, and syntax validity. Graph-Reward-SQL (Weng et al., 2025) further incorporates structural tree matching to capture logical alignment. SkyRL-SQL (Liu et al.) extends RL to multi-turn SQL refinement, but its reward is still derived from the final SQL state. However, these methods mainly define rewards over single or final SQL states, leaving trajectory-level revision behavior underexplored. However, these methods mainly define rewards over single or final SQL states, leaving the trajectory-level revision behavior underexplored.

3 Preliminaries

3.1 Problem Definition

The Text-to-SQL task can be formulated as a semantic parsing problem that translates a natural language question into an executable SQL query.

Formally, let $q = \{q_1, q_2, \ldots, q_l\}$ denote the natural language question and S denote the corresponding database schema (comprising tables, columns, and foreign key constraints). Given the input context $x = (q, S)$, the goal is to generate a target SQL query $y = \{w_1, w_2, \ldots, w_m\}$ which consists of a sequence of tokens and can correctly retrieve the answers from the database engine $\mathcal{E}$.

3.2 Reinforcement Learning Protocol

Existing studies (Schulman et al., 2017) view the generation process through a parameterized policy $\pi_\theta(y \mid x)$, which is initialized from a pre-trained instruction-tuned LLM. We denote it as the reference policy π_{ref}. The Text-to-SQL task can be solved via the reinforcement learning protocol, which optimize the policy model by maximizing:

$$\begin{aligned}\mathcal{J}(\theta) =& \mathbb{E}_{x\sim P(x), y\sim\pi_\theta}[\mathcal{R}(y, y^*) \\ &- \beta \mathbb{D}_{\text{KL}}(\pi_\theta(\cdot|x) \| \pi_{\text{ref}}(\cdot|x))],\end{aligned}$$

where x is sampled from a Text-to-SQL dataset, π_{ref} is the initial reference model, and π_θ is iteratively updated. KL divergence penalty prevents the updated policy from degrading its fundamental language capabilities during training. Reward function measures the distance between the generated SQL query y and the gold SQL query y^*. The similar protocol with different RL algorithms such as GRPO (Shao et al., 2024), GSPO (Zheng et al., 2025) are also widely utilized to solve the task.

3.3 One-shot Reward Design

In RL protocol, one of the core objectives is to maximize the reward $\mathcal{R}(y, y^*)$, which can be deemed as a one-shot measurement between the generated SQL and the gold SQL. We notice that a number of one-shot rewards have been proposed for solving Text-to-SQL task, which can be summarized below.

Execution Matching. Measuring the consistency of the execution results between the generated SQL and the gold SQL is an intuitive way to evaluate the policy model. Standard approach formulates it as a binary reward (1 for exact match, 0 otherwise) (Pourreza et al., 2025b). Due to the sparse reward signal, recent studies improve the reward by introducing fractional execution based on the proportion of matching columns and cells (Hao et al., 2025; Papicchio et al., 2025).

Query Matching. To construct a denser reward landscape, researchers incorporate static structural similarities between y and y^*. Reasoning-SQL (Pourreza et al., 2025b) calculates the Jaccard similarity of extracted schema items and 2-grams between the generated and the annotated SQLs, thus guiding the query alignment beyond database execution. Besides the semantics matching, syntax is also considered in the reward (Pourreza et al., 2025b; Ali et al., 2025).

Format and Process Regularization. With the advent of reasoning models (*e.g.*, DeepSeek-R1 (Guo et al., 2025)), enforcing specific Chain-of-Thought (CoT) behaviors has become integral to reward design. Models are incentivized via format rewards to encapsulate their reasoning processes within specific tags (*e.g.*, `<think>` and `<sql>`) (Ma et al., 2026; Papicchio et al., 2025; Pourreza et al., 2025b). Furthermore, to prevent reward hacking and excessive verbosity, regularization terms such as tag count (Papicchio et al., 2025) and length penalties (Ma et al., 2026) are applied to penalize redundant reasoning. And some studies even involve schema keywords, runtime log in the rewards (Berdnyk and Collery, 2025; Ma et al., 2026).

4 Methodology

4.1 Multi-turn Rollout with ODT Feedback

Standard RL methods for Text-to-SQL usually implement single-turn rollout, where the policy generates a SQL query once and receives sparse rewards from the single decoded SQL. To address this limitation, we extend the standard rollout process into a multi-turn SQL debugging trajectory. Inspired by SkyRL-SQL (Liu et al.), we allow the policy model to iteratively revise its SQL prediction based on feedback from previous turns as shown in Figure 2. Remarkably, we introduce an *Oracle-guided Diagnostic Tree* (ODT) as the structural feedback after each attempt to refine the next-turn rollout.

Formally, given a user question q and database schema S, the initial input is defined as:

$$x^{(1)} = (q, S).$$

At the t-th turn, the policy model π_θ generates a new SQL prediction as the rollout with the consideration of the input $x^{(t)}$:

$$y^{(t)} \sim \pi_\theta(\cdot \mid x^{(t)}).$$

Regarding $x^{(t)}$, we construct the input by appending ODT structural feedback generated from

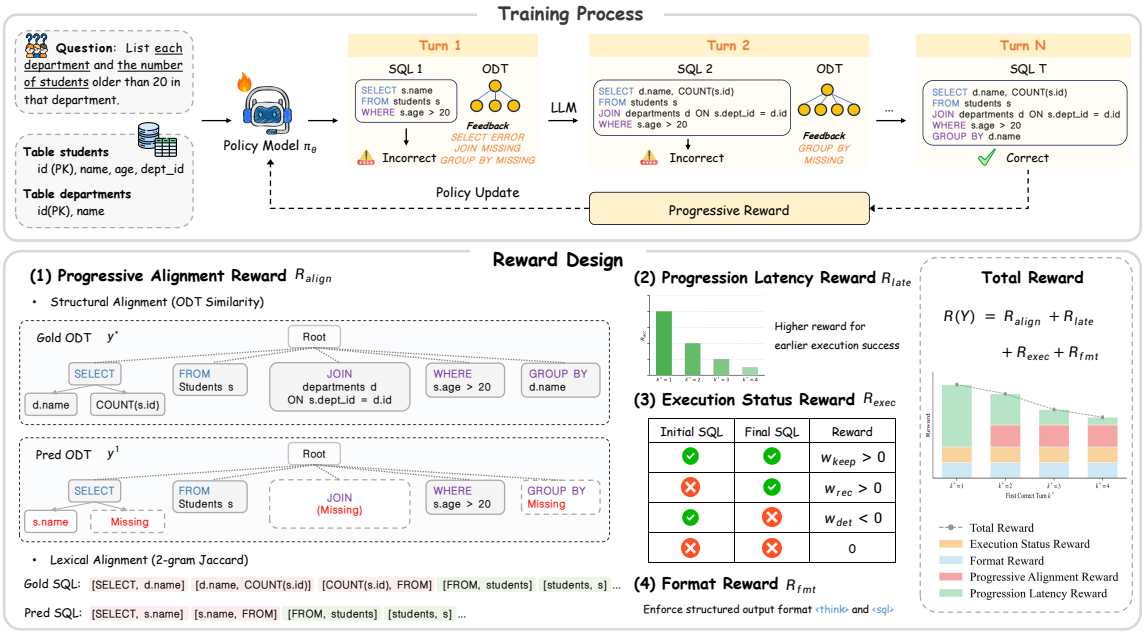


Figure 2: Overall framework of Progress-SQL, our multi-turn reinforcement learning method for Text-to-SQL. The policy model iteratively generates SQL queries and receives ODT-based diagnostic feedback after each execution. The final trajectory is optimized using progressive rewards that combine structural/lexical alignment improvement, progression latency reward, execution-status transition reward, and format reward.

the previous prediction:

$$f^{(t-1)} = \mathrm{ODT}(y^{(t-1)}, y^*),$$
$$x^{(t)} = (q, S, y^{(1)}, f^{(1)}, \ldots, y^{(t-1)}, f^{(t-1)}).$$

The rollout continues until the model produces a correct executable SQL or the maximum interaction budget T is reached. This process yields a multi-turn trajectory $Y = (y^{(1)}, y^{(2)}, \ldots, y^{(T)})$, which is then evaluated by our progressive rewards.

Oracle-guided Diagnostic Tree. Above, directly comparing a predicted SQL with the gold SQL is non-trivial due to the compositional and nested syntax of SQL queries. Following the common practice of using Abstract Syntax Trees (ASTs) to capture SQL syntactic structures, we parse both the predicted SQL y and the gold SQL y^* into ASTs. Rather than directly matching raw AST nodes, we abstract each SQL into an Oracle-guided Diagnostic Tree (ODT). As shown in Figure 2, each node in the tree represents a clause-level structural profile, such as selected columns, join signatures, filtering predicates, grouping columns, ordering clauses, or nested subqueries. By comparing the ODTs of y and y^*, the diagnostic module produces two outputs: a structural similarity score and a set of clause-level error tags. The error tags are verbalized into diagnostic feedback for the next-turn refinement, while the structural similarity score serves as the structural alignment term in our reward function (refer to Section 4.2). For example, in Figure 2, the first-turn SQL only selects student names with an age predicate, while the gold SQL requires department-level aggregation by joining `students` with `departments` and grouping by department. The ODT comparison detects mismatches in selection, join, and grouping structures, producing tags such as `SELECT_ERROR`, `JOIN_MISSING`, and `GROUP_BY_MISSING`. These tags are verbalized as feedback to guide the next-turn SQL refinement.

