Reflective Reasoning for SQL Generation

Anonymous ACL submission

Abstract

Robust text-to-SQL over complex, real-world databases remains brittle even with modern LLMs: iterative refinement often introduces syntactic and semantic drift, corrections tend to be non-transferable across queries, and naive use of large context windows scales poorly. We propose a controlled text-to-SQL framework built around *reflective refinement*. Instead of repeatedly rewriting the current SQL instance, the system decomposes generation into typed stages and applies feedback as persistent updates to the *stage-level generation mechanism*. A Reflection-Refinement Loop localizes violations to the responsible stage maximize preservation of previously validated constraints and support monotonic improvement over a query set. The method operates without gold SQL by combining interpreter-based checks with LLM-based semantic coverage verification as epistemic judges. Experiments on Spider and BIRD demonstrate consistent gains over strong prompting baselines, robust convergence within a small refinement budget, and improved execution accuracy across both frontier and open-weight model families.

1 Introduction

Large language models (LLMs) have substantially advanced natural-language interfaces to databases, yet robust text-to-SQL generation over complex, real-world schemas remains unsolved (Yu et al., 2019; Wang et al., 2021). Even frontier models frequently fail under realistic database heterogeneity: iterative correction introduces new inconsistencies, schema grounding remains brittle, and scaling via larger context windows is not sustainable.

Recent work has increasingly shifted from single-shot semantic parsing toward *LLM-driven, multi-stage* text-to-SQL pipelines that decompose the task and incorporate refinement. Decomposition-based prompting improves robustness by separating schema interpretation, predicate grounding, and SQL construction (Pourreza and Rafiei, 2023a), while execution-guided refinement leverages runtime feedback to diagnose and repair errors (Mao et al., 2024). More broadly, agentic reasoning patterns have popularized iterative refinement as a general strategy (Yao et al., 2023).

However, most existing approaches refine text-to-SQL by repeatedly *rewriting the current SQL instance*. As a result, corrections remain local, brittle, and often non-transferrable: fixing one query does not improve the generation mechanism for the next (Wang et al., 2023). We argue that this is a fundamental limitation of instance-level refinement. Instead, refinement should update the *stage-level generators* that produce SQL, enabling persistent improvement across a query set.

Why is text-to-SQL intrinsically hard? Three failure modes make iterative text-to-SQL particularly brittle:

Challenge 1: Drift under refinement. Iterative correction often introduces syntactic or semantic drift. Later rewrites violate previously satisfied constraints, producing new join or predicate inconsistencies (Wang et al., 2023; Yao et al., 2023).

Challenge 2: No persistent generalization. Most systems solve queries independently, without accumulating transferable corrections. Thus, refinement does not yield monotonic improvement across queries within the same database.

Challenge 3: Uncontrolled grounding and context scaling. Text-to-SQL fundamentally requires mapping natural-language referents onto heterogeneous schemas and value domains. Naively expanding context windows to include schema and instance evidence increases hallucination risk and becomes infeasible for large databases.

REFLECT-SQL: Reflective refinement over localized feedback. To address these challenges, we propose REFLECT-SQL, a controlled text-to-SQL framework that integrates:

Feedback localization. Addressing *Challenge 1*, we decompose generation into typed stages (schema grounding, predicate grounding, SQL realization, see Figure 1 (a)) and localize violations to the responsible component rather than regenerating monolithically.

Reflective refinement. Addressing *Challenge 2*, instead of rewriting SQL instances, we update the *parameters of the stage generators* (e.g., prompt constraints) using critic diagnostics. This enables targeted correction while preserving previously validated semantics, yielding monotonic improvement across a query set.

Database context proxies. We construct a compact schema-value proxy (Figure 1 (c)) that disentangles structural information from sampled value evidence, reducing reliance on large context windows and improving controlled grounding.

Together, these components form a self-correcting reflective loop that improves text-to-SQL generation without requiring gold SQL supervision. Progress is assessed via unsupervised epistemic judges combining interpreter-based execution checks with intent-level semantic verification. Experiments on Spider and BIRD demonstrate consistent gains, robust convergence within a small refinement budget, and fine-grained error attribution across both proprietary and open-weight model families.

Contributions. This work makes three main contributions:

- We introduce **reflective refinement**: a mechanism-level alternative to instance-level SQL rewriting that yields transferable improvements across queries.
- We formalize controlled multi-stage text-to-SQL as constraint satisfaction with localized feedback and selective restarts.
- We provide extensive empirical analysis on Spider and BIRD, demonstrating strong execution gains, convergence behavior, and component-level error attribution.

Section 2 introduces our formalization, Sections 3–?? describe REFLECT-SQL, and Sections 4–5 report experimental results and analysis. Section 7 concludes. All supplementary material can be found in the Appendix.

