

Beyond Query-Level Comparison: Fine-Grained Reinforcement Learning for Text-to-SQL with Automated Interpretable Critiques

Anonymous ACL submission

Abstract

Text-to-SQL, a pivotal natural language processing (NLP) task that converts textual queries into executable SQL, has seen substantial progress in recent years. However, existing evaluation and reward mechanisms used to train and assess the text-to-SQL models remain a critical bottleneck. Current approaches heavily rely on manually annotated gold SQL queries, which are costly to produce and impractical for large-scale evaluation. More importantly, most reinforcement learning (RL) methods in text-to-SQL leverage only the final binary execution outcome as the reward signal, a coarse-grained supervision that overlooks detailed structural and semantic errors from the perspective of rubrics. To address these challenges, we propose **RuCo-C**, a novel generative judge model for fine-grained, query-specific automatic evaluation using *interpretable critiques* without human intervention. Our framework first automatically generates query-specific evaluation rubrics for human-free annotation, linking them to interpretable critiques. Subsequently, it integrates densified reward feedback through a “progressive exploration” strategy during the RL training process, which dynamically adjusts the rewards to enhance the model’s performance. Comprehensive experiments demonstrate that RuCo-C outperforms existing methods in text-to-SQL evaluation, yielding significant performance gains.

1 Introduction

As a pivotal natural language processing (NLP) task, text-to-SQL turns textual queries into executable SQL, has achieved significant advancement over the past few years (Liu et al., 2025a; Zhang et al., 2024). Current text-to-SQL solutions focus on optimizing various components of the text-to-SQL workflows, including the pre-processing module (e.g., schema linking (Guo et al., 2019; Talaei et al., 2025)), the translation module (Wang et al., 2020; Pourreza et al., 2025a; Li et al., 2025) and

the post-processing (e.g., output consistency (Gao et al., 2024; Li et al., 2024)). More recently, various reinforcement learning (RL) methods have been proposed to enhance the reasoning and generalization capabilities of text-to-SQL. In terms of reward function design, some RL approaches primarily use execution accuracy (EX) as the feedback signal (e.g., SQL-R1 (Ma et al., 2025)), while other works combines multiple reward functions such as n-gram similarity, syntax correctness, and executed table cell results (e.g., Reasoning-SQL (Pourreza et al., 2025b), Think2SQL (Papicchio et al., 2025)).

Despite the aforementioned advancements, existing evaluation and reward mechanisms for training and assessing the text-to-SQL models remain a critical bottleneck. Specifically, current evaluation methods primarily utilize query-based pairwise comparisons between the predicted SQL query and the gold standard query (e.g., exact match (SONG et al., 2024)), execution matching (Zhong et al., 2020), and embedding matching (Zhan et al., 2025)). These approaches rely on manually annotated gold SQL queries, which are costly to produce and impractical for large-scale evaluation. Moreover, most RL approaches in text-to-SQL leverage only the final binary execution outcome as the reward signal (Pourreza et al., 2025b; Weng et al., 2025), a coarse-grained supervision that ignores detailed structural and semantic errors in the intermediate reasoning steps.

To address these challenges, we propose **RuCo-C** (**R**ubrics and **r**eward **C**onsistency **C**ritique model), a novel generative judge model that enables fine-grained, query-specific automatic evaluation using interpretable critiques without human intervention. First, we design *interpretable critiques* by a self-asking approach and presented in Question-Answer (QA) pairs (i.e., the *<think>* components in Figure 1): the question identifies whether the items that need inspection for a user query are correct, while the answer elaborates on

```

Critique Response
<think>
1. Did I correctly specify the column to filter the bond
type?
- Yes, the "WHERE bond_type = 'triple'" clause cor-
rectly filters for triple bond types.
2. Did I join the necessary tables to retrieve the re-
quired information?
- No, the query does not join the "molecule" table,
which is necessary to determine if the molecules
are carcinogenic or not. The "bond" table has a
"molecule_id" column can be used to join with the
"molecule" table.
...
</think>
<result> False </result>
<correctedSQL>
1 SELECT b.bond_id, m.label
2 FROM bond b
3 JOIN molecule m ON b.
4   molecule_id = m.molecule_id
5 WHERE b.bond_type = 'triple'
</correctedSQL>

```

Figure 1: An example of the critique response illustrates the designed output format, which will be used for reward score in the RL training of the critique model.

inspection conclusions based on specific evidence. This complete and detailed QA format fully embodies the evaluation rubrics. Then, we employ Supervised Fine-Tuning (SFT) to internalize rubric-based evaluation logic, aligning the model’s scoring behavior more closely with human expert judgments. Finally, in the RL training phase, we design fine-grained process rewards and a reward consistency mechanism. Specifically, the Outcome Reward Model (ORM) is calculated based on execution accuracy, whereas the Process Reward Model (PRM) is primarily quantified based on our designed interpretable critique responses. To account for task difficulty variations while preserving a consistent process reward, we dynamically tune the PRM’s weight coefficient according to difficulty levels and correction status. These components densify the evaluation scores, enrich the feedback signals, and ultimately increase the stability of the training, all while improving the accuracy and robustness of the model’s evaluation. Comprehensive experiments demonstrate that our RuCo-C outperforms existing methods in text-to-SQL evaluation, significantly improving overall performance.

In a nutshell, our contributions includes:

- **Evaluation Rubrics with Interpretable Critiques:** We automatically generate query-

specific rubrics for human-free annotation, linking them to interpretable critiques.

- **Enhanced RL Training:** We integrate a novel rubric-related generative task into the RL training process, using a “progressive exploration” strategy to dynamically adjust rewards and enhance the model’s inferential ability.
- **Novel Reward Design:** We construct the final reward function by combining ORM with PRM. In exploring PRM, we determine the combination weight of PRM by integrating the total number of rubric items and the specific responses guided by each rubric.

2 Related Work

2.1 Text-to-SQL Evaluation Methods

Existing evaluation methods for text-to-SQL mainly focus on query-based comparison evaluation (Renggli et al., 2025), which can be systematically classified into three main categories: semantic matching (SONG et al., 2024), execution matching (Zhong et al., 2020), and embedding matching (Zhan et al., 2025). The semantic matching parses both gold and predicted SQL queries into an intermediate format (e.g., abstract syntax tree, AST (Cao et al., 2023)) and tests their equivalence via schema-based algebraic transformations (SONG et al., 2024). The execution matching assesses the equivalence by running two SQL queries on the same database instance and comparing the results. For instance, Zhong et al. (2020) proposed Test-Suite Accuracy, which creates a small test suite of database via random generation to maximize the coverage of the gold query and calculates the denotation accuracy in this distilled set to estimate the semantic accuracy. The embedding-based matching approaches, such as Codebertscore (Zhou et al., 2023) and FuncEvalGMN (Zhan et al., 2025), assess the semantic equivalence of SQL statements by computing the similarity between their vector representations.

While existing query-level pairwise comparison methods have advanced the field of text-to-SQL assessment, they face several challenges: reliance on a single gold query, risk of spurious evaluations, and difficulty in gauging complex SQL semantics. By contrast, our designed generative judge model requires no gold SQL, which is better suited for online real-time scenarios. Furthermore, the generated critique information is interpretable and can

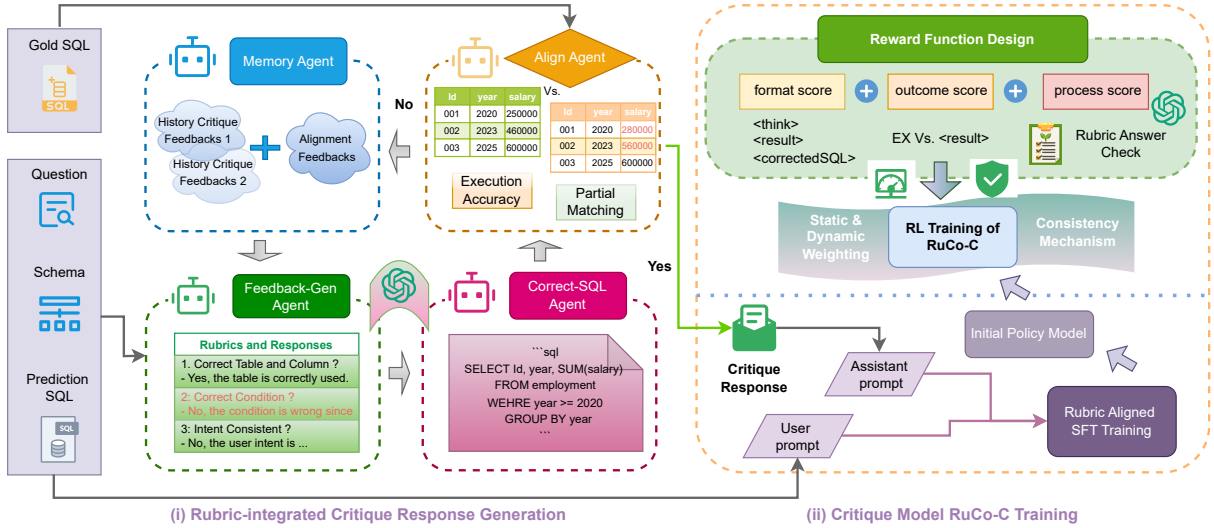


Figure 2: The working pipeline of our solution. It first automatically generates rubric-integrated critique response via a multi-agent framework. Then, it integrates densified reward feedbacks through a consistency mechanism combined with weighting strategy during RL training, dynamically adjusting rewards to enhance model performance.

be converted into more granular reward signals to guide the training of both the judge model itself and text-to-SQL models, thereby advancing innovation and precision in the field.