ODT acts as a fixed, non-differentiable diagnostic component of the environment. It compares the predicted SQL with the gold SQL and appends discrete error tags as next-turn observations; these tags do not participate in back-propagation, and gradients are computed only through the policy model over generated tokens. Details of the multi-turn algorithm and ODT construction are provided in Appendix C and Appendix D, respectively.

4.2 Progressive Reward Formulation

Given a multi-turn SQL trajectory $Y = (y^{(1)}, y^{(2)}, \ldots, y^{(T)})$ and the gold SQL y^*, our goal is not only to evaluate the final prediction, but also to measure whether the trajectory makes meaningful progress. We therefore design a comprehensive reward composed of four parts: *Progressive Alignment Reward*, *Progression Latency Reward*,

Execution Status Reward, and *Format Reward*.

Progressive Alignment Reward. To measure the similarity between the predicted and ground truth SQLs, we define both structural and lexical aspects of alignment:

- **Structural Alignment.** The structural alignment score is computed by the ODT tree described above. Specifically, we abstract both the generated SQL and the gold SQL into ODTs and compare their clause-level structural profiles. To focus on structural correspondence, lexical values are normalized or replaced with placeholders before matching. For each node, we compute a similarity score by combining local clause similarity and child-subtree similarity:

$$s_{\text{node}} = \alpha \cdot s_{\text{local}} + (1-\alpha) \cdot s_{\text{child}},$$

  where s_{local} measures the weighted Jaccard similarity over clause-level feature sets. For nested structures, s_{child} is computed recursively by matching child subtrees under node-type compatibility constraints. The final structural alignment score is given by the root node score:

$$\mathcal{F}_{\text{struct}}(y, y^*) = s_{\text{root}}.$$

  Hence, structural alignment measures clause-level structural consistency between y and y^* while reducing sensitivity to surface lexical differences. The detailed ODT construction and scoring are provided in Appendix D.

- **Lexical Alignment.** Structural parsing may fail when the generated SQL contains severe syntax errors, especially during early training. To avoid zero-reward regions in such cases, we further introduce a lexical fallback score. Specifically, we compute the 2-gram Jaccard similarity between the tokenized generated SQL and the gold SQL, denoted as $\mathcal{F}_{\text{lex}}(y, y^*)$. The lexical score provides a dense signal when structural parsing is unavailable.

Rather than rewarding only endpoint correctness, we explicitly measure whether the final SQL improves over the initial prediction. The final alignment score is defined as:

$$\begin{aligned}\mathcal{F}(y, y^*) =& \frac{1}{2}\left(\mathcal{F}_{\text{struct}}(y, y^*) + \mathcal{F}_{\text{lex}}(y, y^*)\right), \\ \Delta =& \mathcal{F}(y^{(T)}, y^*) - \mathcal{F}(y^{(1)}, y^*).\end{aligned}$$

This endpoint-difference design reduces the influence of intermediate oscillations and encourages the trajectory to end with a better SQL query than it starts with.

The progressive reward is defined as:

$$\mathcal{R}_{\text{align}} = \begin{cases} \omega^{+}_{\text{align}} \cdot \Delta, & \text{if } \Delta > 0, \\ \omega^{-}_{\text{align}}, & \text{if } \Delta \leq 0, \end{cases} \tag{1}$$

where $\omega^{+}_{\text{align}}$ controls the reward for positive progress and $\omega^{-}_{\text{align}}$ penalizes stagnant trajectories. This encourages the model to perform meaningful revisions instead of meaningless repeats.

Progression Latency Reward. To encourage immediate progression to SQL correction, we terminate the rollout once the generated SQL first matches the execution result of the gold SQL. Let k^* denote the first successful turn, where success means execution result matches with the gold SQL. We apply a geometric decay to late corrections:

$$\mathcal{R}_{\text{late}} = \begin{cases} \omega_{\text{acc}} \cdot \gamma^{k^*-1}, & 1 \leq k^* \leq T, \\ 0, & \text{otherwise}, \end{cases} \tag{2}$$

where $\omega_{\text{acc}} > 0$ controls the base reward for execution correctness and $\gamma \in (0, 1)$ is the per-turn decay factor. Earlier successful turns therefore receive larger rewards, encouraging the model to produce a correct SQL faster.

Execution Status Reward. Execution correctness can be too sparse, but executability itself provides useful information about whether the model is recovering from invalid SQL. We therefore reward status transitions in executability between the initial and final predictions. Let $\mathcal{E}(y)$ be a boolean indicator that returns true if y can be executed by the database engine without syntax or runtime errors. We define:

$$\mathcal{R}_{\text{exec}} = \begin{cases} \omega_{\text{keep}}, & \text{if } \mathcal{E}(y^{(1)}) \wedge \mathcal{E}(y^{(T)}), \\ \omega_{\text{rec}}, & \text{if } \neg\mathcal{E}(y^{(1)}) \wedge \mathcal{E}(y^{(T)}), \\ \omega_{\text{det}}, & \text{if } \mathcal{E}(y^{(1)}) \wedge \neg\mathcal{E}(y^{(T)}), \\ 0, & \text{otherwise}. \end{cases} \tag{3}$$

Here, $\omega_{\text{keep}} > 0$ rewards SQLs that remain executable across refinement, $\omega_{\text{rec}} > 0$ rewards recovery from an initially unexecutable SQL to an executable one, and $\omega_{\text{det}} < 0$ penalizes deterioration from an executable SQL to an invalid one.

Format Reward. Following prior endpoint-reward designs (Ma et al., 2026; Papicchio et al., 2025; Pourreza et al., 2025b), we define the format reward as $\mathcal{R}_{\text{fmt}} = \omega_{\text{fmt}}$ if the final output follows the required `<think>` and `<sql>` templates, and $\mathcal{R}_{\text{fmt}} = 0$ otherwise.

Overall Reward. The final trajectory-level reward is the sum of all components:

$$\mathcal{R}(Y) = \mathcal{R}_{\text{fmt}} + \mathcal{R}_{\text{exec}} + \mathcal{R}_{\text{late}} + \mathcal{R}_{\text{align}}. \quad (4)$$

This reward jointly evaluates format validity, executability transition, endpoint correctness, and the improvement of progressive alignment over the multi-turn trajectory.

Reward Analysis. Figure 2 illustrates the theoretical upper bound of reward under different first correct turn k^* with keeping format and execution status unchanged. We elaborately configure the defined weights (refer to Appendix B) to show an overall downward trend as shown in the Figure 2. This upper-bound trend shows that the reward design explicitly favors earlier positive progression and also encourages the success of the earlier rollout.

5 Experiments

5.1 Experimental Setup

Datasets. We conduct experiments on widely recognized Text-to-SQL benchmarks:

- **BIRD** (Li et al., 2023): A large-scale, cross-domain dataset featuring complex reasoning questions and real-world database schema. It is highly challenging and serves as our primary testbed for multi-turn reasoning capabilities.
- **Spider & its variants**: We also evaluate on the standard **Spider** (Yu et al., 2018b) dataset to assess cross-domain generalization. To further test the model's robustness against schema synonyms and domain knowledge perturbations, we report results on its challenging variants: **Spider-Syn** (Gan et al., 2021a), **Spider-Realistic** (Deng et al., 2021), and **Spider-DK** (Gan et al., 2021b).

Evaluation Metrics. We employ two standard automatic evaluation metrics in recent Text-to-SQL research:

- **Execution accuracy (EX)**: Measures whether the execution output of the generated SQL matches the exact result of the ground-truth SQL on the target database.
- **Test-Suite accuracy (TS)**: A more rigorous metric that evaluates the generated SQL across multiple augmented database test cases to ensure generalization and prevent false positives caused by coincidental execution matches.

Baselines. We compare our models against two categories of strong baselines, corresponding to the groupings in Table 1.

- **Base and Supervised Fine-tuned Models**: This category includes our backbone models, Qwen2.5-Coder-7B/14B-Instruct, evaluated before applying our RL training, as well as supervised fine-tuned Text-to-SQL models such as OmniSQL (Li et al., 2025) and SFT CodeS (Li et al., 2024). This comparison measures the gains brought by our RL framework beyond standard supervised or instruction-tuned models.
- **Comparative RL methods**: This category covers recent Text-to-SQL systems trained with reinforcement learning or reasoning-oriented post-training objectives, including both single-turn and multi-turn RL methods. All models in this group are based on Qwen2.5-Coder-7B-Instruct, making them directly comparable to our 7B model under the same backbone family. Specifically, we include Reasoning-SQL (Pourreza et al., 2025b), Reward-SQL (Zhang et al., 2026), Graph-Reward-SQL (Weng et al., 2025), SQL-R1 (Ma et al., 2026), SkyRL-SQL (Liu et al.), and SQL-Trail (Hua et al., 2026). The first four methods use single-turn rollouts with one-shot rewards, while SkyRL-SQL and SQL-Trail introduce multi-turn refinement but still optimize rewards mainly associated with final SQL states. These baselines provide direct comparisons to our progressive multi-turn RL framework.