To support reproducibility, we release the full codebase, prompts, experimental configurations, and evaluation scripts used in this work, together with instructions for reproducing all reported results.[1]

2 Preliminaries

Problem definition. Let $\mathcal{DB}$ be a relational database. Given a natural-language query q, the text-to-SQL task is to produce an executable SQL query Q over $\mathcal{DB}$ whose execution semantics matches the intent expressed by q. We model this as a mapping

$$\mu : \mathcal{Q} \times \mathcal{DB} \to \mathcal{L}_{\text{SQL}}(\mathcal{DB}), \qquad (q, \mathcal{DB}) \mapsto Q, \tag{1}$$

where $\mathcal{Q}$ is the space of natural-language queries and $\mathcal{L}_{\text{SQL}}(\mathcal{DB})$ denotes the set of executable SQL queries over $\mathcal{DB}$. Correctness is semantic (denotational): with execution semantics $[\![\cdot]\!]_{\mathcal{DB}}$,

$$[\![\mu(q, \mathcal{DB})]\!]_{\mathcal{DB}} \;\approx\; \text{Ans}(q, \mathcal{DB}), \tag{2}$$

where $\text{Ans}(q, \mathcal{DB})$ denotes the intended answer implied by q, and $\approx$ abstracts away from surface-form differences.

Compositional generation. Rather than inducing μ as a monolithic synthesizer, we represent it as a composition of typed generative stages $\{\pi_t\}_{t=1}^{T}$, where each $\pi_t : \mathcal{X}_{t-1} \to \mathcal{X}_t$ maps a structured intermediate representation to the next:

$$\mu \triangleq \pi_T \circ \pi_{T-1} \circ \cdots \circ \pi_1. \tag{3}$$

Localized stages. In our framework, stages correspond to disjoint abstraction layers of the text-to-SQL problem: schema grounding π^{SCH} (tables, attributes, joins), value grounding π^{VAL} (canonical literals and domains), predicate construction π^{PRED} (filters and operators), aggregation/projection π^{AGG}, and final SQL realization π^{SQL} under the constraints of $\mathcal{L}_{\text{SQL}}$.

3 Reflective refinement

Overview. Figure 1 illustrates the overall architecture. Given a natural language question q and a database $\mathcal{DB}$, the system first produces an initial SQL hypothesis through a sequence of structured

[1] Link omitted for anonymity.

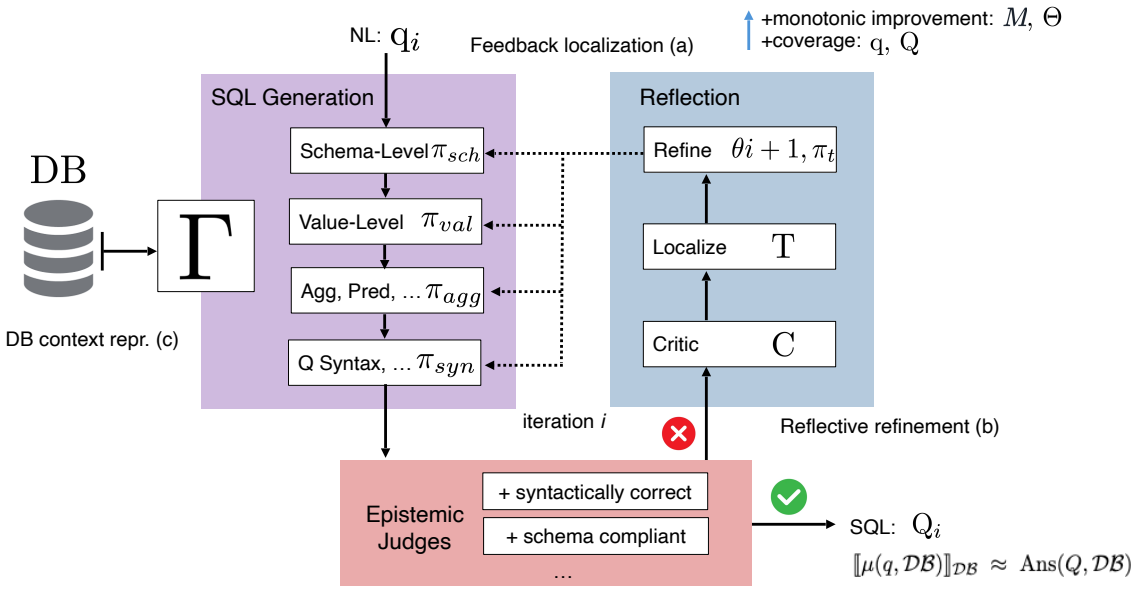

Figure 1: High-level overview of REFLECT-SQL, showing staged generation, critic feedback, semantic checking, and selective reflective refinement.

generation stages. The resulting query is then verified using execution-based and semantic consistency checks. If violations are detected, the system enters a bounded reflective loop that (i) diagnoses the failure, (ii) localizes it to a specific abstraction stage, and (iii) selectively refines only the responsible component while preserving previously validated constraints. The process terminates when all checks pass or when a fixed refinement budget is reached. A procedural specification is provided in Appendix 1.