2.2 Reward Models for RL Training

Currently, the prevalent approach in most Reinforcement Learning with Verifiable Rewards (RLVR) (Lambert et al., 2025; Yue et al., 2025) models is to combine rule scores with final outcome correctness scores as the reward score. For example, SQL-R1 (Ma et al., 2025) considers syntactic correctness and result correctness through execution accuracy (EX) as the reward score in the reinforcement learning of the text-to-SQL task. Furthermore, Reasoning-SQL (Poureza et al., 2025b) adopts a suite of reward functions (i.e., format, syntax check, schema linking, n-gram similarity, EX, and LLM-as-a-judge rewards) to produce a composite reward. Although Reasoning-SQL employed RL from AI Feedback, the calculation still relied on gold SQL queries. In contrast, this study focuses on training a point-wise LLM judge model, which can address the applicability in scenarios where gold SQL is unavailable. Additionally, some general reward methods, such as DeekSeek-GRM (Liu et al., 2025b), JudgeLRM (Chen et al., 2025a), and RM-R1 (Chen et al., 2025b), employ reasoning models as generative reward models. However, the final rewards utilized by these methods remain relatively coarse-grained, as they mainly focus on the ultimate answer outcomes while neglecting the more informative reasoning processes. Consequently,

they fail to meet the requirements for fine-grained evaluation of assessment tasks.

3 Our Approach

3.1 Overview

The framework of our proposed method RuCo-C is illustrated in Figure 2, comprising two core components. First, we design a rubric-integrated critique response and adopt a multi-agent framework that synergizes generation and verification processes for data construction, ensuring data quality to support the subsequent rubric aligned SFT training of the critique model. Second, we treat the trained SFT model as the initial policy model for RL training, integrating fine-grained reward functions with a consistency mechanism that incorporates static and dynamic weighting. This yields a generative evaluation model capable of assessments without reliance on gold queries.

3.2 Rubric-integrated Critique Response Generation

For an interpretable evaluation model, we expect its output to include not only direct binary classification results but also specific evaluation criteria and corresponding rationale. Furthermore, for SQL statements identified as erroneous, we anticipate that the model can correct such errors. Therefore, we design an output structure comprising three key elements, referred to as a **critique response**. For more intuitive understanding, a concrete example of a critique response is provided

in Figure 1. The stepwise rubric-aligned evaluation reasoning process is presented between tags ‘<think>’ and ‘</think>’, a ‘False’ judgment result indicates that the predicted SQL contains certain inconsistencies with the rubric’s logic or grammar. The corrected SQL queries for erroneous predictions are enclosed within the ‘<correctedSQL>’ and ‘</correctedSQL>’.

Unlike text-to-SQL generation models, our generative critique model requires not only user questions, corresponding database metadata, and model-predicted SQL, but also critique feedback (i.e. critique response). To reduce the cost of manual annotation, we propose a generative and verification based multi-agent architecture (Sengupta et al., 2025; Tang et al., 2025) to synthesize high-fidelity training datasets.

Specifically, we implement an agent-based workflow in the data construction pipeline. As shown in Figure 2, the Memory Agent maintains a historical feedback buffer, which serves as a knowledge repository to inform subsequent iterative refinement processes. The Feedback-Gen Agent utilizes a self-questioning, stepwise reasoning paradigm to construct sample-specific critique metrics and associated reasoning trajectories. Meanwhile, the Correct-SQL Agent conducts systematic rectification of faulty SQL statements, ensuring data integrity and semantic consistency across the dataset. These agents operate in a synergistic manner, leveraging their complementary capabilities to optimize the end-to-end data generation pipeline. Upon data generation, the Align Agent employs a two-stage validation mechanism, comprising the execution accuracy method (Zhong et al., 2020) and the partial matching algorithm (Zhan et al., 2025) to filter instances where binary classification results align with ground truth labels or corrected SQL statements match the gold SQL.

3.3 Training of RuCo-C Model

The training of the RuCo-C model is divided into two phases: Supervised Fine-Tuning (SFT) and Reinforcement Learning (RL).

3.3.1 Rubric Aligned SFT Training

Problem Formulation. To train a generative critique model, we formulate the input and output of the model, which refer to the information in the user prompt and the LLM response. The input of sample i contains the question, database schema and the predicted SQL, which is denoted

as $X^i = \{q, m, \hat{c}\}$, q represents the question, m is corresponding database schema, \hat{c} denotes the predicted SQL which is generated by the model being evaluated. The output is the critique response, which contains the stepwise rubric reasoning process $s^i = \{s_1, s_2, \dots, s_{N_i}\}$, the classification result of predicted SQL \hat{y}^i , and the refined SQL statements \tilde{c}^i . Each reasoning information contains the specific rubric question b and answers a , then the k -th reasoning step can be denoted as $s_k = \{b_k, a_k\}$. Since the rubric information for each sample is implicit in the stepwise reasoning process, a rubric-aligned critique model can be trained on the aforementioned input-output data.

Supervised Fine-Tuning. In this study, we conduct SFT model to enhance the model’s capacity for instruction adherence and generation within the text-to-SQL judgment domain. We investigate two distinct SFT training strategies. The first one employs instructions focused solely on binary classification to determine the correctness of predicted SQL. The second one adopts rubric reasoning generation instructions, which prompt the development of rubric-based reasoning processes prior to reaching the final conclusion. Additionally, by appropriately controlling the ratio of positive to negative samples in the training data, we aim to obtain a critique model with unbiased category judgment.

3.3.2 RL Training with Novel Reward Design

Reinforcement Learning. During the reinforcement learning stage, we integrate the Group Relative Policy Optimization (GRPO) algorithm (Shao et al., 2024) into our training framework. This algorithm eliminates reliance on a value model, boasts lower memory consumption, and allows for precise formulation of reward objectives. These qualities make it an ideal selection for efficiently optimizing the policy model of text-to-SQL evaluators.

For each natural language question paired with its corresponding database schema and the predicted SQL statements, the policy model generates a set of G candidate critique responses $O = \{o_1, o_2, \dots, o_G\}$ through the old policy π_{old} . These candidates are rigorously evaluated using a composite reward function that assigns specific scores. By focusing on the relative performance of these critique response candidates within the group, GRPO effectively computes rewards for each output, thereby directing policy updates to align with our predefined objectives:

$$\mathcal{J}_{GRPO}(\theta) = \mathbb{E}_{\mathbf{u} \sim P(\mathbf{U}), \{o_i\}_{i=1}^G \sim \pi_{old}(O|\mathbf{u})} \left[\frac{1}{G} \sum_{i=1}^G (\min(r_i^{ratio} A_i, \text{clip}(r_i^{ratio}, 1 - \epsilon, 1 + \epsilon) A_i) - \beta D_{KL}(\pi_\theta || \pi_{ref})) \right], \quad (1)$$

where $r_i^{ratio} = \frac{\pi_\theta(o_i|\mathbf{U})}{\pi_{old}(o_i|\mathbf{U})}$ denotes the importance sampling ratio, which measures the relative probability of generating critique output o_i under the new policy π_θ as opposed to π_{old} . A_i refers to the advantage computed exclusively from the relative rewards of outputs within each group. ϵ is clipping-related hyperparameter for stabilizing training, the hyperparameter β controls the regularization of divergence between the trained policy π_θ and the reference policy π_{ref} .

Reward Function Design. Conventional reward mechanisms often struggle to capture the intricacies of reasoning processes and are vulnerable to the influence of noisy data. To overcome these limitations and enhance model performance, we developed a multi-component reward design. This design aims to address key challenges, including ensuring model performance with limited outcome-based rewards, preventing oversight of intermediate reasoning steps, and mitigating the impact of noisy dataset labels. By dynamically combining outcome-based and process-aware signals, adjusted according to their credibility, we seek to explore whether a more comprehensive reward structure can better achieve improved performance.

Intuitively, the basic reward components centered on output format correctness R_{format} and final binary judgment outcome R_{out} are essential for meeting basic task requirements. However, recognizing the importance of intermediate rubric reasoning steps for robust model behavior, we incorporated a process reward (PR) by evaluating the correctness of answers in the critique model’s reasoning trajectory (aligned with the query and given information). This process reward refines the reward signal based on step-by-step reasoning accuracy, testing whether granular process supervision, conditional on a correct final judgment, can enhance model reliability. Specifically, the rubric-guided process reward score is designed as:

$$R_{rubric} = 1 - \frac{1}{N} \sum_{i=1}^N \mathbb{I}(a_i \text{ is incorrect}), \quad (2)$$

where the symbol a_i means the answer in the i -

th reasoning process. The final process reward score is derived by verifying the correctness of the answer at each individual step via strong LLMs.

Considering the consistency of final results and the partial guiding significance of reasoning processes, we meticulously design the coefficients for PR. The design is divided into two main components: static PR coefficient and dynamic PR coefficient, denoted as γ_s and γ_d respectively.