Implementation Details. We adopt Qwen2.5-Coder-7B/14B-Instruct (Hui et al., 2024) as base models and train them with GRPO (Shao et al., 2024) using the `verl` framework (Sheng et al., 2025). Following prior multi-turn Text-to-SQL training setups (Hua et al., 2026), we sample $G = 8$ rollouts per input with temperature $\tau = 1.0$. Method-specific hyperparameters are selected by small-scale pilot experiments on the development set and then kept fixed across all experiments, with the maximum interaction turn set to $T = 4$ and the per-turn decay factor set to $\gamma = 0.5$. ODT is used only during training for feedback construction and

Model	Reward	BIRD Dev	Spider Dev		Spider Test
		EX(%)	EX(%)	TS(%)	EX(%)
Base and Supervised Fine-tuned Models					
Qwen2.5-Coder-7B-Instruct	–	52.3	79.7	73.8	81.8
Qwen2.5-Coder-14B-Instruct	–	60.2	83.4	77.0	86.1
OmniSQL-7B (Li et al., 2025)	–	63.0	85.8	81.0	87.2
SFT CodeS-7B (Li et al., 2024)	–	57.2	85.4	80.3	–
Other Post-training Methods					
Reasoning-SQL-7B (Pourreza et al., 2025b)	One-shot	64.0	78.7	–	–
Reward-SQL-7B (Zhang et al., 2026)	One-shot	59.7	77.1	–	–
Graph-Reward-SQL-7B (Weng et al., 2025)	One-shot	63.0	81.6	–	–
SQL-R1-7B† (Ma et al., 2026)	One-shot	59.8	81.8	67.6	81.7
SkyRL-SQL-7B† (Liu et al.)	One-shot	60.2	84.5	74.9	85.7
SQL-Trail-7B† (Hua et al., 2026)	One-shot	57.0	85.6	77.4	84.0
Our Reinforcement Learning Models					
Progress-SQL-7B(Ours)	Progressive	64.3(+12.0)	87.1(+7.4)	80.9(+7.1)	87.8(+6.0)
Progress-SQL-14B(Ours)	Progressive	67.5(+7.3)	87.3(+3.9)	80.4(+3.4)	88.1(+2.0)

Table 1: Performance comparison on the primary Text-to-SQL benchmarks. All comparative 7B baselines post-trained by RL algorithm are based on Qwen2.5-Coder-7B-Instruct. We report EX and TS if they are available. Numbers in red fonts indicate absolute gains over the same-scale base model. † denotes baselines reproduced using the same training mixture as Progress-SQL, while following the prompt templates, interaction procedures, reward formulations, and optimization settings described in the original papers. No additional task-specific supervised fine-tuning is applied.

progressive reward computation; evaluation follows standard single-turn inference without ODT feedback, with transfer behavior analyzed in Appendix E.4. Test results are reported with majority voting over 8 samples (Vote@8). Full hyperparameters are provided in Appendix B.

5.2 Main Results

Table 1 reports the performance on the widely recognized Text-to-SQL benchmarks. We have the following observations accordingly:

(1) LLMs fine-tuned by our RL method consistently improve over their corresponding base models, which indicates the generalization of our RL method. For the 7B backbone LLMs, Progress-SQL achieves an average EX improvement of 8.5% across the three benchmarks, together with a 7.1% TS improvement on Spider Dev. For the 14B backbone LLMs, Progress-SQL achieves an average EX improvement of 4.4% across the three benchmarks, together with a 3.4% TS improvement on Spider Dev. (2) Compared with existing post-training methods, our approach achieves competitive or superior performance across the evaluated benchmarks. In particular, Progress-SQL-7B reaches 87.1% EX on Spider Dev and 87.8% EX on Spider Test, surpassing recent 7B SQL RL methods (e.g. SQL-R1-7B and SkyRL-SQL-7B) on these benchmarks. Progress-SQL-14B further improves the results, suggesting that our progressive rewards can scale to stronger base models.

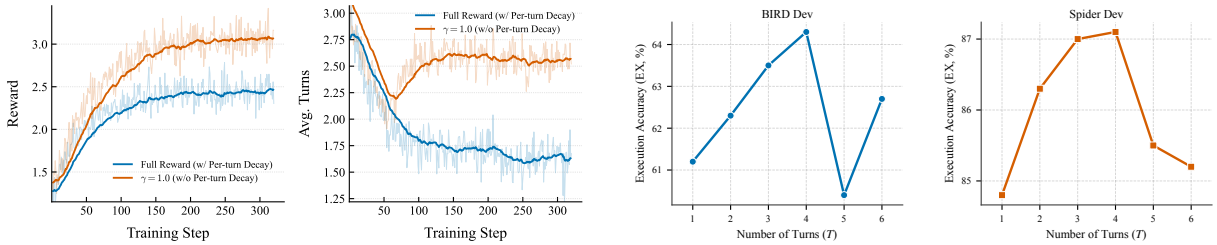
5.3 Further Analysis

Robustness Evaluation. Table 2 presents the robustness evaluation on Spider-Syn, Spider-Realistic, and Spider-DK, which include various knowledge perturbation. Our models consistently improves over the corresponding Qwen2.5-Coder base models across all three variants, which showcases the robustness of our method on different Text-to-SQL scenarios. Moreover, our models achieve competitive results compared with existing post-training methods across the Spider variants, suggesting superior effect of post-training for Text-to-SQL.

Ablation Study. We conduct an ablation study on BIRD Dev and Spider Dev. Table 3 reports the results under Vote@8 decoding. As we can see, iteratively removing components from the comprehensive reward leads to performance drop on both benchmarks. The Single-turn RL variant removes the multi-turn rollout and the progressive alignment reward $\mathcal{R}_{\text{align}}$, reducing our framework to a standard one-shot RL setting where the model generates a single SQL and is optimized using only final-state rewards. Among all the variants, removing per-turn decay causes the largest performance drop. This indicates that per-turn decay is critical for preventing the policy from over-relying on the late feedback. Figure 3a further illustrates the train-

Model	Reward	Spider-Syn		Spider-Realistic		Spider-DK
		EX(%)	TS(%)	EX(%)	TS(%)	EX(%)
Base and Supervised Fine-tuned Models						
Qwen2.5-Coder-7B-Instruct	–	70.9	62.3	74.4	66.9	66.9
Qwen2.5-Coder-14B-Instruct	–	72.3	65.6	81.5	74.6	72.3
OmniSQL-7B (Li et al., 2025)	–	77.0	70.2	80.5	75.0	74.6
SFT CodeS-7B (Li et al., 2024)	–	76.9	70.0	82.9	77.2	72.0
Other Post-training Methods						
Reasoning-SQL-7B (Pourreza et al., 2025b)	One-shot	69.3	–	–	–	73.3
SQL-R1-7B† (Ma et al., 2026)	One-shot	73.7	58.0	80.1	61.6	69.9
SkyRL-SQL-7B† (Liu et al.)	One-shot	72.1	62.2	83.5	73.0	70.7
SQL-Trail-7B† (Hua et al., 2026)	One-shot	73.7	65.1	83.7	74.4	75.4
Our Reinforcement Learning Models						
Progress-SQL-7B(Ours)	Progressive	76.6 (+5.7)	68.8 (+6.5)	84.1 (+9.7)	76.4 (+9.5)	75.9 (+9.0)
Progress-SQL-14B(Ours)	Progressive	77.1 (+4.8)	69.1 (+3.5)	84.1 (+2.6)	76.6 (+2.0)	76.4 (+4.1)

Table 2: Robustness evaluation on Spider variants. All comparative 7B RL baselines are based on Qwen2.5-Coder-7B-Instruct. We report EX and TS if they are available. Numbers in red fonts denote absolute gains over the same-scale base model. † denotes baselines reproduced using the same training mixture as Progress-SQL, while following the prompt templates, interaction procedures, reward formulations, and optimization settings described in the original papers. No additional task-specific supervised fine-tuning is applied.



(a) Effects of per-turn decay.

(b) Effects of maximum interaction turns T.

Figure 3: Effects of per-turn decay and interaction budget. (a) Removing per-turn decay leads to less stable multi-turn optimization and longer correction trajectories. (b) Performance improves up to $T = 4$ and then decreases when more interaction turns are allowed.

Variant	BIRD	Spider	
	EX(%)	EX(%)	TS(%)
Full	**64.3**	**87.1**	**80.9**
Single-turn RL	61.2 (-3.1)	84.8 (-2.3)	76.9 (-4.0)
w/o $\mathcal{F}_{\text{struct}}$	61.6 (-2.7)	85.5 (-1.6)	78.4 (-2.5)
w/o $\mathcal{F}_{\text{lex}}$	62.1 (-2.2)	85.9 (-1.2)	78.8 (-2.1)
w/o $\mathcal{R}_{\text{exec}}$	63.1 (-1.2)	86.5 (-0.6)	78.1 (-2.8)
$\gamma = 1$	47.5 (-16.8)	61.3 (-25.8)	56.5 (-24.4)

Table 3: Ablation study on BIRD Dev and Spider Dev under Vote@8 decoding. Numbers in blue fonts in parentheses denote absolute drops compared with the full reward.

ing dynamics behind this degradation. Without per-turn decay, the model is not penalized for solving the query in later turns and therefore tends to rely more heavily on ODT feedback to progressively repair its SQL. This increases the average number of interaction turns and allows the trajectory-level reward to improve, but weakens the model's first-attempt SQL generation ability. Consequently, when evaluated under the standard single-turn protocol without ODT feedback, the no-decay model performs substantially worse. More detailed analysis is provided in Appendix E.2.