Stage-structured generation. We represent text-to-SQL synthesis as a composition of typed generative stages $\{\pi_t\}_{t\in\mathcal{T}}$, where each stage operates over a structured intermediate representation. Each π_t is parameterized by a stage-level constraint specification θ_t (e.g., a prompt template), and the full generator is denoted by

$$\mu_\Theta \triangleq \pi_{T,\theta_T} \circ \cdots \circ \pi_{1,\theta_1}, \qquad \Theta = \{\theta_t\}_{t\in\mathcal{T}}. \quad (4)$$

Unlike standard refinement approaches that repeatedly rewrite the SQL instance itself, REFLECT-SQL applies refinement at the level of these stage mechanisms.

Reflective refinement loop. Given an input query q, the system performs a bounded sequence of refinement iterations indexed by $k = 0, 1, \ldots, K$. At iteration k, the system maintains a current set of stage constraints $\Theta^{(k)}$. Each cycle proceeds as follows:

1. **Generate.** Produce a candidate SQL query:

$$Q^{(k)} \leftarrow \mu_{\Theta^{(k)}}(q, \mathcal{DB}). \quad (5)$$

2. **Evaluate.** Compute diagnostic feedback using an unsupervised proxy evaluator:

$$r^{(k)} \leftarrow \mathrm{eval}(Q^{(k)}, q, \mathcal{DB}), \quad (6)$$

where eval combines (i) interpreter-based execution and syntax checks, and (ii) an intent-preservation judge that verifies semantic consistency between q and $Q^{(k)}$.

3. **Critique and localize.** A critic analyzes the diagnostic signal and returns a localized violation:

$$(t, c, Q') \leftarrow f_{\text{CRITIC}}(r^{(k)}, \Gamma, Q^{(k)}), \quad (7)$$

where $t \in \mathcal{T}$ denotes the abstraction type responsible for the error (e.g., schema grounding, predicate construction), c is a criterion label, and $Q' \subseteq Q^{(k)}$ is the implicated clause or span.

4. **Selective refinement.** Only the constraint parameters of the responsible stage are updated:

$$\theta_t^{(k+1)} \leftarrow \mathrm{Reflect}\big(\theta_t^{(k)}, r^{(k)}, c\big), \quad (8)$$

while all other stages remain fixed:

$$\theta_{t'}^{(k+1)} = \theta_{t'}^{(k)} \qquad \forall\, t' \neq t. \quad (9)$$

The refinement loop terminates early if eval confirms that all structural and semantic constraints are satisfied, and otherwise stops after reaching the maximum refinement budget K.

Mechanism-level refinement. The essential distinction from prior iterative correction methods is that edits are applied to the stage mechanism parameters θ_t (e.g., prompt-level constraints), rather than directly rewriting the SQL instance $Q^{(k)}$. This enables targeted correction without destabilizing previously validated components, and supports transferable improvements across queries within the same database abstraction.

Critic criteria. The critic implements a structured inspection over abstraction types $\mathcal{T}$ using the database context proxy Γ (Section **??**). Violation categories include: (i) schema-level errors (invalid tables, columns, or joins), (ii) value grounding errors (incorrect literals or formatting), (iii) predicate and aggregation mismatches, and (iv) syntactic or execution failures. The critic returns localized feedback sufficient to selectively re-invoke only the implicated stage. Prompt templates and criteria definitions are provided in Appendix D.5.

Unsupervised evaluative proxies. Because no gold SQL supervision is assumed, refinement progress is defined entirely through eval. In our setting, eval instantiates two complementary judges: (i) an interpreter-based validator for syntactic correctness and executability, and (ii) an LLM-based semantic judge for intent preservation between the natural-language query and the generated SQL. Together, these proxies provide structured feedback signals that enable refinement without access to reference queries.

Scope of refinement. All refinement updates are applied only within the bounded trajectory of a single query instance. Stage constraints are reset between independent inputs, ensuring that REFLECT-SQL does not perform dataset-level learning or cross-example adaptation at inference time.

4 Experiments

We evaluate REFLECT-SQL on two multi-domain text-to-SQL benchmarks, Spider 1.0 and BIRD, to test (i) end-to-end execution accuracy, (ii) convergence under iterative refinement, and (iii) the contribution of localized feedback and semantic coverage checking.

4.1 Setup

Benchmarks. We use the official development splits of Spider 1.0 and BIRD. The Spider 1.0 dev set contains 1,034 questions spanning 20 databases drawn from the benchmark's pool of 200 cross-domain schemas, emphasizing compositional generalization over heterogeneous structures. BIRD consists of 12,751 total pairs across 95 databases, and adds more challenging SQL patterns while evaluating both query validity and efficiency, providing a complementary stress test for complex relational reasoning.