For static PR, we first consider whether the binary classification result predicted by the critique model is consistent with the ground-truth binary classification result y , which is $R_{out} = \mathbb{I}(R_{rubric} < 1) \oplus y$, the symbol \oplus means the XOR gate. When they are consistent (i.e. $R_{out} = 1$), a coefficient value matching the binary classification result is assigned. Otherwise refer to the situation of $R_{out} = 0$. To combat “reward hacking” caused by noisy labels in the dataset, where the model is unjustly penalized for sound reasoning due to conflicting flawed labels, we introduces a verify code (VC) mechanism to assess the credibility of intermediate reasoning, which the coefficient value denoted as R_{cons} . The whole static PR coefficient can be calculated as:

$$\gamma_s = \begin{cases} 2 * R_{rubric}, & R_{out} = 1 \\ R_{cons}, & R_{out} = 0 \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

For cases where there is inconsistency with the final result, we further evaluate whether the reasoning process identifies errors. If an error is identified and the corrected SQL generated by the critique model passes verification, a small reward is given. Conversely, if an error is identified but the corrected SQL fails verification, which indicates low credibility in the critique model’s reasoning process, then a small penalty coefficient is applied. The detailed design is formulated as:

$$R_{cons} = \begin{cases} 1, & \mathbb{I}(R_{rubric} < 1) = 1 \ \& \ R_{verify} = 1 \\ -1, & \mathbb{I}(R_{rubric} < 1) = 1 \ \& \ R_{verify} = 0 \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

Here the $R_{verify} \in \{0, 1\}$ means the corrected SQL fail or pass the verification, we use Test-Suite as the verifier by comparing between the predicted SQL and the corrected SQL. By validating the consistency of the generated correction SQL and the SQL being judged, we can provide positive feedback for sound reasoning despite label noise, ex-

Method Type	Method	Spider			BIRD			Spider-DK		
		AUC	ACC	F1	AUC	ACC	F1	AUC	ACC	F1
Proprietary	o3-mini	67.68	66.24	72.32	64.44	49.08	46.04	75.54	67.61	60.52
	GPT-4.1	<u>68.80</u>	67.88	71.46	67.55	58.08	48.64	<u>76.28</u>	72.19	62.12
Open-Source	DeepSeek-R1-250120	69.72	69.40	69.93	67.98	63.29	49.33	77.20	<u>77.41</u>	64.55
	Deepseek-V3-241226	64.67	62.90	71.04	61.31	42.32	43.87	72.85	63.29	57.70
	Kimi-K2	66.55	64.96	<u>71.88</u>	63.25	47.70	45.12	75.79	67.50	60.70
	DeepSeek-Coder-6.7B-Instruct	59.60	59.00	61.79	55.42	48.94	38.11	58.75	53.54	44.67
	Qwen2.5-Coder-7B-Instruct	56.05	53.71	66.64	54.25	33.19	39.49	60.85	44.17	48.17
	Qwen2.5-Coder-14B-Instruct	59.68	57.54	68.50	56.12	33.83	40.77	66.04	52.96	51.77
SFT based	Qwen2.5-Coder-7B-Instruct	56.36	57.54	43.89	56.51	60.80	36.54	56.95	63.83	38.44
	Qwen2.5-Coder-14B-Instruct	65.87	66.24	62.37	72.21	67.15	53.65	70.15	75.49	56.19
RL based	Qwen2.5-Coder-7B-Instruct	68.76	<u>68.37</u>	69.30	66.60	65.57	48.10	74.86	75.01	61.53
	Qwen2.5-Coder-14B-Instruct	66.63	66.48	66.13	67.97	<u>69.13</u>	49.81	69.76	64.05	55.07
	RuCo-C (7B, BIRD trainset)	68.15	68.07	67.33	<u>72.40</u>	68.29	<u>54.04</u>	<u>75.60</u>	77.84	<u>63.12</u>
	RuCo-C (14B, BIRD trainset)	67.40	67.82	63.69	72.56	75.78	56.12	70.08	<u>77.41</u>	56.29

Table 1: Performance comparison of various methods across three testing datasets: **best results** are highlighted in bold, and *second-best* results are indicated in underline.

410 ploring whether this mechanism can reduce the
411 adverse effects of noisy data.

412 For dynamic PR, it is important to note that the
413 total number of steps in the reasoning process may
414 vary across different questions. Based on our sta-
415 tistical data, the number of steps in the reason-
416 ing process exhibits a clear correlation with ques-
417 tion difficulty. Specifically, questions of easy and
418 medium difficulty typically involve around 5 rea-
419 soning steps, while those of hard and extra diffi-
420 culty generally require 6 to 7 or more steps. Given
421 this data characteristic, we further introduce a dy-
422 namically adjusted coefficient γ_d for the rubric-
423 based process reward, which is defined based on
424 the number of reasoning steps as follows:

$$425 \gamma_d = \begin{cases} 1, & N_i > 5 \\ 0, & \text{otherwise} \end{cases} \quad (5)$$

426 Hence, the final total reward score can be formu-
427 lated as:

$$428 R_{total} = R_{format} + 2 * R_{out} + (\gamma_s + \gamma_d) * R_{rubric} \quad (6)$$

429 4 Experiment

430 4.1 Experimental Setup

431 **Dataset.** Our experiments are validated in three
432 public text-to-SQL datasets: Spider (Yu et al.,
433 2018), BIRD (Li et al., 2023) and Spider-DK (Gan
434 et al., 2021). Since the information contained in
435 these development datasets does not align with the
436 requirements of the evaluation scenario, we con-
437 structed additional negative sample data based on

438 them. The detailed construction process can be
439 seen in Appendix A. We conducted statistical anal-
440 yses on the size and question difficulty distribution
441 of the three datasets, with detailed information pro-
442 vided in Table 3.

443 **Baselines.** We compared open-source and prop-
444 rietary LLMs as baselines, all of which were
445 tested via the Prompt Engineering (PE) approach.
446 For reproducibility, the prompts used in these
447 baseline methods are identical to those in our
448 model’s training and inference, please refer to
449 the section D.3 in Appendix for details. Among
450 proprietary models, we selected OpenAI’s state-
451 of-the-art o3-mini and GPT-4.1. As for open-
452 source models, we chose large foundational mod-
453 els (e.g., DeepSeek-R1 (DeepSeek-AI et al.,
454 2025), DeepSeek-V3 (DeepSeek-AI et al., 2024)
455 and Kimi-K2 (Team et al., 2025)) based on
456 scale, plus smaller-scale models (e.g., about 7B
457 scale model DeepSeek-Coder-6.7B-Instruct and
458 Qwen2.5-Coder-7B-Instruct, and 14B scale model
459 Qwen2.5-Coder-14B-Instruct) for comparison. Ad-
460 ditionally, we compared the performance of mod-
461 els trained with SFT, and further evaluated the
462 model’s effectiveness with and without the critique
463 responses for its outputs.

464 **Evaluation Metrics.** For evaluating the correct-
465 ness of text-to-SQL, we employ metrics such as
466 *AUC*, *accuracy (ACC)*, and *F1-score (F1)*. Con-
467 sidering that the classes are imbalanced in our test-
468 ing dataset, we adopt AUC metric to measure the
469 model’s ability of distinguishing correct and wrong
470 predictions, and the F1-score is used to balance

Method	Spider			BIRD			Spider-DK											
	AUC	ACC	F1	AUC	ACC	F1	AUC	ACC	F1									
EX (w/o rubric)	58.73	59.31	52.92	67.10	66.17	48.65	59.00	67.55	40.12									
EX	65.19	↑ 6.36	66.06	↑ 6.75	58.05	↑ 5.13	66.88	↓ 0.22	75.01	↑ 8.84	49.04	↑ 0.39	69.59	↑ 10.59	79.01	↑ 11.46	55.73	↑ 15.61
EX+PR (static)	65.40	↑ 6.57	64.90	↑ 5.59	67.12	↑ 14.20	68.97	↑ 1.87	62.71	↓ 4.54	50.18	↑ 1.53	74.92	↑ 15.92	74.16	↑ 6.61	61.35	↑ 21.23
EX+RR+VC (static)	66.36	↑ 7.63	66.12	↑ 6.81	66.34	↑ 13.42	70.58	↑ 3.48	69.97	↑ 3.80	52.65	↑ 4.00	75.46	↑ 16.46	77.73	↑ 10.18	62.94	↑ 22.82
EX+PR (static+dynamic)	67.34	↑ 8.61	67.46	↑ 8.15	65.42	↑ 12.50	70.83	↑ 3.73	72.35	↑ 6.18	53.37	↑ 4.72	74.90	↑ 15.90	78.48	↑ 10.93	62.59	↑ 22.47
EX+PR+VC (static+dynamic)	68.15	↑ 9.42	68.07	↑ 8.76	67.33	↑ 14.41	72.40	↑ 5.20	68.29	↑ 2.12	54.04	↑ 5.39	75.60	↑ 16.60	77.84	↑ 10.29	63.12	↑ 23.00

Table 2: Ablation study results on RL reward function designing across three testing datasets, all variants use Qwen2.5-Coder-7B-Instruction as the backbone model. **Best results** are highlighted in bold, with accompanying performance improvements or declines calculated relative to the EX (w/o rubric) method.

the trade-off between avoiding false positives and capturing true positives. To offer a straightforward overall performance, the ACC is used to represent the proportion of correct predictions. To facilitate result comparison, all numerical results in this paper are presented in percentage form.