Impact of Maximum Interaction Turns. We study the maximum interaction budget T, which controls the number of refinement attempts in each rollout. As shown in Figure 3b, performance improves as T increases up to 4, suggesting that additional turns help correct SQL errors. However, larger budgets lead to performance degradation, indicating that more refinement is not always beneficial. Therefore, we set $T = 4$ in all main experiments. Detailed analysis is provided in Appendix E.3.

Data-Matched Comparisons. To control for differences in training data, we reproduce SkyRL-SQL and SQL-Trail using the same training mix-

ture as Progress-SQL. For each baseline, we follow the prompt templates, interaction procedures, reward formulations, and training methodology described in the corresponding paper. As shown in Tables 1 and 2, Progress-SQL consistently outperforms both reproduced baselines in this data-matched setting. These results suggest that the gains of Progress-SQL cannot be attributed solely to differences in the amount or composition of training data. Complete results across decoding strategies and benchmarks are provided in Appendix E.6.

6 Conclusion

In this work, we proposed Progress-SQL, a multi-turn reinforcement learning framework with progressive rewards for Text-to-SQL. Progress-SQL leverages ODT-based diagnostic feedback to guide iterative SQL refinement and optimizes a progressive reward that jointly encourages alignment improvement, early correctness, and recovery from invalid SQL. Experiments on multiple Text-to-SQL benchmarks show consistent gains across primary and robustness evaluations, demonstrating the effectiveness of our framework for improving Text-to-SQL reasoning and refinement.

Limitations

Despite the effectiveness of our framework, two limitations remain. On the one hand, our ODT-based diagnostic feedback relies on the availability of gold SQL during training. Therefore, ODT is mainly designed as a training-time reinforcement learning signal rather than a test-time feedback module, since gold SQL is unavailable during standard inference. Nevertheless, our experiments show that such oracle-guided training feedback can be internalized by the policy model and improve its single-turn inference ability without using ODT at test time. On the other hand, the ODT scorer depends on SQL parsing and structural abstraction. Although lexical alignment is introduced as a fallback signal for ill-formed SQL predictions, severely malformed queries or dialect-specific SQL constructs may still reduce the precision of structural diagnostics. Extending the framework to more robust SQL parsers, diverse SQL dialects, and real-world interactive database settings remains an important direction for future work.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. This work is supported by the National Key Research & Develop Plan (Project No.2023YFF0725100) and Natural Science Foundation of China (Project No. U23A20298).

References

Alnur Ali, Ashutosh Baheti, Jonathan Chang, Ta-Chung Chi, Brandon Cui, Andrew Drozdov, Jonathan Frankle, Abhay Gupta, Pallavi Koppol, Sean Kulinski, et al. 2025. A state-of-the-art sql reasoning model using rlvr. *arXiv preprint arXiv:2509.21459*.

Mariia Berdnyk and Marine Collery. 2025. Llm-based sql generation with reinforcement learning. In *The First Workshop on Neural Reasoning and Mathematical Discovery at AAAI'2025. Workshop Paper*.

Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. Structure-grounded pretraining for text-to-sql. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 1337–1350. Association for Computational Linguistics.

Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. 2021a. Towards robustness of text-to-SQL models against synonym substitution. pages 2505–2515, Online. Association for Computational Linguistics.

Yujian Gan, Xinyun Chen, and Matthew Purver. 2021b. Exploring underexplored limitations of cross-domain text-to-sql generalization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8926–8931. Association for Computational Linguistics.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.*, 17(5):1132–1145.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, and 175 others. 2025. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638.

Heng Hao, Wenjun Hu, Oxana Verkholyak, Davoud Ataee Tarzanagh, Baruch Gutow, Sima Didari, Masoud Faraki, Hankyu Moon, and Seungjai Min. 2025. Paverl-sql: Text-to-sql via partial-match rewards and verbal reinforcement learning. *arXiv preprint arXiv:2509.07159*.

Harper Hua, Zhen Han, Zhengyuan Shen, Meng-Chieh Lee, Sheng Guan, Qi Zhu, Sullam Jeoung, Yueyan Chen, Yunfei Bai, Shuai Wang, Vassilis N. Ioannidis, and Huzefa Rangwala. 2026. SQL-trail: Multi-turn reinforcement learning with interleaved feedback for text-to-SQL. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 36224–36246, San Diego, California, United States. Association for Computational Linguistics.

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

Fei Li and Hosagrahar V Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, 8(1):73–84.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. 2025. Omnisql: Synthesizing high-quality text-to-sql data at scale. *Proc. VLDB Endow.*, 18(11):4695–4709.

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

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM already serve as A database interface? A big bench for large-scale database grounded text-to-sqls. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Shu Liu, Alan Zhu, Sumanth Hegde, Shiyi Cao, Shuo Yuan, Samion Suwito, Tyler Griggs, Matei Zaharia, Joseph E Gonzalez, and Ion Stoica. Skyrl-sql: Multi-turn sql data agents via rl. In *First Workshop on Multi-Turn Interactions in Large Language Models*.

Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2026. Xiyan-sql: A novel multi-generator framework for text-to-sql. *IEEE Trans. Knowl. Data Eng.*, 38(4):2474–2487.

Peixian Ma, Xialie Zhuang, Chengjin Xu, Xuhui Jiang, Ran Chen, and Jian Guo. 2026. Sql-r1: Training natural language to sql reasoning model by reinforcement learning. *Advances in Neural Information Processing Systems*, 38:174505–174537.

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

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th international conference on Intelligent user interfaces*, pages 149–157.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Ö. Arik. 2025a. CHASE-SQL: multi-path reasoning and preference optimized candidate selection in text-to-sql. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wan, Hailong Li, Azalia Mirhoseini, Amin Saberi, Sercan Arik, et al. 2025b. Reasoning-sql: Reinforcement learning with sql tailored partial rewards for reasoning-enhanced text-to-sql. *arXiv preprint arXiv:2503.23157*.

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

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

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297.

Wenbiao Tao, Hanlun Zhu, Keren Tan, Jiani Wang, Yuanyuan Liang, Huihui Jiang, Pengcheng Yuan, and Yunshi Lan. 2024. Finqa: A training-free dynamic knowledge graph question answering system in finance with llm-based revision. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 418–423. Springer.

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.

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. In *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, pages 12917–12943. Association for Computational Linguistics.

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

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. 2018a. Typesql: Knowledge-based type-aware neural text-to-sql generation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 588–594.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018b. 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, Brussels, Belgium, October 31 - November 4, 2018*, pages 3911–3921. Association for Computational Linguistics.

John M Zelle and Raymond J Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pages 1050–1055.

Yuxin Zhang, Meihao Fan, Ju Fan, Mingyang Yi, Yuyu Luo, Guoliang Li, Bin Wu, and Wenchao Zhou. 2026. Reward-sql: Boosting text-to-sql via stepwise execution-aware reasoning and process-supervised rewards. *Proceedings of the ACM on Management of Data*, 4(3 (SIGMOD):1–27.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. 2025. Group sequence policy optimization. *Preprint*, arXiv:2507.18071.

A Use of Large Language Models

The research presented in this paper, including the core ideas, experimental design, and quantitative results, is the original work of the authors. A large language model was used as a writing assistant for tasks such as polishing prose, improving clarity, and correcting grammatical errors in the manuscript. All final content was reviewed and edited by the authors to ensure it accurately reflects our research and contributions.

B Implementation Details

Training Data Construction The RL training corpus is constructed from three sources: the **BIRD** training set (Li et al., 2023), the **Spider** training set (Yu et al., 2018b), and **SynSQL-2.5M** (Li et al., 2025), a large-scale synthetic dataset covering diverse SQL complexity patterns. We randomly sample a total of 20,000 training instances from the union of these sources for reinforcement learning.

Training Setup and Infrastructure. We adopt Qwen2.5-Coder-7B/14B-Instruct as our base model for all reinforcement learning experiments. The training pipeline is developed and executed utilizing the `verl` framework across 8 NVIDIA H100 (80GB) GPUs. We employ Group Relative Policy Optimization (GRPO) as our core algorithm, which inherently bypasses the need for a separate value network (critic), thereby substantially reducing memory overhead and improving training efficiency. During the rollout phase, we sample $G = 8$ outputs per input query with a temperature of $\tau = 1.0$ and a maximum sequence length of 8192 tokens. The maximum number of interaction turns with the database engine is set to $T = 4$. The comprehensive hyperparameters for both the multi-turn rollout and policy optimization phases are summarized in Table 4.

Evaluation Setup. We follow the evaluation scripts and protocol released by OmniSQL (Li et al., 2025)[1] to ensure comparability with recent Text-to-SQL baselines. At evaluation time, ODT-based diagnostic feedback is not used, and all models are evaluated under standard single-turn inference.

Reward Hyperparameters. The scalar weights for our multi-turn reward formulation (defined in Section 4.2) are systematically detailed in Table 4.