Baselines. We compare against strong GPT-4 based prompting systems: DIN-SQL (Pourreza and Rafiei, 2023b), DAIL-SQL (Gao et al., 2023), MAC-SQL (Wang et al., 2025a), and MCS-SQL (Lee et al., 2024), as well as a GPT-4 zero-shot baseline (OpenAI et al., 2024). Baseline scores are taken directly from the original publications under their reported GPT-4 configurations, since exact re-execution is sensitive to API version drift and model snapshot availability.

Models and decoding. We instantiate REFLECT-SQL with a representative set of foundation models spanning both proprietary and open-weight regimes. For OpenAI models, we use an Azure-hosted GPT-4o endpoint (`2024-12-01-preview`, not GPT-4o-mini). Additional proprietary models are accessed via their official APIs, including `claude-sonnet-4-5` and `gemini-2.5-pro`. These models serve as strong reference points for state-of-the-art general-purpose reasoning under controlled prompting.

In parallel, we evaluate competitive open-weight models in the 30B parameter range (e.g., Qwen3-30B and DeepSeek-R1-32B), which offer a favorable trade-off between capacity and accessibility and are commonly used in recent text-to-SQL systems. To test whether the proposed mechanism relies on scale or persists under constrained capacity, we additionally run targeted ablations using a substantially smaller open-weight model (LLaMA 3.1 8B).

Unless noted otherwise, all stages use temperature 0 decoding wherever supported to minimize variance across runs. Fixed token budgets are applied per stage to ensure comparability across models.

Refinement budget. We allow up to T refinement steps per example, with early stopping when

Method	BIRD		Spider
	VES	EX	EX
GPT-4 (zero-shot)	49.8	46.4	74.6
DIN-SQL	58.8	50.7	82.8
DAIL-SQL	56.1	54.8	84.4
MAC-SQL	58.8	57.7	86.8
MCS-SQL	64.8	63.4	89.5
REFLECT-SQL			
— base	58.5	54.6	81.7
— GPT-4o	66.5	65.2	93.8
— GPT-5	**68.9**	**67.8**	**95.4**
— Claude Sonnet	67.4	66.0	94.6
— Gemini Pro	66.0	64.5	94.0
— Qwen3-30B	61.2	59.9	92.1
— DeepSeek-R1-32B	63.7	62.1	94.3
— LLaMA-3.3-70B	64.9	63.5	94.8
— DeepSeek-Coder-33B	62.0	60.8	93.2
— DeepSeek-R1-Distill-70B	65.5	64.0	95.0

Table 1: **REFLECT-SQL consistently outperforms prior prompting and agentic baselines, demonstrating the benefit of reflective refinement beyond staged decomposition alone.** Baselines are GPT-4 based prompting systems, while REFLECT-SQL can be instantiated with different foundation models. Results are reported on Spider 1.0 and BIRD using execution accuracy (EX) and valid efficiency score (VES).

both structural checks and semantic coverage checks pass. Unless otherwise stated, we set $T = 4$ on Spider 1.0 and $T = 5$ on BIRD, based on the observed early saturation of refinement gains. The refiner selectively restarts only the stage implicated by critic diagnostics (schema interpretation, predicate grounding, or query construction). Across benchmarks, convergence typically occurs within 3–4 refinement iterations on average.

Semantic coverage checking. Because REFLECT-SQL does not assume access to gold SQL during refinement, semantic coverage is verified via an intent-consistency judge that checks whether the generated query preserves all constraints implied by the natural-language question.

Metrics. We report execution accuracy (EX) on both benchmarks and Valid Efficiency Score (VES) on BIRD.

4.2 Main Results

Across all benchmarks, REFLECT-SQL outperforms prior prompting baselines and agentic systems. The full method, including localized feedback and reflective refinement, consistently improves over the staged base generator, indicating that targeted mechanism-level refinement adds substantial value beyond decomposition alone. Gains persist across both proprietary and open-weight model families, suggesting that the approach is not tied to a specific foundation model.

5 Analysis

5.1 Convergence under Refinement

Early refinement yields the majority of execution gains, after which performance saturates and may degrade under over-refinement. Figure 2 plots EX as a function of refinement iteration k. Across both benchmarks and all foundation models, performance improves sharply from $k = 0$ to $k = 3$, after which gains become marginal and may reverse for smaller models. All curves report mean EX over the full development set under temperature-0 decoding.