Implementation Details. The SFT and RL experiments were performed based on 8 GPUs. For RL training, we employed the GRPO strategy (Shao et al., 2024) based on the verl architecture (Sheng et al., 2025), with the following hyperparameters configurations: a batch size of 32, a learning rate of $1e^{-6}$, and 5 rollouts. Additional details about hyperparameters are provided in Appendix B.2.

4.2 Performance Evaluation

We conduct a comparative analysis of our proposed method RuCo-C against the performance of baselines. To evaluate generalization capability, our RuCo-C model was trained on only a subset of the BIRD training dataset during the GRPO phase. Evaluation results across three test datasets are presented in Table 1, from which the following key findings are derived: i) Both open-source and proprietary LLMs exhibit significantly superior evaluation performance across all three test datasets compared to small-scale models with 7B or 14B parameters. ii) For models trained via SFT, where training was restricted to a subset of the Spider dataset, cross-scale comparisons of model parameters reveal that 14B-parameter models yield more pronounced performance gains from SFT than their 7B-parameter counterparts. iii) Our proposed model RuCo-C, which integrates the custom-designed reward function into RL training, achieves notably better evaluation performance than SFT-trained models. Even when RL training is confined to a subset of the BIRD dataset, the model still demonstrates substantial improvements in inference performance on the other two test datasets. iv) Comparing two segments of the RL-based method (both RL-trained under our designed reward function), the only distinction is whether the RL train-

ing is initialized from an SFT-trained model. Results from the upper segment (direct RL without prior SFT) confirm the necessity of SFT in our framework. Notably, SFT endows the model with human-aligned critic ability and superior generalization over direct RL.

4.3 Ablation Study

To verify the rationality and effectiveness of the reward design, we conducted a series of ablation studies of GRPO methods with varied reward signals, i.e., EX, EX+PR and EX+PR+VC (RuCo-C), the detailed description can be seen in Appendix B.2.

The performance of the aforementioned variant methods is presented in Table 2. Analysis of these results yields the following conclusions. First, the two methods at the top of the table are both EX variants, differing in whether rubric information is output in the evaluation results during the SFT phrase. A comparison of their results demonstrates that incorporating rubric-based critique responses into SFT training achieves superior effectiveness. Second, comparisons between the two types of methods (EX+PR and EX+PR+VC) in the middle and bottom sections of the table confirm the validity of integrating the verify code strategy for overall performance. This effectiveness is particularly notable under static coefficients, with an average 4% improvement observed in the ACC metric. Finally, overall comparisons indicate that adding our designed process reward to the judge model significantly enhances evaluation performance, compared to judge models that use only EX as the reward. Furthermore, we integrate process reward with both static and dynamic coefficients, which further improves the model’s overall evaluation capability.

4.4 Reward Score Consistency Analysis

To more intuitively illustrate the relationship between designed reward scores and ground truth labels, we used box plots and kernel density plots to compare the distributions of reward scores and true labels of EX method and RuCo-C method

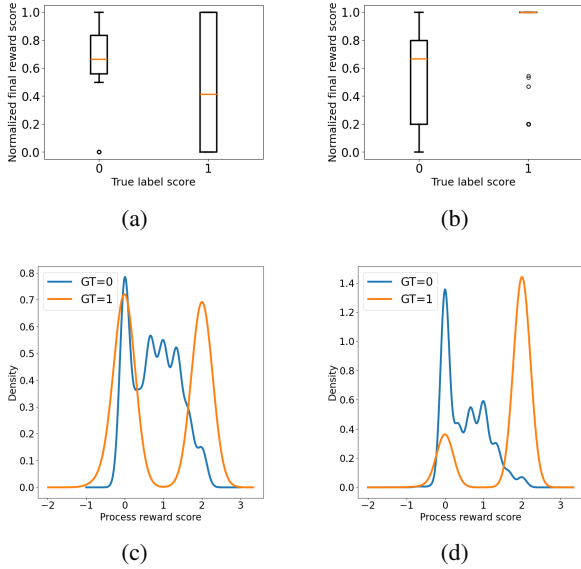


Figure 3: Box and kernel density plots comparing the distributions of ground truth labels and reward function under the EX (left) and our RuCo-C method (right).

(Figure 3). Box plots (a)(b) show the central tendency differences of normalized final reward scores for positive/negative samples, while kernel density plots (c)(d) display the probability density distributions of process reward scores for GT=1 (positive) and GT=0 (negative) samples.

It can be clearly observed that, compared with plot (a), negative sample reward scores in plot (b) are more concentrated. Besides, the kernel density plots show significant overlap between two categories in (c) but much less in (d). This indicates RuCo-C outperforms EX in terms of classification effectiveness, verifying that process reward scores improve final classification performance and reduce false negatives.

4.5 Case Study

As a query-based comparative assessment method, EX scoring may produce false judgments in specific scenarios, due to three main reasons: 1) inaccurate gold SQL directly causing misclassification; 2) most benchmarks providing only one reference answer, making it difficult to identify alternative solutions; 3) incomplete database values leading to inadequate and inaccurate assessments.

Figure 4 illustrates that EX scoring errors can occur when the execution result does not match the user’s required order, especially due to incorrect gold SQL. In contrast, the critique response generated by our proposed RuCo-C can identify such issues in the predicted SQL without relying on the gold SQL. Specifically, it points out that the

Order-insensitive & Wrong Gold SQL

Question: Which country’s gas station had the first paid customer in 2012/8/25?

Pred SQL:

```

1 SELECT g.country
2 FROM transactions_1k t
3 JOIN gasstations g ON t.gasstationid = g.
   gasstationid
4 WHERE t.date = '2012-08-25'
5 LIMIT 1

```

Gold SQL:

```

1 SELECT T2.Country
2 FROM transactions_1k AS T1
3 INNER JOIN gasstations AS T2
4 ON T1.GasStationID = T2.GasStationID
5 WHERE T1.Date = '2012-08-25'
6 ORDER BY T1.Time DESC
7 LIMIT 1

```

Critique Response (part):

```

<think>
...
4: Did I correctly identify the first paid customer?
- The query uses LIMIT 1 to get the first transaction matching the date filter but
  doesn't guarantee it's the first paid customer by time. To do so, transactions
  should be ordered by time ascendingly.
</think>
<result> False </result>
...

```

Figure 4: The false positive example of EX scoring (EX score is 1 while the true label is 0), along with the generated critique response by our RuCo-C to indicate the correct judgment.

predicted SQL fails to align with the user’s intent of “first paid customer” and recommends sorting the transaction table by the time field in ascending order (ASC), whereas the gold SQL uses descending order (DESC), revealing the latter’s inaccuracy. Based on this critique, our model achieves a correct assessment. Additional case analyses are provided in Appendix C.

5 Conclusion

In this paper, we propose a novel generative judge model **RuCo-C** tailored for text-to-SQL evaluation. By integrating interpretable rubrics with the designed consistent reward functions, RuCo-C enables fine-grained, query-specific automatic evaluation, thereby eliminating the need for human intervention and addressing key bottlenecks in the text-to-SQL assessment field. Complementary experiments conducted on three distinct test sets demonstrate that our proposed model achieves superior evaluation performance compared to existing baseline methods. Looking ahead, we plan to explore the integration of RuCo-C into end-to-end text-to-SQL training pipelines, enabling real-time feedback during the post-training phase to accelerate performance improvement.

6 Limitations

Despite the proposed evaluation method eliminating reliance on SQL query comparisons, avoiding gold SQL annotation costs, and achieving better performance than existing baseline methods, it still has limitations. Specifically, the small size of the training dataset restricts the generalization ability of the post-training model, leaving room for further improvement. Accordingly, future work will explore integrating this evaluation method into training data synthesis and text-to-SQL model training. Generated results will be fed back into the training and capability enhancement of the evaluation model, ultimately forming a complete closed loop of data generation, training, and evaluation.

References

Ruisheng Cao, Lu Chen, Jieyu Li, Hanchong Zhang, Hongshen Xu, Wangyou Zhang, and Kai Yu. 2023. A heterogeneous graph to abstract syntax tree framework for text-to-sql. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(11):13796–13813.

Nuo Chen, Zhiyuan Hu, Qingyun Zou, Jiaying Wu, Qian Wang, Bryan Hooi, and Bingsheng He. 2025a. Judgelrm: Large reasoning models as a judge. *arXiv preprint arXiv:2504.00050*.

Xiusi Chen, Gaotang Li, Ziqi Wang, Bowen Jin, Cheng Qian, Yu Wang, Hongru Wang, Yu Zhang, Denghui Zhang, Tong Zhang, and 1 others. 2025b. Rm-r1: Reward modeling as reasoning. *arXiv preprint arXiv:2505.02387*.

Ziru Chen, Shijie Chen, Michael White, Raymond Mooney, Ali Payani, Jayanth Srinivasa, Yu Su, and Huan Sun. 2023. Text-to-sql error correction with language models of code. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1359–1372.