[1] https://github.com/RUCKBReasoning/OmniSQL

Hyperparameter	Value
RL Training	
Optimizer	AdamW
Learning Rate	1×10^{-6}
Train Batch Size	64
Sampled Outputs G	8
Sampling Temperature τ	1.0
Max Response Length	8192
Max Prompt Length	6000
Clip Ratio ε	0.2
Max Interaction Turns T	4
KL Penalty	None
Reward Weights	
Formatting Reward ω_{fmt}	0.5
Base Accuracy Reward ω_{acc}	2.0
Alignment Improvement $\omega^{+}_{\text{align}}$	1.0
Alignment Penalty $\omega^{-}_{\text{align}}$	-0.25
Executable Keeping ω_{keep}	0.5
Execution Recovery ω_{rec}	0.25
Execution Deterioration ω_{det}	-0.25
Per-turn Decay Factor γ	0.5

Table 4: Hyperparameters for RL training and progressive reward formulation.

C Multi-turn Rollout with ODT Feedback

Algorithm 1 summarizes the training-time rollout procedure. At each turn, the policy generates a SQL prediction and receives ODT-based diagnostic feedback if the prediction is not yet correct. Here, T denotes the maximum turn budget, k^* denotes the first successful turn if execution correctness is reached, and K denotes the actual stopping turn. If the rollout succeeds early, then $K = k^*$; otherwise, $K = T$.

D ODT Construction and Scoring

Implementation Overview. We implement the Oracle-guided Diagnostic Tree (ODT) module as a deterministic structural feedback component. In our implementation, SQL queries are parsed with JSQLParser 4.6 and converted into recursive structural profile trees. Given a predicted SQL y and the gold SQL y^*, ODT extracts clause-level structural profiles, computes a structural similarity score, and generates diagnostic tags for next-turn feedback. The diagnostic feedback is used only during multi-turn RL training. At evaluation time, the model performs standard single-turn inference without ODT feedback.

SQL Parsing and Failure Handling. For each generated SQL, ODT first parses the query into an Abstract Syntax Tree (AST). Before parsing, we apply lightweight normalization, including removing redundant quotation marks, standardizing

Algorithm 1 Multi-turn SQL Refinement with ODT Feedback

Require: Question x, database D, gold SQL y^*, policy model π_θ, maximum turns T
Ensure: Trajectory $Y = \{y^{(1)}, \ldots, y^{(K)}\}$ and final SQL $y^{(K)}$, where $K \leq T$

```
Initialize prompt p^(1) ← x, trajectory Y ← ∅
Initialize first successful turn k* ← ∅ and stopping turn K ← T
for t = 1 to T do
    Generate SQL prediction y^(t) ~ π_θ(· | p^(t)) and append it to Y
    Execute y^(t) on database D to obtain execution status E(y^(t))
    if y^(t) matches the execution result of gold SQL y* then
        Set k* ← t and K ← t; break
    end if
    if t < T then
        Compare y^(t) with y* using the ODT diagnostic module
        Obtain clause-level diagnostic tags and verbalize them into feedback f^(t)
        Update prompt p^(t+1) ← Append(p^(t), y^(t), f^(t))
    end if
end for
Compute alignment improvement Δ = F(y^(K), y*) − F(y^(1), y*)
Compute total reward using alignment, decayed accuracy, execution status, and formatting terms
return Trajectory Y and final SQL y^(K)
```

whitespace, lower-casing identifiers, and stripping trailing semicolons. If the predicted SQL cannot be parsed, ODT does not produce fine-grained structural tags. Instead, it returns a generic feedback message asking the model to first fix syntax and identifier errors. In this case, the structural similarity is set to zero, and the reward relies on lexical alignment and execution-status signals.

ODT Construction. Each parsed SQL is converted into a recursive ODT profile tree. Each node corresponds to a structural unit, including `ROOT`, `SELECT`, `CTE`, `SET_OP`, or `SUBQUERY`. A `SELECT` node contains a local structural profile. A `CTE` node represents a common table expression, a `SET_OP` node represents set operations such as `UNION`, `INTERSECT`, or `EXCEPT`, and a `SUBQUERY` node represents a nested query. Subqueries appearing in `FROM`, `WHERE`, `HAVING`, or SELECT expressions are recursively detected and attached as child nodes. This representation allows ODT to compare both flat clause-level structures and nested SQL compositions.

Local Structural Profile. For each `SELECT` node, ODT extracts a local structural profile containing selected columns, involved tables, join signatures, filtering predicates, grouping columns, having predicates, ordering signatures, DISTINCT usage, aggregation usage, and structural counts. Concretely, the profile contains:

$$p = (\mathbf{c}, \mathbf{b}, \mathcal{T}, \mathcal{S}, \mathcal{W}, \mathcal{J}, \mathcal{G}, \mathcal{H}, \mathcal{O}),$$

where $\mathbf{c}$ denotes count features such as JOIN count and SELECT-item count, and $\mathbf{b}$ denotes Boolean indicators for WHERE, GROUP BY, HAVING, ORDER BY, LIMIT, aggregation, and DISTINCT. The set $\mathcal{T}$ contains normalized table names, $\mathcal{S}$ contains SELECT projection signatures, $\mathcal{W}$ contains normalized WHERE predicates, $\mathcal{J}$ contains JOIN-condition signatures, $\mathcal{G}$ contains GROUP BY column signatures, $\mathcal{H}$ contains HAVING predicates, and $\mathcal{O}$ contains ORDER BY signatures.

Feature Normalization. To reduce sensitivity to surface-level SQL variation, ODT normalizes structural features before comparison. Table names and column references are lower-cased, and quotation marks are removed. String and numeric literals are replaced with placeholders. Common predicate operators, including `=`, `<>`, `!=`, `>=`, `<=`, `>`, `<`, `LIKE`, `IN`, `BETWEEN`, `IS NULL`, and `IS NOT NULL`, are mapped into canonical predicate signatures. For equality predicates where both sides are identifiers, the operands are sorted lexicographically, so `a=b` and `b=a` are treated as equivalent. Conjunctive predicates are split by `AND` and stored as sets of normalized atomic predicates, making the order of independent conditions irrelevant. Aggregation aliases are ignored during structural comparison; for example, `AVG(price) AS avg_price` and `AVG(price) AS average_price` are treated as equivalent if the aggregation function and argument match.

Structural Scoring. Given the predicted ODT and the gold ODT, we recursively compute node-level similarity. For each `SELECT` node, the local similarity is computed as a weighted average over feature-level similarities, with the feature weights listed in Table 5:

$$s_{\text{local}}(p, g) = \frac{\sum_{f \in \mathcal{F}} w_f \cdot \text{sim}_f(p_f, g_f)}{\sum_{f \in \mathcal{F}} w_f}.$$

For set-valued features, we use Jaccard similarity:

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

If both sets are empty, the similarity is set to 1.0; if only one side is empty, it is set to 0.0. For count-valued features, including JOIN count and SELECT-item count, we use:

$$\text{NumSim}(a, b) = 1.0 - \min\left(1.0, \frac{|a - b|}{\max(1, b)}\right).$$

For Boolean features such as DISTINCT usage, the similarity is 1.0 if the two sides match and 0.0 otherwise.

Feature	Weight
JOIN conditions	0.22
WHERE predicates	0.20
SELECT projections	0.16
FROM/JOIN tables	0.12
GROUP BY columns	0.10
Structural counts	0.10
ORDER BY clauses	0.06
DISTINCT usage	0.04

Table 5: Feature weights used in ODT structural scoring.

Recursive Child Matching. The final score of an ODT node combines local similarity and child-subtree similarity:

$$s_{\text{node}} = \alpha \cdot s_{\text{local}} + (1 - \alpha) \cdot s_{\text{child}}.$$

We set $\alpha = 0.70$ for SELECT nodes. For non-SELECT nodes, which do not contain local profiles, the score is determined by child matching. To compute s_{child}, ODT matches predicted and gold children with compatible node types. For CTE nodes, the CTE names are matched case-insensitively. For each gold child, the scorer selects the highest-scoring unmatched predicted child. The child score is normalized by the maximum number of children on the two sides:

$$s_{\text{child}} = \frac{\sum_{(u,v)\in\mathcal{M}} s_{\text{node}}(u, v)}{\max(|\mathcal{C}_{\text{pred}}|, |\mathcal{C}_{\text{gold}}|)},$$

where $\mathcal{M}$ is the set of matched child pairs. Unmatched gold children indicate missing nested structures, while unmatched predicted children indicate redundant subqueries or CTEs. The final structural similarity is the root score:

$$\mathcal{F}_{\text{struct}}(y, y^*) = s_{\text{root}}.$$

Diagnostic Tag Generation. In addition to the continuous structural similarity score, ODT produces discrete diagnostic tags for feedback construction. For each diagnostic dimension, we compare the corresponding feature similarity against a predefined threshold, as listed in Table 6. If the similarity falls below the threshold, the corresponding error tag is emitted. Table 7 summarizes the diagnostic tags and their meanings. These tags are then verbalized into concise natural-language feedback and appended to the next-turn prompt. The raw structural similarity score and feature-level similarities are not exposed to the model; they are used only for reward computation.

Diagnostic Dimension	Threshold
JOIN conditions	0.60
WHERE predicates	0.60
SELECT projections	0.60
GROUP BY columns	0.65
ORDER BY clauses	0.65
FROM/JOIN tables	0.70

Table 6: Mismatch thresholds for generating ODT diagnostic tags.