On Spider 1.0, an average improvement of ∼10–12 EX points occurs within the first three iterations. For example, GPT-5 improves from ∼85 EX at $k = 0$ to ∼95 EX by $k = 3$, and GPT-4o increases from ∼82 to ∼94 over the same range. Medium-sized open-weight models such as Qwen3-30B and DeepSeek-R1-32B show even larger relative gains when their single-shot performance is further from saturation (e.g., DeepSeek-R1-32B improves from 72.4 to 89.5 EX by $k = 3$). Smaller models (Qwen3-8B, LLaMA 3.1 8B) also benefit substantially early on (e.g., Qwen3-8B: 59.5 to 78.4 EX by $k = 3$), but exhibit higher volatility beyond this point.

Beyond $k = 3$, additional refinement yields diminishing returns and can hurt performance. For strong models (GPT-5, GPT-4o, DeepSeek-R1-32B), EX remains stable or slightly decreases after convergence, suggesting that later edits mostly introduce benign perturbations. In contrast, weaker models show clear signs of over-refinement: Qwen3-8B and LLaMA 3.1 8B drop by 5–8 EX points between $k = 3$ and $k = 6$. Qualitatively, we observe cases where an early iteration introduces a hallucinated join or predicate; subsequent iterations then refine around this mistaken hypothesis rather than reverting it, causing the system to "lock in" to an incorrect execution plan.

The behavior on BIRD mirrors these trends but with lower absolute scores and a slightly delayed saturation point. All models start substantially below their Spider 1.0 counterparts at $k = 0$ (e.g., GPT-5 at 56.2 EX and GPT-4o at 54.6 EX), re-

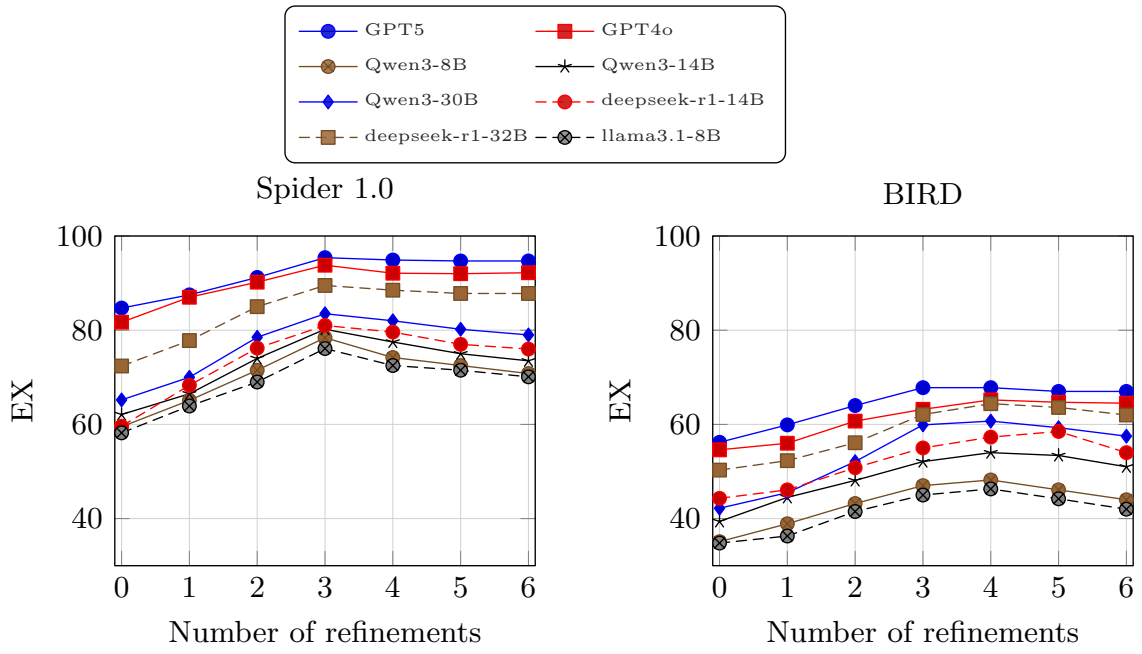

Figure 2: **Most gains emerge within the first few refinement iterations, while excessive iterations yield diminishing returns or over-refinement.** Execution accuracy (EX) is reported at each refinement iteration k for a range of base models on Spider 1.0 (dev) and BIRD (dev).

flecting the higher schema complexity and SQL difficulty of BIRD. However, the first three to four refinement steps again provide substantial improvements: GPT-5 reaches 67.8 EX by $k = 3$ and remains stable thereafter, while GPT-4o climbs to 65.2 EX by $k = 4$. Smaller models gain 10–15 EX points up to $k = 3$, but degrade noticeably for $k \geq 5$, again consistent with over-refinement around brittle intermediate beliefs.

Overall, these curves suggest that (i) a small refinement budget ($k \approx 3$ on Spider 1.0 and $k \approx 3$–4 on BIRD) captures most available gains, (ii) increasing the iteration budget beyond this range offers little benefit for strong models and can harm weaker ones, and (iii) reflective refinement systematically narrows, but does not erase, the gap between small and frontier-scale models. In practice, this motivates using a moderate, benchmark-specific iteration cap rather than running refinement to exhaustion.