Ganqu Cui, Yuchen Zhang, Jiacheng Chen, Lifan Yuan, Zhi Wang, Yuxin Zuo, Haozhan Li, Yuchen Fan, Huayu Chen, Weize Chen, Zhiyuan Liu, Hao Peng, Lei Bai, Wanli Ouyang, Yu Cheng, Bowen Zhou, and Ning Ding. 2025. The entropy mechanism of reinforcement learning for reasoning language models. *Preprint*, arXiv:2505.22617.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 181 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, and 81 others. 2024. Deepseek-v3 technical report. *CoRR*, abs/2412.19437.

Yujian Gan, Xinyun Chen, and Matthew Purver. 2021. Exploring underexplored limitations of cross-domain text-to-sql generalization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8926–8931.

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. *Proceedings of the VLDB Endowment*, 17(5):1132–1145.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jianguang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-sql in cross-domain database with intermediate representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, and 4 others. 2025. Tulu 3: Pushing frontiers in open language model post-training. *Preprint*, arXiv:2411.15124.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tiejing Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. 2025. Omnisql: Synthesizing high-quality text-to-sql data at scale. *Proceedings of the VLDB Endowment*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiayi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357.

Zhishuai Li, Xiang Wang, Jingjing Zhao, Sun Yang, Guoqing Du, Xiaoru Hu, Bin Zhang, Yuxiao Ye, Ziyue Li, Rui Zhao, and Hangyu Mao. 2024. Pet-sql: A prompt-enhanced two-stage text-to-sql framework with cross-consistency. *CoRR*, abs/2403.09732.

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 Transactions on Knowledge and Data Engineering*, pages 1–20.

Zijun Liu, Peiyi Wang, Runxin Xu, Shirong Ma, Chong Ruan, Peng Li, Yang Liu, and Yu Wu. 2025b.

831 *on Knowledge and Data Engineering*, 36(11):6699–
832 6718.

833 Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic
834 evaluation for text-to-sql with distilled test suites. In
835 *Proceedings of the 2020 Conference on Empirical*
836 *Methods in Natural Language Processing (EMNLP)*,
837 pages 396–411.

838 Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham
839 Neubig. 2023. Codebertscore: Evaluating code gen-
840 eration with pretrained models of code. In *The 2023*
841 *Conference on Empirical Methods in Natural Lan-*
842 *guage Processing*.

Distribution	Label	Spider	BIRD	Spider-DK
	Total	1644	2977	1877
Binary Distribution	Positive	776 (47.20%)	693 (23.28%)	503 (26.80%)
	Negative	868 (52.80%)	2284 (76.72%)	1374 (73.20%)
Hardness Distribution	Easy	57 (3.47%)	18 (0.60%)	79 (4.21%)
	Medium	620 (37.71%)	1047 (35.17%)	1017 (54.18%)
	Hard	246 (14.96%)	811 (27.24%)	247 (13.16%)
	Extra	721 (43.86%)	1101 (36.98%)	534 (28.45%)

Table 3: Statistical overview of binary and difficulty distributions across three testing datasets.

A Dataset Construction

In this chapter, we elaborate on the construction methods of the training sets and test sets employed in the experiments, including key details of data sources, negative sample generation, sample sizing, and data validation as required.

A.1 Training Dataset

The training data is primarily divided into two phases: the SFT phase and the RL phase. The data source, negative sample generation method, sample size, and data validation details for each phase are explicitly clarified as follows:

SFT Phase Data Construction. For the SFT phase, we mainly conduct data synthesis based on the Spider training data. The original Spider (Yu et al., 2018) dataset contains 10,181 questions and 5,693 complex SQL queries, sourced from 200 multi-table databases across 138 domains. The entire dataset is divided into three non-overlapping subsets: training (8,659), development (1,034) and test datasets. Since the original Spider dataset includes only gold SQL queries, our experiments require a large number of erroneous SQL queries as negative samples. To address this, we adopt a synthesized dataset (Chen et al., 2023) with erroneous SQL queries based on the training set of Spider: this dataset retains the question information from the original Spider and generates multiple erroneous SQL queries for each question. Using this dataset, we constructed approximately 40,000 data samples and generated critique responses for both positive and negative samples via the proposed multi-agent framework. Considering the need for balanced positive-negative sample ratios, we ultimately sampled and used 15,000 balanced samples as the training dataset for the SFT.

RL Phrase Data Construction. For the RL phase, we construct data using the more complex BIRD training data. The original BIRD (Li et al., 2023) is another large-scale and cross-domain text-

to-SQL dataset known for its complexity. It contains 9,428 training examples and 1,534 development examples, which drawn from 95 databases cross more than 37 professional fields, making it a challenging testbed for evaluating the generalization capability in text-to-SQL task. In RL training, we also need to construct a set of negative samples. To this end, we generate SQL queries using several text-to-SQL models, determine their correctness via existing evaluation methods (e.g., Test-Suite Accuracy), and filter out samples with erroneous SQL queries as negative samples.

While we now have positive and negative samples along with their corresponding labels as ground truth for feedback signals, the BIRD dataset is widely recognized for its “dirty” characters: potentially incorrect SQL queries, ambiguous or poorly described column names, and databases containing null values or irregular encodings. Consequently, we sample and further validate the data, using a combination of multiple automated evaluation methods and limited manual checks to finalize whether each sample is positive or negative, ensuring the reliability of the RL training data.

A.2 Testing Dataset

For the testing dataset, we constructed positive and negative samples following a similar approach to training dataset construction, using existing development sets as the basis. The statistical details of the final test sets employed in our experiments are presented in Table 3, which includes the binary classification distribution (positive/negative sample ratios) and question difficulty distribution (Easy/Medium/Hard/Extra) across three datasets (Spider, BIRD, Spider-DK).

In terms of binary classification distribution, we not only built a Spider test set with nearly balanced positive-negative ratios but also constructed BIRD and Spider-DK test sets with higher proportions of negative samples. This design aims to comprehen-

sively test the evaluation model’s ability to detect erroneous SQL. Additionally, regarding the distribution of question difficulty, our test set composition prioritized evaluating the model’s performance on questions of moderate or higher difficulty. The rationale behind this design is rooted in practical application scenarios: in real-world text-to-SQL tasks, users often encounter complex queries involving multi-table joins, nested subqueries, or advanced aggregation operations (i.e., moderate or higher difficulty questions), and the evaluation of model performance on such queries is more critical for practical deployment. To rule out the possibility that the proposed method performs poorly on easier SQL queries, we emphasize that our test sets (as shown in Table 3) still include a reasonable proportion of easy questions. Moreover, subsequent performance comparison results grouped by question hardness (Section corresponding to Figure 7) will explicitly demonstrate that the proposed method achieves competitive performance on easy questions, ensuring its effectiveness across all difficulty levels. This design balances the focus on practically critical moderate/high-difficulty questions and the comprehensive validation of performance across all difficulty scales.

B Implement Details

B.1 Baselines Implementation

In this subsection, we illustrate the detailed implementation of the baselines used in our experiments, including explicit clarification of baseline types and strict guarantee of prompt consistency.

Baseline Types. The baselines include both proprietary models (e.g., GPT-4.1) and open-source models (e.g., DeepSeek-R1 and Kimi-K2). For all these open source and proprietary models, we utilize the same prompts (as shown in D.3) that were employed in the training phase of our proposed model (RuCo-C) for inference to ensure consistency. This consistent prompt configuration eliminates the interference of prompt differences on the evaluation results, making the performance comparison between baselines and our model more fair and reliable.

Inference and Training Configuration. For each test set, we perform three separate inference runs and average the results to yield the final performance, reducing the randomness of single inference results. Regarding SFT training, data was sampled from the constructed training set as illus-

trated in Section A to maintain a balance between positive and negative samples; subsequently, the appropriate SFT checkpoint was selected as the initial policy model for the GRPO training.

B.2 Ablation Studies Implementation

The ablation studies of the GRPO methods shown in Table 2 are all based on Qwen2.5-Coder-7B-Instruct, which mainly include three types of variant, within each variant category, there are two distinct methods:

- EX as a reward: the reward score used in GPRO consists primarily of a format score (check the format of critique response) and a correction score (check the binary result is consistent with the ground-truth label).
 - EX (w/o rubric): during the SFT training phrase, the training examples do not include any rubric-related critique response but only binary results.
 - EX: during the SFT training phrase, the training examples contain the critique response as in RuCo-C.
- EX+PR as a reward: Based on using EX as reward, this variant further checks the QA pairs (i.e., content in tag <think> of critique response) generated by the judge model to determine whether the answer in each QA pair is correctly elaborated, i.e., whether the elaboration aligns with the intentions of the query and the given information.
 - EX+PR (static): based on the coefficient γ_s but without considering the situation $R_{out} = 0$.
 - EX+PR (static+dynamic): based on the EX+PR (static) variant, further consider the coefficient γ_d to dynamically combine the rubric-guided process reward R_{rubric} .
- EX+PR+VC as a reward: Based on EX+PR, we further verify the credibility of the judge information through the verify code (VC) method: by re-judging the generated correction SQL, if the correction SQL is correct, the PR credibility is considered positive; otherwise, it is negative.
 - EX+PR+VC (static): only using coefficient γ_s to combine the rubric-guided process reward R_{rubric} .