Usage During RL Rollouts. For each unsuccessful non-final rollout turn, ODT compares the current prediction $y^{(t)}$ with the gold SQL y^* and returns verbalized structural feedback $f^{(t)}$. The next-turn input is constructed as:

$$x^{(t+1)} = (q, S, y^{(1)}, f^{(1)}, \ldots, y^{(t)}, f^{(t)}).$$

ODT is a fixed, non-differentiable component of the training environment. It does not participate in back-propagation, and gradients are computed only through the policy model over generated tokens.

E Additional Analysis

E.1 Training Dynamics

To better understand the impact of our progressive reward formulation on optimization dynamics, we analyze the training curves of reward and response length. Figure 4 compares our full multi-turn progressive reward with a single-turn one-shot reward baseline. The single-turn baseline generates one SQL prediction per rollout and assigns a score of 2.0 if the generated SQL matches the gold execution result, 0.5 if the SQL is executable, and 0.5 if the output follows the required format. Unlike our method, this baseline does not use multi-turn refinement, ODT-based structural feedback, or trajectory-level improvement.

Tag	Trigger	Meaning
JOIN_MISSING	JOIN count differs from the reference.	Required JOIN structures are missing.
JOIN_KEY_MISMATCH	JOIN-condition similarity is below threshold.	JOIN keys or ON conditions are inconsistent.
FROM_OR_JOIN_TABLE_MISMATCH	Table-set similarity is below threshold.	Missing, redundant, or incorrect tables are used.
WHERE_ERROR	WHERE-predicate similarity is below threshold.	Filtering predicates differ from the reference.
GROUP_BY_MISSING	Gold SQL contains GROUP BY columns but the predicted SQL has no GROUP BY clause.	Required grouping structure is missing.
GROUP_BY_ERROR	GROUP-BY similarity is below threshold.	Grouping columns or granularity are inconsistent.
ORDER_BY_MISMATCH	ORDER-BY similarity is below threshold.	Ordering column, direction, or limit structure is inconsistent.
SELECT_ERROR	SELECT-projection similarity is below threshold.	Selected columns or aggregation expressions differ.
AGGREGATE_ERROR	Aggregation usage differs.	Aggregation functions may be missing or misused.
DISTINCT_MISMATCH	DISTINCT usage differs.	Duplicate-removal behavior is inconsistent.
SUBQUERY_MISSING	A gold subquery child has no compatible match.	An expected nested query branch is missing.
EXTRA_SUBQUERY_OR_CTE	Predicted tree contains unmatched extra children.	Redundant nested or CTE branches are introduced.

Table 7: Diagnostic tags emitted by the ODT module.

Reward Convergence. As shown in the left panel of Figure 4, both the single-turn one-shot baseline and our progressive reward improve steadily during training and reach comparable reward levels. This indicates that our multi-turn objective does not make optimization harder despite introducing ODT-based feedback and trajectory-level reward components. Compared with the single-turn baseline, our reward provides additional structural and lexical alignment signals that guide the policy toward refinement-oriented behavior rather than only endpoint correctness.

Different Response-Length Dynamics. The right panel of Figure 4 shows that the two methods lead to clearly different response-length behaviors. The single-turn one-shot baseline quickly converges to shorter responses, since it only rewards final correctness, executability, and format compliance. In contrast, our method maintains longer responses during training, which is consistent with the multi-turn refinement setting where the model needs to reason about previous SQL attempts and diagnostic feedback. Although longer responses are not directly rewarded, this suggests that the policy learns to utilize the additional context and feedback rather than collapsing to a minimal endpoint-oriented generation strategy.

Overall, these trends show that the proposed progressive reward achieves stable optimization while inducing training dynamics better aligned with multi-turn SQL correction.

Figure 4: Training dynamics of reward and response length. The single-turn one-shot reward baseline assigns rewards based on execution correctness, executability, and format compliance, while our method further incorporates multi-turn refinement, ODT-based structural feedback, and trajectory-level progress. Both methods show stable reward improvement, but our progressive reward leads to different response-length dynamics that are more consistent with multi-turn SQL refinement.

E.2 Effect of Per-turn Decay

Figure 3a analyzes the effect of removing the per-turn decay term by setting $\gamma = 1.0$. Although the accumulated reward continues to increase, the average number of interaction turns first decreases and then rebounds to a high level. This suggests that removing per-turn decay makes multi-turn RL optimization unstable: the policy is no longer encouraged to produce correct SQL early, and instead tends to rely on longer correction trajectories with more ODT feedback.

This instability also explains the large perfor-

mance drop in Table 3. Without per-turn decay, late corrections receive the same accuracy reward as early corrections, which weakens the pressure to improve the first-attempt SQL. As a result, the learned policy becomes poorly aligned with the standard single-turn inference setting used at evaluation time. The per-turn decay term therefore acts as an important regularizer for multi-turn RL training, encouraging earlier successful executions and stabilizing the transfer from multi-turn training to single-turn inference.

E.3 Maximum Interaction Turns

Figure 3b shows the effect of varying the maximum interaction budget T. Increasing T from 1 to 4 consistently improves execution accuracy on both BIRD Dev and Spider Dev, suggesting that additional turns provide useful opportunities for structural correction. However, further increasing T causes performance degradation. We attribute this to context accumulation and over-correction: longer trajectories introduce more historical SQL attempts and feedback messages, which may distract the model or make it deviate from the original question. Based on these observations, we set $T = 4$ in all main experiments.

E.4 Single-turn Inference Behavior after Multi-turn RL

During training, our framework exposes the policy to multi-turn ODT-based diagnostic feedback, while during evaluation we deliberately use standard single-turn inference without ODT feedback or iterative refinement. This setting ensures fair comparison with existing Text-to-SQL baselines, but also raises an important question: how can a model trained in a multi-turn debugging environment benefit at single-turn test time?

We hypothesize that multi-turn RL does not merely teach the model to react to external feedback, but also helps the policy internalize structural debugging patterns. In other words, after optimization with progressive rewards, the model may learn to anticipate common SQL construction errors before producing the final query. Such internalization should be reflected in fewer structural errors in the first generated SQL, even when no ODT feedback is provided at inference time.

To verify this hypothesis, we apply the ODT diagnostic module only as an offline evaluator for single-turn greedy predictions on Spider Dev. Specifically, after each model generates its SQL, we compare the predicted SQL with the gold SQL using ODT and map fine-grained diagnostic tags into four coarse structural categories: SELECT, JOIN, WHERE, and GROUP errors. Importantly, this analysis is performed after generation; the diagnostic tags are never exposed to the model during evaluation.

As shown in Table 8, RL-trained models consistently reduce the average structural error rate under single-turn inference on Spider Dev. For the 7B model, the average error rate decreases from 34.6% to 29.0%, with especially clear reductions in JOIN and GROUP errors. For the 14B model, the average error rate decreases from 37.1% to 26.1%, mainly due to large reductions in SELECT, WHERE, and GROUP errors. Although the JOIN error rate of the 14B model increases, the overall structural error rate still drops substantially. These results suggest that the benefit of multi-turn ODT-based training is not limited to interactive correction: the learned policy also improves first-attempt SQL construction without relying on test-time oracle feedback.

E.5 Decoding Strategy Analysis by Difficulty

To provide a fine-grained view of the inference behavior of our RL-trained models, we report their performance across SQL query difficulty levels under two decoding strategies. Specifically, we evaluate Progress-SQL-7B and Progress-SQL-14B on BIRD Dev, Spider Dev, Spider Test, and three robustness variants: Spider-DK, Spider-Realistic, and Spider-Syn. For each benchmark, we include both Greedy Decoding and Majority Voting (Vote@8).

Overall Observation. The results show that decoding strategy can affect performance differently across benchmarks and difficulty subsets. Vote@8 often changes the aggregate score compared with greedy decoding, but the effect is not uniform across all settings. In some hard or extra-hard subsets, majority voting may select a suboptimal candidate when generated SQLs differ in subtle structural details. Therefore, this analysis is intended to provide a more detailed view of decoding behavior rather than to claim that one decoding strategy consistently dominates the other.

BIRD Dev. Table 9 reports the BIRD Dev results. We report both greedy decoding and Vote@8 results for Progress-SQL-7B and Progress-SQL-14B, showing how performance varies across simple, moderate, and challenging examples.

Spider Dev and Spider Test. Table 10 reports the results on Spider Dev and Spider Test. We in-

Model	SELECT Err.	JOIN Err.	WHERE Err.	GROUP Err.	Avg. Err.
Qwen2.5-Coder-7B	47.3	38.1	33.6	19.3	34.6
Progress-SQL-7B	42.0 (↓ 5.3)	29.9 (↓ 8.2)	30.7 (↓ 2.9)	13.4 (↓ 5.9)	29.0 (↓ 5.6)
Qwen2.5-Coder-14B	68.6	17.5	39.2	23.3	37.1
Progress-SQL-14B	37.8 (↓ 30.8)	24.1 (↑ 6.6)	31.0 (↓ 8.2)	11.5 (↓ 11.8)	26.1 (↓ 11.0)

Table 8: ODT-based structural mismatch rates for single-turn greedy inference on Spider Dev. The ODT diagnostic module is used only as an offline evaluator and is not exposed to the model during inference. These rates measure clause-level structural mismatches rather than execution failures, and thus are not directly complementary to EX. Arrows indicate absolute changes compared with the corresponding same-scale base model; lower values are better. Avg. Err. averages SELECT, JOIN, WHERE, and GROUP mismatch rates.