5.2 Feedback Granularity

Highly localized, categorical feedback improves execution accuracy with only a modest increase in refinement iterations. Table 2 compares three feedback styles for a fixed base model (GPT-4o). Coarse feedback provides only a binary pass/fail signal with minimal localization. Granular feedback returns stage-typed categorical violations with localized spans (e.g., incorrect joins, missing predicates, aggregation mismatches). Epistemic-only feedback focuses on intent-level assumption mismatches (e.g., misinterpreted entity relations) without providing fully structural correction hints.

Feedback Style	BIRD (EX)	Spider (EX)	Avg. Iters (Spider)
Coarse	61.0	90.7	2.3
Granular	**65.2**	**93.8**	**3.1**
Epistemic-only	64.7	92.5	2.6

Table 2: **Localized, categorical feedback enables more precise corrections and higher execution accuracy than coarse or epistemic-only feedback.** Comparison of feedback styles for a fixed base model (GPT-4o) on Spider 1.0 and BIRD. Average refinement iterations are reported on Spider 1.0; trends are similar on BIRD.

Coarse feedback already yields strong performance (61.0 EX on BIRD and 90.7 EX on Spider 1.0) with 2.3 refinement steps on average. Granular feedback leads to consistent improvements on both benchmarks, reaching 65.2 and 93.8 EX respectively, at the cost of slightly more iterations (3.1 on average). Epistemic-only feedback sits between these extremes, achieving 64.7 and 92.5 EX with 2.6 refinement steps.

Configuration	VES	EX	ΔVES	ΔEX
Full method	**66.5**	**65.2**		
No critic (execution-only)	62.4	61.6	−4.1	−3.6
No semantic checker	63.1	58.2	−3.4	−7.0
Single-shot (no stages)	58.5	54.6	−8.0	−10.6

Table 3: **Removing staged decomposition, critic diagnostics, or semantic intent verification substantially reduces accuracy.** Component ablations on BIRD. Δ columns report changes relative to the full system.

Overall, localized categorical feedback enables more reliable correction by pinpointing which semantic constraints must be revised, rather than triggering global regeneration.

5.3 Component Contributions

Performance gains arise from the interaction between staged decomposition, localized refinement, and intent-level verification. To isolate where improvements originate, we ablate major components (Table 3). Removing the critic reduces both VES and EX, indicating that execution-grounded diagnostics are critical for targeted correction. Disabling the semantic checker causes an even larger degradation (−7.0 EX), highlighting the importance of intent-level verification beyond syntactic validity. Importantly, the semantic checker does not access gold SQL, but only evaluates consistency between the natural-language intent and the generated query. Finally, collapsing the system into a single-shot generator without staged constraints yields the largest performance loss, confirming that decomposition is necessary but insufficient on its own.

Taken together, reliable improvement emerges from the combination of constraint-emitting stages and selective reflective refinement, rather than from any single component in isolation.

5.4 Qualitative Cases and Benchmark Limitations

Exact-match evaluation may penalize alternative-but-valid SQL realizations, motivating complementary intent-level verification. Exact-match evaluation conflates true semantic failures with (i) alternative SQL forms that yield equivalent denotations and (ii) underspecified reference annotations that admit multiple valid interpretations. We observe three recurring patterns. First, **schema/join ambiguity**: multiple join paths can yield equivalent answers, but benchmarks often provide a single canonical query. Second, **boundary and literal errors**: failures are frequently localized to operator choice or literal grounding (e.g., $>$ vs. $\geq$), which our staged design can attribute to predicate grounding. Third, **over-/under-specification**: model outputs may include plausible constraints absent from the reference SQL (or omit constraints that the reference implicitly assumes), exposing mismatches between benchmark minimalism and practical intent interpretation.

While we continue to report standard EX for comparability, these observations motivate semantic coverage checks that evaluate whether required constraints are present independent of surface form.

Summary. Overall, REFLECT-SQL converges within a small refinement budget, benefits strongly from localized categorical feedback, and derives gains from the interaction between staged decomposition, critic diagnostics, and semantic intent verification.

6 Related Work

Schema linking and grounding. Aligning natural-language mentions to schema structure and value evidence remains a core bottleneck in text-to-SQL. Classical systems such as RAT-SQL highlight the importance of relation-aware schema encoding (Wang et al., 2021). Recent work scales schema linking to large multi-database settings (LinkAlign) (Wang et al., 2025b), while contextual schema-link graphs improve grounding and in-context demonstration selection (Lee et al., 2025). Content-aware prompting and value-grounded rewriting further reduce ambiguity prior to generation (Mao et al., 2024).