Hyperparameter	Default Value
Optimizer	AdamW
Learning Rate	$1e^{-6}$
Training Batch Size	32
Training Total Epochs	3
Max Prompt Length	4096
Max Response Length	2048
PPO Mini Batch Size	8
Log Probability Micro Batch Size per GPU (Rollout)	8
Tensor Model Parallel Size (Rollout)	2
Group Sampling Number (Rollout)	5
GPU Memory Utilization (Rollout)	0.6
Temperature (Rollout)	1.0
Do Sample (Rollout)	True
Clip Ratio (ϵ in Eq. (1))	0.2
KL Loss Coefficient (β in Eq. (1))	0.001
Normalize Advantages by Standard Deviation	True

Table 4: Hyperparameters setting during GPRO training.

- EX+PR+VC (static+dynamic): using the final total reward score as shown in Eq. (6).

All the aforementioned variant methods are trained based on the verl architecture (Sheng et al., 2025) and share largely consistent core experimental hyperparameters, with specific details provided in Table 4. There are some different settings compared to the original GRPO paper, for example, we adopt the “token-mean” configuration for loss aggregation instead of sample-level loss which may be unstable in long-CoT scenarios.

B.3 Visualized Comparison on Training Process

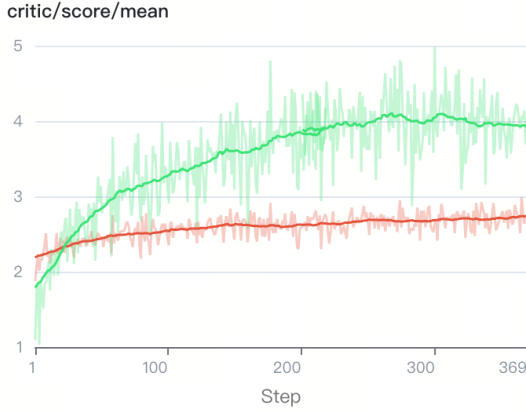
To visually examine the GPRO process under different reward functions, we present the variation trends of reward scores across training steps (Figure 5), along with changes in entropy loss and generated response lengths during training (Figure 6). In this two figures, the red curve corresponds to the GPRO method using only EX rewards, while the green curve represents our proposed method RuCo-C integrating process and outcome rewards.

As shown in Figure 5, the left and right panels depict the trend of average reward scores across training steps and reward score changes on the validation set, respectively. It can be observed that although the reward score of our method was lower than that of the EX method at the initial training

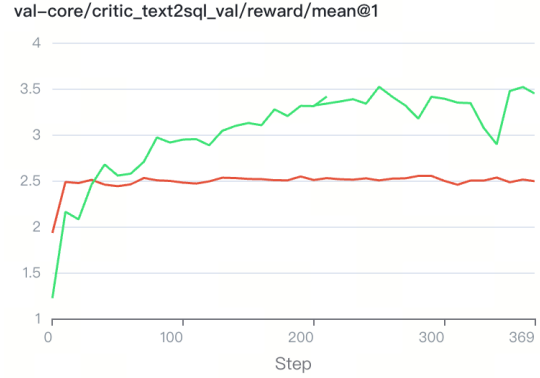
stage, as training iterations progressed, the model captured more feedback signals, guiding it toward optimization for better reward objectives. Both the training and validation sets exhibit a distinct upward trend in reward scores. In contrast, the red curve (EX method) shows limited growth in reward scores. This may stem from the discrete distribution of EX-based reward scores, which fails to reflect fine-grained feedback signals for individual samples, thereby restricting its ability to guide model optimization.

Despite the differences in the numerical ranges of the reward scores that our method shows a wider range and higher values in the growth trend of the training set, the results of the validation set clearly indicate that the EX method reaches a plateau early in training without further improvement. This confirms that the performance of our method continues to improve throughout the training process.

Figure 6a illustrates the trend of entropy loss in the policy model during RL training for both methods. In reinforcement learning, an action that receives both high/low probability and high/low advantage will lower the entropy, and vice versa. According to the entropy mechanism (Cui et al., 2025): at the beginning stage, the policy exhibits a high covariance with the training data, indicating that its confidence is well calibrated. This allows it to safely exploit high-confidence trajectories, strengthening its beliefs and minimizing entropy.



(a) Average reward score for batched training data



(b) Reward score of validation set

Figure 5: The reward score trends on the training set (left) and validation set (right) during GRPO training steps. The red curve corresponds to baseline model based on the EX reward, while the green curve indicates to our RuCo-C model using designed reward function. Both model are training with Qwen2.5-Coder-7B-Instruct backbone under the GRPO framework. Despite starting from a relatively low initial score, our RuCo-C model exhibits a distinct upward trend, indicating that the feedback obtained with increasing training steps yields significant optimization for the model.

The trends of the two curves in Figure 6a generally confirm the effectiveness of both methods in GPRO training. Comparing their values, the entropy loss of our model RuCo-C consistently remains below that of the EX method, indicating that our method has higher confidence in exploration. Additionally, based on the negative exponential relationship between performance and entropy, this indirectly demonstrates that our method outperforms the EX method in terms of performance.

Furthermore, we also analyzed the length of the responses generated by the model during training, with relevant results presented in Figure 6b. It can be observed that between training steps 100 and 200, the response length generated by our model showed a significant reduction compared to the baseline model, and then stabilized around 300 steps. Given that the response comprises three components, the primary reduction in length is likely concentrated in the rubric-related reasoning phase. This indicates that as confidence in training exploration increases, the rubric reasoning information becomes more explicit and focused. By eliminating irrelevant rubric-related questions, the model can converge more quickly to core evaluation criteria while maintaining its evaluation capability.

B.4 Performance Comparison Group By Hardness

Beyond the overall performance comparison, we further explore the performance differences of the

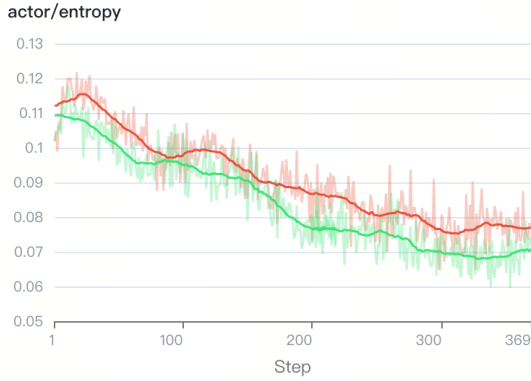
models across varying difficulty levels of the questions (named hardness here). The performance metrics used in this comparison include AUC (distinguishing correct/erroneous SQL) and F1-score (balancing precision/recall), which are defined in Section 4.1 (Evaluation Metrics) and linked to the core evaluation target (SQL correctness judgment + critique generation) in Section 4.2.

According to the gold SQL per question, we systematically evaluate SQL query complexity, a four-level classification framework (Easy, Medium, Hard, Extra) is proposed, based on quantitative statistics of complex syntax components and qualitative analysis of query structures. The core logic relies on three metrics: count of basic complex clauses (for instance, WHERE, GROUP BY, ORDER BY), count of advanced complex operations (like EXCEPT, UNION, INTERSECT, nested subqueries), and count of additional complex features. Table 5 below summarizes the core classification logic, typical characteristics, and representative SQL examples for each difficulty level. Notably, for evaluation purposes, Easy and Medium queries can be grouped into a “Basic Category”, this grouping is justified by their intrinsic differences, enabling more effective observation of evaluation result variations across difficulty levels.

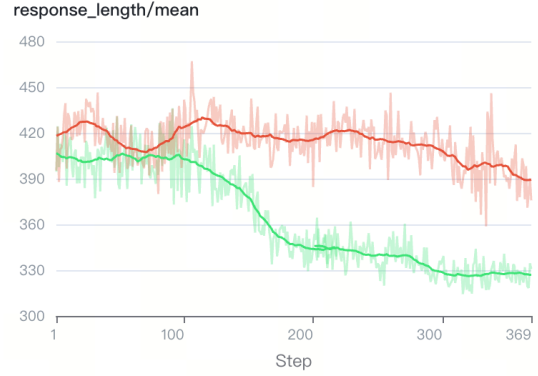
As shown in Figure 7, we compare our RuCo-C model with the EX-based reward model under different hardness in the BIRD test set. As indicated in Table 3, the BIRD test set contains

Hardness	Typical Characteristics	SQL Example
Easy	<ul style="list-style-type: none"> • Only basic SELECT statements • No JOIN/OR/LIKE/nested queries/set operations • Less than 1 selected column/aggregated function/WHERE condition 	<pre>SELECT name FROM users WHERE id = 1</pre>
Medium	<ul style="list-style-type: none"> • 1-2 basic complex clauses (e.g., 'WHERE' + 'GROUP BY') • No nested/set operations • Max 1 extra complex feature (e.g., >1 aggregate functions) 	<pre>SELECT department, COUNT(*) FROM employees WHERE salary > 5000 GROUP BY department</pre>
Hard	<ul style="list-style-type: none"> • Multiple basic complex clauses (greater than 3) • Simple nested subqueries or single set operation • Less than 2 extra complex features 	<pre>SELECT d.name, COUNT(e.id) FROM departments d JOIN employees e ON d.id = e.department_id WHERE e.salary > 5000 GROUP BY d.name ORDER BY COUNT(e.id) DESC LIMIT 10</pre>
Extra	<ul style="list-style-type: none"> • Numerous basic complex clauses (greater than 5) • Multi-layer nested queries or multiple set operations • Greater than 3 extra complex features • Combined advanced/basic complex components 	<pre>SELECT dept.name, SUM(e.salary) FROM (SELECT * FROM employees WHERE hire_date > '2020-01-01' UNION SELECT * FROM contractors WHERE hire_date > '2020-01-01') e JOIN departments dept ON e.dept_id = dept.id WHERE e.salary > 5000 GROUP BY dept.name HAVING SUM(e.salary) > 100000 ORDER BY SUM(e.salary) DESC</pre>

Table 5: The details of the classification of SQL query hardness, along with their core features and examples.