Model	Decoding	BIRD Dev (EX%)			
		Simple	Moderate	Chall.	All
Progress-SQL-7B	Greedy	68.6	54.5	49.7	62.6
	Vote@8	71.2	55.2	49.0	64.3
Progress-SQL-14B	Greedy	72.2	58.0	53.8	66.2
	Vote@8	73.8	58.4	55.9	67.5

Table 9: Execution accuracy breakdown by difficulty on BIRD Dev for our RL-trained models under greedy decoding and Vote@8.

clude both greedy decoding and Vote@8 results to show how decoding choices affect different difficulty subsets. The aggregate scores are generally close between the two decoding strategies, while the difficulty-level breakdown reveals non-uniform changes across easy, medium, hard, and extra-hard examples. In particular, Vote@8 does not consistently improve hard or extra-hard subsets, suggesting that majority voting may select a suboptimal candidate when generated SQLs differ in subtle structural details.

Robustness Variants. Tables 11 and 12 report difficulty-level results on Spider-DK, Spider-Realistic, and Spider-Syn. These results further show that the effect of decoding strategy varies across robustness settings and difficulty levels. Therefore, we present both greedy and Vote@8 results to provide a more complete view of inference behavior under different perturbation scenarios.

E.6 Detailed Results of Data-Matched Baselines

To provide a complete data-matched comparison, we reproduce SkyRL-SQL and SQL-Trail using the same training mixture as Progress-SQL. All methods are initialized from Qwen2.5-Coder-7B-Instruct. For each reproduced baseline, we follow the prompt templates, interaction procedures, reward formulations, and training methodology described in the corresponding paper, while replacing its original training data with our training mixture. Each method retains its native interaction and inference protocol.

Table 13 reports the complete results under both Greedy and Vote@8 decoding across the primary and robustness benchmarks. Progress-SQL consistently outperforms the reproduced baselines in this data-matched setting, indicating that its improvements cannot be explained solely by differences in the amount or composition of training data. Because the methods retain their respective interaction and inference protocols, these results should be interpreted as a data-matched comparison of complete systems rather than an isolated comparison of reward functions.

E.7 Generalization Across RL Algorithms

To verify that our proposed multi-turn progressive reward $\mathcal{R}(Y)$ is algorithm-agnostic, we evaluate it under two distinct RL training paradigms: **GRPO** (Group Relative Policy Optimization) (Shao et al., 2024), our primary algorithm, and **GSPO** (Group Sequence Policy Optimization) (Zheng et al., 2025). For a controlled comparison, both algorithms are initialized from the same Qwen2.5-Coder-7B-Instruct base model without any task-specific supervised fine-tuning, and are optimized using identical multi-turn reward signals.

For a given input query x, both algorithms sample a group of G responses $\{y_i\}_{i=1}^{G}$ from the old policy $\pi_{\theta_{\text{old}}}(\cdot|x)$, and compute the group-based advantage estimation $\hat{A}_i$ using our multi-turn reward $\mathcal{R}(y_i)$:

$$\hat{A}_i = \frac{\mathcal{R}(y_i) - \text{mean}(\{\mathcal{R}(y_i)\}_{i=1}^{G})}{\text{std}(\{\mathcal{R}(y_i)\}_{i=1}^{G})}$$

GRPO Formulation. GRPO optimizes the policy at the token level. It computes the advantage for each token $y_{i,t}$ as $\hat{A}_{i,t} = \hat{A}_i$, and ap-

Model	Decoding	Spider Dev										Spider Test				
		EX(%)					TS(%)					EX(%)				
		Easy	Med.	Hard	Extra	All	Easy	Med.	Hard	Extra	All	Easy	Med.	Hard	Extra	All
Progress-SQL-7B	Greedy	96.0	90.4	82.2	66.9	86.6	95.2	85.9	71.8	50.0	80.0	93.8	90.2	82.9	74.2	86.8
	Vote@8	96.8	90.8	81.6	68.7	87.1	95.6	87.2	71.8	51.2	80.9	93.0	91.0	84.2	78.2	87.8
Progress-SQL-14B	Greedy	95.6	90.6	82.2	68.7	86.8	93.5	86.8	74.7	46.4	79.9	94.7	90.5	82.9	77.3	87.6
	Vote@8	96.0	92.4	81.0	67.5	87.3	94.8	86.1	74.7	49.4	80.4	94.9	91.2	83.6	77.3	88.1

Table 10: Execution accuracy (EX%) and test-suite accuracy (TS%) breakdown by difficulty on Spider Dev and Spider Test for our RL-trained models under greedy decoding and Vote@8. Spider Test reports EX only.

Model	Decoding	Spider-DK (EX%)					Spider-Realistic (EX%)					Spider-Syn (EX%)				
		Easy	Med.	Hard	Extra	All	Easy	Med.	Hard	Extra	All	Easy	Med.	Hard	Extra	All
Progress-SQL-7B	Greedy	82.7	77.6	67.6	60.0	73.8	89.9	91.6	80.8	61.9	83.5	87.1	78.6	71.2	55.0	75.5
	Vote@8	82.7	79.7	73.0	61.9	75.9	91.7	91.1	83.8	60.8	84.1	89.1	79.5	71.8	55.6	76.6
Progress-SQL-14B	Greedy	90.0	80.5	58.1	67.6	76.8	96.3	91.1	77.8	64.9	84.4	87.9	82.0	70.6	53.3	76.8
	Vote@8	87.3	82.5	62.2	61.0	76.4	96.3	91.1	75.8	63.9	84.1	88.3	81.6	70.6	55.6	77.1

Table 11: Execution accuracy (EX%) breakdown by difficulty on Spider-DK, Spider-Realistic, and Spider-Syn for our RL-trained models under greedy decoding and Vote@8.

Model	Dec.	Spider-Realistic (TS%)					Spider-Syn (TS%)				
		Easy	Med.	Hard	Extra	All	Easy	Med.	Hard	Extra	All
Progress-SQL-7B	Greedy	90.8	84.7	70.7	41.2	75.0	82.3	74.1	62.7	33.7	67.5
	Vote@8	92.7	85.7	73.7	41.2	76.4	84.3	75.0	62.7	36.1	68.8
Progress-SQL-14B	Greedy	94.5	85.2	65.7	46.4	76.0	84.3	75.0	63.8	34.3	68.7
	Vote@8	94.5	86.7	67.7	44.3	76.6	83.5	75.5	64.4	36.1	69.1

Table 12: Test-suite accuracy (TS%) breakdown by difficulty on Spider-Realistic and Spider-Syn for our RL-trained models under greedy decoding and Vote@8. Spider-DK is omitted because test-suite accuracy is unavailable.

plies clipping to the token-level importance ratio $w_{i,t}(\theta) = \frac{\pi_\theta(y_{i,t}|x, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t}|x, y_{i,<t})}$:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{x\sim\mathcal{D},\{y_i\}_{i=1}^G\sim\pi_{\theta_{\text{old}}}(\cdot|x)}\Bigg[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|y_i|}\sum_{t=1}^{|y_i|}\min\Big(w_{i,t}(\theta)\hat{A}_{i,t}, \text{clip}\big(w_{i,t}(\theta), 1-\varepsilon, 1+\varepsilon\big)\hat{A}_{i,t}\Big)\Bigg]$$

GSPO Formulation. In contrast, GSPO applies clipping to entire responses instead of individual tokens to better match sequence-level rewards. To reduce variance and control numerical range, it defines a length-normalized sequence importance ratio $s_i(\theta)$:

$$s_i(\theta) = \left(\frac{\pi_\theta(y_i|x)}{\pi_{\theta_{\text{old}}}(y_i|x)}\right)^{\frac{1}{|y_i|}} = \exp\left(\frac{1}{|y_i|}\sum_{t=1}^{|y_i|}\log\frac{\pi_\theta(y_{i,t}|x, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t}|x, y_{i,<t})}\right)$$

The sequence-level optimization objective is then formulated as:

$$\mathcal{J}_{\text{GSPO}}(\theta) = \mathbb{E}_{x\sim\mathcal{D},\{y_i\}_{i=1}^G\sim\pi_{\theta_{\text{old}}}(\cdot|x)}\Bigg[\frac{1}{G}\sum_{i=1}^{G}\min\Big(s_i(\theta)\hat{A}_i, \text{clip}\big(s_i(\theta), 1-\varepsilon, 1+\varepsilon\big)\hat{A}_i\Big)\Bigg]$$

As shown in Table 14, both GRPO and GSPO achieve strong performance under the same multi-turn progressive reward formulation, suggesting that our reward design is not tied to a specific RL

Method	Decoding	BIRD	Spider Dev		Spider Test	Spider-DK	Spider-Realistic		Spider-Syn	
		EX	EX	TS	EX	EX	EX	TS	EX	TS
SkyRL-SQL†	Greedy	58.8	84.1	75.2	85.1	70.5	83.1	72.2	71.8	62.1
SkyRL-SQL†	Vote@8	60.2	84.5	74.9	85.7	70.7	83.5	73.0	72.1	62.2
SQL-Trail†	Greedy	55.7	84.7	77.6	83.9	73.5	82.6	73.1	72.7	64.4
SQL-Trail†	Vote@8	57.0	85.6	77.4	84.0	75.4	83.7	74.4	73.7	65.1
Progress-SQL	Greedy	**62.6**	**86.6**	**80.0**	**86.8**	**73.8**	**83.5**	**75.0**	**75.5**	**67.5**
Progress-SQL	Vote@8	**64.3**	**87.1**	**80.9**	**87.8**	**75.9**	**84.1**	**76.4**	**76.6**	**68.8**

Table 13: Complete results of the data-matched comparison. All methods are initialized from Qwen2.5-Coder-7B-Instruct and trained using the same training mixture. † denotes baselines reproduced using our training mixture while following the prompt templates, interaction procedures, reward formulations, and training methodology described in the corresponding papers. Each method retains its native interaction and inference protocol.

optimizer. GRPO performs slightly better on BIRD Dev and Spider Dev TS, while GSPO obtains a marginally higher Spider Dev EX under Vote@8. These results indicate that the proposed progressive reward can transfer across different policy optimization algorithms, with only minor variations in final performance.