Decomposition, planning, and structured prompting. Many recent systems improve reliability by decomposing SQL synthesis into intermediate reasoning steps. DIN-SQL operationalizes decomposed in-context learning with iterative correction (Pourreza and Rafiei, 2023a), while PTD-SQL partitions by query type for targeted refinement (Luo et al., 2024). ROUTE explores robust multi-task collaboration to stabilize cross-subtask generalization (Qin et al., 2025). These approaches demonstrate the benefits of staged reasoning, but refinement is typically applied at the level of the current SQL instance.

Execution-guided verification and agentic refinement. Execution feedback has become central for

improving correctness. DART-SQL repairs candidates using execution-guided refinement (Mao et al., 2024), and ExeSQL bootstraps self-training signals via execution validation across dialects (Zhang et al., 2025). Multi-candidate reasoning with preference-based selection further improves final execution accuracy (Pourreza et al., 2025). However, these methods primarily refine by regenerating or repairing query instances rather than updating the underlying generation mechanism.

Data synthesis and fine-tuning for open models. Closing the gap between open-weight and proprietary models often relies on synthetic supervision and targeted tuning. DTS-SQL applies staged decomposition for smaller LLMs (Pourreza and Rafiei, 2024), and weak-to-strong synthesis pipelines provide scalable training data for instruction tuning (Yang et al., 2024).

Positioning of this work. Across these threads, prior systems improve text-to-SQL by strengthening grounding, decomposing generation, and applying feedback through *instance-level SQL repair* or additional supervision. In contrast, REFLECT-SQL treats refinement as an update to the *stage-level generation mechanism* (typed generators with prompt-level constraints), driven by localized critic diagnostics and unsupervised execution/semantic judges. This enables corrections to transfer across queries within a database abstraction, rather than remaining brittle per-instance edits.

7 Conclusion

This work reframes iterative improvement in text-to-SQL as *reflective* refinement mechanism, rather than instance-level SQL rewriting. Starting from the three core failure modes of current LLM-guided pipelines—drift under refinement, lack of transferable generalization, and poor scalability of schema/value grounding—we introduced a controlled, staged generation process coupled to a Reflect–Refine Loop that localizes violations. The central idea of *reflective refinement* is to update the parameters of the stage generators (e.g., prompt constraints) using localized diagnostic context, while maximizing backward preservation of already-validated constraints, enabling monotonic coverage improvements over a query set within a database. Complementarily, we proposed a compact, reusable database context proxy that disentangles schema structure from sampled value/domain evidence, reducing context-window dependence and improving controlled grounding.

Within the scope of the presented design, the resulting system yields a self-correcting loop that is both more controllable and more reusable than monolithic retry strategies: refinement targets the smallest responsible abstraction layer, and progress is assessed via unsupervised epistemic judges combining interpreter-based syntactic/execution checks with intent-level semantic coverage verification. This combined mechanism provides a practical pathway to robust text-to-SQL behavior without assuming gold SQL supervision, while remaining compatible with diverse foundation models. Beyond text-to-SQL, the mechanism suggests a general template for structured prediction in which iterative improvement is driven by localized diagnostics and constraint-preserving updates to the underlying generative operators.

Limitations

The proposed approach is bounded by a fixed refinement budget and returns the final SQL candidate even if issues persist once this budget is exhausted, which may leave some difficult cases unresolved. Its effectiveness depends on the quality of the Critic and Analyzer components: inaccurate or coarse diagnostics misattribute failures to the wrong abstraction layer, leading to inefficient refinement or missed corrections. Moreover, the current semantic checker enforces a predefined set of correctness constraints (e.g., schema consistency, join feasibility, aggregation semantics) and may fail to detect subtler semantic mismatches that remain executable but deviate from the user's intent. While the framework is intentionally designed around semantic faithfulness and controllability, it does not explicitly reason about query efficiency or execution cost. Incorporating efficiency-oriented epistemic judges—e.g., based on query plans or execution statistics—remains an important direction for future work. Finally, although demonstrated here for NL-to-SQL, the framework is general. Applying it to other structured prediction tasks (e.g., text-to-SPARQL) would require task-specific instantiations of generators and evaluators, rather than changes to the core reflective refinement mechanism.

References

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023.

Text-to-sql empowered by large language models: A benchmark evaluation. *Preprint*, arXiv:2308.15363.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2024. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation. *Preprint*, arXiv:2405.07467.

Jihyung Lee, Jin-Seop Lee, Jaehoon Lee, YunSeok Choi, and Jee-Hyong Lee. 2025. DCG-SQL: Enhancing in-context learning for text-to-SQL with deep contextual schema link graph. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15397–15412, Vienna, Austria. Association for Computational Linguistics.

Ruilin Luo, Liyuan Wang, Binghuai Lin, Zicheng Lin, and Yujiu Yang. 2024. PTD-SQL: Partitioning and targeted drilling with LLMs in text-to-SQL. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3767–3799, Miami, Florida, USA. Association for Computational Linguistics.