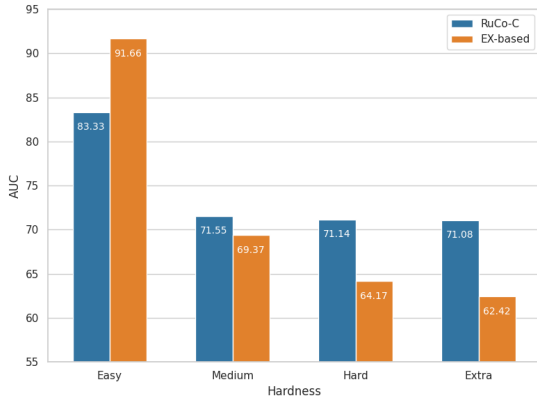


(a) The entropy loss under training steps

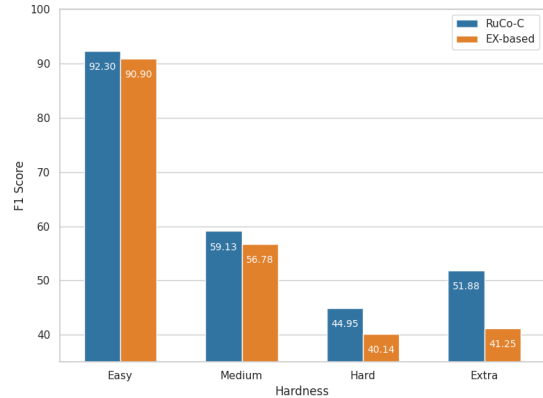


(b) The averaged response length under training steps

Figure 6: The trend of entropy loss and averaged response length during GRPO training steps. The red curve correspond to baseline model based on the EX reward, while the green curve is our RuCo-C model using designed reward function. Both model are training with Qwen2.5-Coder-7B-Instruct backbone under the GRPO framework.



(a) Performance comparison under AUC metric



(b) Performance comparison under F1-score metric

Figure 7: Performance comparison between our RuCo-C and baseline methods on AUC and F1-score metrics across different hardness of questions in BIRD test set.

1141 fewer easy questions, most of which are medium
 1142 or higher difficulty. Given the imbalanced positive-
 1143 negative sample ratio in the BIRD test set, we use
 1144 the AUC metric for comparison. It can be observed
 1145 that the EX-based method performs better on simple
 1146 questions, but our method demonstrates a significant
 1147 advantage on moderate and higher difficulty levels,
 1148 which account for a larger proportion of the dataset.
 1149 Additionally, our method achieves better results in
 1150 terms of F1-score, indicating that it also exhibits
 1151 superior performance in avoiding false positives
 1152 compared to the EX-based method.

1153 C Examples of False Annotations Based 1154 on EX

1155 As a pairwise assessment approach, EX scoring
 1156 has the potential to yield inaccurate outcomes in
 1157 specific scenarios. Firstly, inaccuracies in gold

SQL which can directly lead to misclassifications,
 1158 wrongly identifying correct results as false positives.
 1159 Secondly, dealing with multiple reference
 1160 answers poses a challenge. Most benchmarks offer
 1161 just one reference answer. As a result, EX scoring,
 1162 constrained by narrow criteria, struggles to
 1163 identify valid alternative solutions, incorrectly classifying
 1164 them as false negatives. Lastly, incomplete
 1165 database values contribute to misjudgments. Without
 1166 sufficient data for evaluation, EX scoring cannot
 1167 conduct comprehensive and accurate assessments,
 1168 resulting in false outcomes.
 1169

As illustrated in Figure 8, here is an example of a
 1170 false negative due to the inconsistency between the
 1171 given input and the gold SQL. In both the user’s
 1172 question and the evidence, the term is stated as
 1173 “cryokinesis” (with a lowercase initial letter). However,
 1174 the value corresponding to the “power_name”
 1175 field in the actual database is “Cryokinesis” (with
 1176

1177 an initial letter in uppercase). After executing the
1178 SQL statements, the gold SQL can retrieve the re-
1179 sult, while the predicted SQL returns an empty
1180 result, which leads to a false negative in the EX
1181 score. However, based on the contextual informa-
1182 tion given, the predicted SQL is correct, as it is
1183 derived from step-by-step reasoning in the QA pro-
1184 cess without knowing the complete values in the
1185 database.

1186 Another example (as shown in Figure 9) in-
1187 volves cases where there are field-value errors in
1188 the predicted SQL, yet its execution results are
1189 consistent with those of the gold SQL. Specifi-
1190 cally, the user’s query requires counting players
1191 involved in attack actions. The filter condition “de-
1192 fensive_work_rate=‘low’” in the gold SQL only
1193 emphasizes that the players have a low probability
1194 of participating in defense, without indicating a
1195 high probability of participating in attacking. Ad-
1196 ditionally, the predicted SQL contains an error in
1197 the value of ‘attack_work_rate’ in its filter con-
1198 dition, which should be ‘high’ instead of ‘defen-
1199 sive’. Such a value error evaded the executor’s er-
1200 ror detection capability due to the logical condition
1201 ‘!=’. However, in the critique response generated by
1202 our model, the red-highlighted text shows that our
1203 model can identify such errors and provide correct
1204 modification suggestions.

Input and Gold SQL Inconsistency

Question What is the power ID of cryokinesis ?

Evidence power ID refers to superpower.id, cryokinesis refers to power_name = 'cryokinesis'

Pred SQL

```
1 SELECT id
2 FROM superpower
3 WHERE power_name = 'cryokinesis'
```

Gold SQL

```
1 SELECT id
2 FROM superpower
3 WHERE power_name = 'Cryokinesis'
```

EX label: 0

Critique Response

<think>

1. Did I use the correct tables for the query?

- Yes, the superpower table contains the power_name and id columns, which are relevant to the question about the power ID of cryokinesis.

2. Did I correctly specify the column to select?

- Yes, the id column is the correct column to select as it represents the power ID.

3. Did I correctly filter the data based on the question?

- Yes, the WHERE clause correctly filters the rows where power_name is 'cryokinesis', which is the power mentioned in the question.

4. Did I include any unnecessary columns or calculations?

- No, the query is straightforward and only selects the id column, which is what the question asks for.

5. Have I ensured that my query accurately targets the required data without adding unnecessary complexity?

- Yes, the query is simple and directly answers the question without any unnecessary complexity.

</think>

<result> True </result>

Figure 8: False negative example of EX scoring errors, along with their corresponding critique response generated by our RuCo-C model.

Wrong Value but Same Execution Results

Question Among the players with an overall rating between 60 to 65, how many players whose going to be in all of your attack moves instead of defending?

Pred SQL

```
1 SELECT COUNT(*)
2 FROM player_attributes
3 WHERE overall_rating BETWEEN 60 AND 65
4 AND attacking_work_rate != 'defensive'
5 AND defensive_work_rate = 'low'
```

Gold SQL

```
1 SELECT COUNT(id)
2 FROM
3 Player_Attributes
4 WHERE overall_rating BETWEEN 60 AND 65
5 AND defensive_work_rate = 'low'
```

Execution Result of Pred SQL [(3594,)]

Execution Result of Gold SQL [(3594,)]

EX label: 1

Critique Response

<think>

1. Did I use the correct table for the query?

- Yes, the player_attributes table contains the overall_rating, attacking_work_rate, and defensive_work_rate columns, which are relevant to the question.

2. Did I correctly filter the players based on their overall rating?

- Yes, the condition overall_rating BETWEEN 60 AND 65 correctly filters players with an overall rating between 60 and 65.

3. Did I correctly interpret the question regarding attacking and defensive work rates?

- The question asks for players who are “going to be in all of your attack moves instead of defending.” This implies that the players should have a high attacking work rate and a low defensive work rate.

- The predicted SQL uses attacking_work_rate != 'defensive' and defensive_work_rate = 'low'. **However, the condition attacking_work_rate != 'defensive' is not specific enough. It should be attacking_work_rate = 'high' to ensure the player is focused on attacking.**

4. Did I correctly count the players?

- Yes, the COUNT(*) function is correctly used to count the number of players that meet the specified conditions.

5. Have I ensured that my query accurately targets the required data without adding unnecessary complexity?

- The query is straightforward and correctly targets the required data, but the condition for attacking_work_rate needs to be more specific.