Algorithm	**Decoding**	**BIRD Dev**	**Spider Dev**	
		EX	**EX**	**TS**
GSPO	Greedy	61.6	85.3	78.3
	Vote@8	63.8	87.5	79.9
GRPO	Greedy	62.6	86.6	80.0
	Vote@8	64.3	87.1	80.9

Table 14: Generalization of our multi-turn progressive reward $\mathcal{R}(Y)$ across RL algorithms. Both GRPO and GSPO are initialized from Qwen2.5-Coder-7B-Instruct and trained with identical reward signals.

E.8 Generalization to Qwen3-8B

To further validate the generalizability of our framework, we apply the same training pipeline to Qwen3-8B (Yang et al., 2025) without architecture-specific modification.

As shown in Table 15, our method consistently improves Qwen3-8B on both BIRD Dev and Spider Dev under greedy decoding and Vote@8. These results demonstrate that the proposed reward design is not tied to the Qwen2.5-Coder architecture. Instead, it generalizes effectively to Qwen3-8B, suggesting that our structural feedback and progressive reward formulation can benefit stronger base models as well. The consistent gains across both EX and TS suggest that our training objective

Model	**Decoding**	**BIRD Dev**	**Spider Dev**	
		EX(%)	**EX(%)**	**TS(%)**
Qwen3-8B	Greedy	59.5	82.2	70.1
	Vote@8	62.3	84.6	72.0
Progress-SQL-8B	Greedy	64.0(+4.5)	84.5(+2.3)	74.1(+4.0)
	Vote@8	65.8(+3.5)	85.9(+1.3)	74.9(+2.9)

Table 15: Generalization results on Qwen3-8B. Subscript values denote absolute gains over the corresponding Qwen3-8B baseline.

improves not only final execution correctness, but also the robustness of the generated SQL programs.

F Prompt Templates

The following templates are used across all training and rollout phases. Placeholders are denoted in monospace format: `{db_engine}` is filled with `SQLite`; `{db_details}` contains the DDL schema of the target database; `{question}` is the natural-language query.

System Prompt

Task Overview:
You are a data science expert. The user message contains the database schema and the natural language question. Your task is to understand the schema and generate a valid SQL query to answer the question.

Database Engine:
`{db_engine}`

Instructions:

- **Dialect Strictness:** Ensure the gener-

ated SQL strictly conforms to the specific syntax and functions of the provided {db_engine}.

- **Precision:** Only output the exact information asked in the question. Do not include extra columns in the SELECT clause.
- **Ambiguity Prevention:** Always use short table aliases (e.g., t1, t2) and explicitly qualify all column names with their respective table aliases, especially when JOINs are involved.
- **Robustness:** Consider edge cases such as handling NULL values appropriately and string matching case conventions where applicable.
- **Thinking Process:** Before generating the final SQL query, step-by-step analyze the tables needed, the JOIN conditions, and the filtering criteria. Enclose your reasoning strictly within <think> and </think> tags.

Output Format:
Please strictly follow this exact output structure. Do not output any other text or markdown outside of these tags:

```
<think>
Your step-by-step reasoning here.
</think>
<sql>
SELECT ...
</sql>
```

User Prompt — Round 1

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

Question:
{question}

User Prompt — Round 2+ (ODT Feedback Injection)

Feedback: your previous SQL has structural issues: {error_codes}, {error_descriptions} . Revise your previous answer using these issues. Revise your previous answer for SQLite. Put step-by-step reasoning inside <think>...</think>, then output exactly one corrected SQLite query inside <sql>...</sql>. Do not output any text outside those tags.

Note: When the ODT service fails to parse the predicted SQL, or when the gold SQL cannot be fully parsed and no cached structural profile is available, we inject a generic fallback message instead of clause-level diagnostic feedback.

G Case Study: Progressive Refinement with Structural Feedback

To qualitatively demonstrate the efficacy of our ODT-based dense feedback, we present a complete three-turn debugging trajectory. This case highlights how clause-level feedback helps the model correct aggregation granularity, projection targets, and redundant join structures during iterative SQL refinement.

Question:

> *Highly-rated suppliers refer to those who have received a rating of 4 or higher from customers. What are the names and average prices of contact lenses for highly-rated suppliers who have sold at least one lens?*

Gold SQL

```
SELECT s.supplier_name, AVG(cl.price) AS
    average_price
FROM contact_lenses cl
JOIN suppliers s ON s.supplier_id =
    cl.supplier_id
JOIN feedback f ON cl.lens_id = f.lens_id
WHERE f.rating >= 4
GROUP BY s.supplier_name
HAVING COUNT(cl.lens_id) > 0;
```

Debugging Trajectory

Round 1 [× EX-Wrong]

Analysis: The agent fundamentally misunderstands the required granularity. It attempts to group by lens_name instead of supplier_name, and over-complicates the query by unnecessar-

ily joining the sales and order_items tables to check for sales.

Model Output

Chain-of-Thought (Extracted):

To answer this question, we need to identify contact lenses from suppliers who have received a rating of 4 or higher... I will join contact_lenses with suppliers, feedback, sales, and order_items. Finally, I will group by lens name and calculate the average price. [snip]

Predicted SQL:

```
SELECT t1.lens_name, AVG(t1.price) AS
    average_price
FROM contact_lenses t1
JOIN suppliers t2 ON t1.supplier_id =
    t2.supplier_id
JOIN feedback t3 ON t1.lens_id = t3.lens_id
JOIN order_items t4 ON t1.lens_id =
    t4.lens_id
WHERE t3.rating >= 4
GROUP BY t1.lens_name
HAVING COUNT(DISTINCT t4.order_item_id) > 0;
```

ODT Feedback

```
Feedback: your previous SQL has structural
    issues: JOIN_KEY_MISMATCH,
FROM_OR_JOIN_TABLE_MISMATCH, SELECT_ERROR,
    GROUP_BY_ERROR. [...]
SELECT projection differs from reference.
    GROUP BY columns are not aligned
with reference, and redundant join
    structures are introduced.
```

Round 2 [× EX-Wrong]

Analysis: Guided by the dense feedback, the agent successfully corrects the projection (SELECT) and grouping (GROUP BY) targets to supplier_name. However, it stubbornly retains the redundant sales and order_items tables, leading to a severe table mismatch warning.

Model Output

Chain-of-Thought (Extracted):

Based on the feedback, there are several issues... The selected column and grouping key should be changed to the supplier name. I will revise the query accordingly, but I still keep the transaction-table joins to check whether a lens has been sold. [snip]

Predicted SQL:

```
SELECT s.supplier_name, AVG(cl.price) AS
    average_price
FROM contact_lenses cl
JOIN suppliers s ON cl.supplier_id =
    s.supplier_id
JOIN feedback f ON cl.lens_id = f.lens_id
JOIN order_items oi ON cl.lens_id =
    oi.lens_id
JOIN sales sa ON oi.order_item_id =
    sa.order_id
WHERE f.rating >= 4
GROUP BY s.supplier_name;
```

ODT Feedback

```
Feedback: your previous SQL has structural
    issues: JOIN_KEY_MISMATCH,
FROM_OR_JOIN_TABLE_MISMATCH. [...] FROM/JOIN
    tables differ from expected, and
the query introduces redundant
    transaction-table joins.
```

Round 3 [✓ EX-Match]

Analysis: Acknowledging the specific FROM_OR_JOIN_TABLE_MISMATCH error, the agent finally drops the hallucinated transaction tables. It simplifies the query to the core tables and recovers the correct aggregation granularity with GROUP BY s.supplier_name and HAVING COUNT(cl.lens_id) > 0, achieving an exact execution match.

Model Output

Chain-of-Thought (Extracted):

Based on the new feedback... FROM_OR_JOIN_TABLE_MISMATCH: The FROM and JOIN clauses in the previous query included tables that are not necessary. I will only join contact_lenses with suppliers and feedback. Then group by supplier name and calculate the average price. [snip]

Predicted SQL:

```
SELECT s.supplier_name, AVG(cl.price) AS
    average_price
FROM contact_lenses cl
JOIN suppliers s ON cl.supplier_id =
    s.supplier_id
JOIN feedback f ON cl.lens_id = f.lens_id
WHERE f.rating >= 4
GROUP BY s.supplier_name
HAVING COUNT(cl.lens_id) > 0;
```