</think>

<result> False </result>

<correctedSQL>

```
1 SELECT COUNT(*)
2 FROM player_attributes
3 WHERE overall_rating BETWEEN 60 AND 65
4 AND attacking_work_rate = 'high'
5 AND defensive_work_rate = 'low'
```

</correctedSQL>

Figure 9: False positive example of EX scoring errors, along with their corresponding critique response generated by our RuCo-C model.

1337
1338
1339
1340
1341
1342
1343
1344

```
- Predicted SQL:
```sql
{incorrect_sql}
```

- Expert Human Feedback: {feedback}

- Asking myself: step-by-step reasoning
```

Prompt 3: Prompt of Memory Agent.

1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1376

```
System Prompt:

You are a helpful AI assistant that
memorizes and manages the messages
from other assistants.

User Prompt:

Please review the following prompts and
outputs for the agent below and
generate a revised version.

So far you have revised the prompt {
iteration_of_try} times.

Agent description: {agent_description} #
The agent could be the Feedback_Gen
Agent and the Correct_SQL Agent.
The previous output of the agent that
was not useful: {agent_outputs[-1]}
The previous prompt of the agent that
you should revise: {agent_prompts
[-1]}
Expert Human Feedback: {feedback} # the
alignment result from Align Agent.

Revise prompt (Return the entire prompt
so it can be directly passed to the
agent):
```

D.2 Prompt for LLM-based Process Reward Model

1377
1378

Prompt 4: Prompt for the rubric process judging.

1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401

```
Evaluate the correctness of a predicted
SQL query and a provided correction
SQL query for a given user query and
schema by analyzing the reasoning
process. For each step in the
reasoning process, which includes a
Question and an Answer, separately
assess the correctness of the
Question and the Answer.
Additionally, verify whether the
correction SQL query correctly
satisfies the user query and
conforms to the given schema.
Provide justification for all
assessments and summarize the step
numbers of incorrect Questions and
Answers, as well as the correctness
status of the correction SQL, in a
JSON format.

## Steps
```

```
1. **Understand the User Query and
Schema**: Thoroughly understand the
user query and the provided schema
to form the basis for evaluating
both the predicted and correction
SQL queries.

2. **Analyze the Predicted SQL**: Review
the predicted SQL query to ensure
it aligns with the user query and
schema.

3. **Analyze the Correction SQL**:
Evaluate the correction SQL query to
determine if it correctly answers
the user query and is valid with
respect to the schema. Check for
syntactic correctness, semantic
alignment with the user query, and
schema compliance.

4. **Evaluate the Reasoning Process**:
For each step in the reasoning
process:
- **Question**: Determine if the
Question is relevant and correctly
framed based on the user query and
schema.
- **Answer**: Assess if the Answer
correctly addresses the Question and
aligns with the predicted SQL.

5. **Provide Justification**: For each
Question and Answer, provide a
detailed justification for your
assessment of its correctness. Also,
justify your evaluation of the
correction SQL query's correctness.

6. **Summarize Errors and Correction SQL
Status**: Identify and list the
step numbers of any incorrect
Questions and Answers. Indicate
whether the correction SQL query is
correct or incorrect.

## Output Format

- Provide a JSON object summarizing the
incorrect steps and the correction
SQL evaluation:
```json
{
 "incorrect_answers": [step_numbers],
 "incorrect_answer_ratio": 0.xx,
 // number of incorrect answers /
 total reasoning steps, rounded to
 two decimals
 "correction_sql_correct": true|false
 , // true if correction SQL
 correctly satisfies the user query
 and schema, else false
 "correction_sql_justification": "[
 detailed justification of the
 correction SQL correctness]"
}
```

## Examples
**Example 1:**
```

```

1472 - **User Query**: "Find the names of all
1473 employees in the 'Sales' department
1474 ."
1475 - **Schema**: Tables: Employees,
1476 Departments. Columns: Employees (id,
1477 name, department_id), Departments (
1478 id, name).
1479 - **Predicted SQL**: `SELECT Employees.
1480 name FROM Employees JOIN Departments
1481 ON Employees.department_id =
1482 Departments.id WHERE Departments.
1483 name = 'Sales';`
1484 - **Correction SQL**: `SELECT name FROM
1485 Employees WHERE department_id IN (
1486 SELECT id FROM Departments WHERE
1487 name = 'Sales');`
1488 - **Reasoning Process**:
1489 - **Step 1**:
1490 - **Question**: "What tables are
1491 involved in the query?"
1492 - **Answer**: "Employees and
1493 Departments."
1494 - **Step 2**:
1495 - **Question**: "How are the tables
1496 related?"
1497 - **Answer**: "Through the
1498 department_id in Employees and id in
1499 Departments."
1500 - **Step 3**:
1501 - **Question**: "What condition
1502 filters the results?"
1503 - **Answer**: "Departments.name = '
1504 Sales'."
1505 - **Assessment**:
1506 - **Incorrect Answers**: []
1507 - **Correction SQL Correctness**: true
1508 - **Justification**: The correction
1509 SQL correctly retrieves employee
1510 names from the Employees table where
1511 the department matches 'Sales' by
1512 using a subquery on Departments,
1513 which aligns with the user query and
1514 schema.
1515 - **Output**:
1516 ```json
1517 {
1518   "incorrect_answers": [],
1519   "incorrect_answer_ratio": 0.00,
1520   "correction_sql_correct": true,
1521   "correction_sql_justification": "The
1522 correction SQL correctly retrieves
1523 employee names from Employees where
1524 department_id matches Departments
1525 with name 'Sales', aligning with the
1526 user query and schema."
1527 }
1528 ---
1529 ## Notes
1530 - Ensure the reasoning process is
1531 logical and follows a clear sequence
1532 .
1533 - Pay close attention to the alignment
1534 between the user query, schema,
1535 predicted SQL, and correction SQL.

```

```

- Consider edge cases such as ambiguous
column names, complex joins, or
nested queries.
- The correction SQL correctness check
should include syntactic validity,
semantic correctness, and schema
compliance.
- Output only the JSON result as
specified, without additional
commentary.

```

D.3 Prompt Used for Critique Model Training and Inference

Prompt 5: Prompt for the training and inference of critique model.

```

## ROLE
You are an expert in identifying
code errors in the NL2SQL (Natural
Language to SQL) task. Based on the
provided question, table structure (
schema), and other information,
determine whether the corresponding
code (Predicted SQL) is correct.
Please follow the steps below:
- self ask step by step reasoning
- Determine whether the SQL solution
is correct based on the reasoning
results
- If there are errors, please
correct the mistakes in the SQL code
and provide the correct SQL code

## Example of Asking myself: step-by-
step reasoning

INPUT:
- Question: How many students, on
average, does each college have
enrolled?

- Predicted SQL:
```sql
select avg(college.enr), count(*)
from college group by college.enr
```

- Schema Overview:
Table schema: There are 1 tables:
college.
the structure of table college is as
follows:
| column name | column type |
| ----- | ----- |
| ""cName"" | text |
| state | text |
| enr | number |
""cName"" is the primary key.

OUTPUT:
---
<think>
1. **Did I use the correct table for
the query?**
- Yes, the `College` table
contains the `enr` column which

```

```

1606 represents the enrollment number, so
1607 it is the correct table to use for
1608 calculating the average enrollment.
1609 2. **Did I correctly specify the
1610 column to calculate the average?**
1611 - Yes, using `avg(college.enr)`
1612 is appropriate as we want to find
1613 the average number of students
1614 enrolled in each college, and `enr`
1615 is the column that stores the
1616 enrollment data.
1617 3. **Did I include any unnecessary
1618 columns or calculations?**
1619 - The predicted SQL includes `
1620 count(*)`, which is not relevant to
1621 the question of finding the average
1622 enrollment per college. The question
1623 only asks for the average
1624 enrollment, so this `count(*)`
1625 should be removed.
1626 4. **Did I use the correct grouping
1627 ?**
1628 - The `group by college.enr` in
1629 the predicted SQL is incorrect.
1630 Grouping by the enrollment value
1631 doesn't make sense for this problem.
1632 We just need to calculate the
1633 average of all the enrollment values
1634 in the `College` table, so no
1635 grouping is required.
1636 5. **Have I ensured that my query
1637 accurately targets the required data
1638 without adding unnecessary
1639 complexity?**
1640 - The original query adds
1641 unnecessary complexity with the `
1642 count(*)` and the incorrect `group
1643 by` clause. The query should be
1644 simplified to just calculate the
1645 average enrollment.
1646 </think>
1647 <result>False</result>
1648 <correctionSQL>
1649 ```sql
1650 SELECT AVG(college.enr) FROM college
1651 ```
1652 </correctionSQL>
1653 ```
1654
1655
1656 ## Output Format:
1657 ```
1658 <think> Your step-by-step
1659 process of thinking and reasoning
1660 </think>
1661 <result>If mistakes are found
1662 during your thinking and reasoning
1663 process, give False; otherwise, give
1664 True </result>
1665 <correctionSQL> A valid SQL
1666 which is either corrected SQL or the
1667 predicted SQL without any change if
1668 you think it is correct already </
1669 correctionSQL>
1670 ```

```