Wenxin Mao, Ruiqi Wang, Jiyu Guo, Jichuan Zeng, Cuiyun Gao, Peiyi Han, and Chuanyi Liu. 2024. Enhancing text-to-SQL parsing through question rewriting and execution-guided refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2009–2024, Bangkok, Thailand. Association for Computational Linguistics.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

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

Mohammadreza Pourreza and Davood Rafiei. 2023a. DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Mohammadreza Pourreza and Davood Rafiei. 2023b. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Preprint*, arXiv:2304.11015.

Mohammadreza Pourreza and Davood Rafiei. 2024. DTS-SQL: Decomposed text-to-SQL with small large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 8212–8220, Miami, Florida, USA. Association for Computational Linguistics.

Yang Qin, Chao Chen, Zhihang Fu, Ze Chen, Dezhong Peng, Peng Hu, and Jieping Ye. 2025. ROUTE: Robust multitask tuning and collaboration for text-to-SQL. In *International Conference on Learning Representations (ICLR)*.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2021. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *Preprint*, arXiv:1911.04942.

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

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. *Preprint*, arXiv:2203.11171.

Yihan Wang, Peiyu Liu, and Xin Yang. 2025b. LinkAlign: Scalable schema linking for real-world large-scale multi-database text-to-SQL. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 977–991, Suzhou, China. Association for Computational Linguistics.

Jiaxi Yang, Binyuan Hui, Min Yang, Jian Yang, Junyang Lin, and Chang Zhou. 2024. Synthesizing text-to-SQL data from weak and strong LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7864–7875, Bangkok, Thailand. Association for Computational Linguistics.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2019. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Preprint*, arXiv:1809.08887.

Jipeng Zhang, Haolin Yang, Kehao Miao, Ruiyuan Zhang, Renjie Pi, Jiahui Gao, and Xiaofang Zhou. 2025. ExeSQL: Self-taught text-to-SQL models with execution-driven bootstrapping for SQL dialects. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 24305–24326, Suzhou, China. Association for Computational Linguistics.

A Error Analysis

This appendix provides a qualitative error analysis intended to contextualize the quantitative results reported in the main paper. The examples in Table 4 illustrate representative model behaviors across different query complexities, including true semantic errors, alternative-valid predictions, and cases where the benchmark gold annotation is incorrect or underspecified. We include both correct and incorrect predictions to highlight how our system's stage-wise design affects failure modes, particularly in the presence of aggregation, joins, and schema ambiguity.

Each row reports the natural-language question, the gold SQL annotation, the model's predicted SQL, a coarse complexity category, and the final outcome label. Outcome labels follow our evaluation protocol: *Alternative-valid* denotes queries that differ syntactically from the gold SQL but are semantically equivalent; *Gold wrong* indicates errors in the reference annotation rather than the model output. This analysis is not exhaustive, but serves to illustrate common error patterns and limitations of both the model and the benchmark.

Question	**Gold SQL**	**Predicted SQL**	**Complexity**	**Outcome**
List the names of students enrolled in courses taught by Professor Smith	`SELECT s.name FROM student s JOIN enrollment e JOIN course c JOIN professor p WHERE p.name = 'Smith'`	`SELECT s.name FROM student s JOIN enrollment e JOIN teaching t JOIN professor p WHERE p.name = 'Smith'`	Medium	Alternative-valid
Find the number of employees hired after 2019	`SELECT COUNT(*) FROM employee WHERE hire_year > 2019`	`SELECT COUNT(*) FROM employee WHERE hire_year >= 2019`	Easy	True error
What are the titles of movies directed by Christopher Nolan?	`SELECT m.title FROM movie m JOIN director d WHERE d.name = 'Christopher Nolan'`	`SELECT m.title FROM movie m JOIN director d WHERE d.full_name = 'Christopher Nolan'`	Easy	Alternative-valid
List departments with more than five employees	`SELECT d.name FROM dept d JOIN emp e GROUP BY d.id HAVING COUNT(*) > 5`	`SELECT d.name FROM dept d JOIN emp e GROUP BY d.id, d.name HAVING COUNT(e.id) > 5`	Medium	Alternative-valid
Find the average salary of employees in the sales department	`SELECT AVG(e.salary) FROM emp e`	`SELECT AVG(e.salary) FROM emp e JOIN dept d WHERE d.name = 'Sales'`	Easy	Gold wrong

Table 4: **Qualitative examples and benchmark limitations.** For each case, we show the question, gold SQL, predicted SQL, complexity category, and outcome.

B Illustrative Example: Selective Reflective Refinement

Figure 3 visualizes a representative inference episode in the proposed controlled NL→SQL framework. The pipeline first decomposes the natural-language question into schema-level grounding and predicate-level constraint signals, which are then realized as an executable SQL query. In the first iteration, the generated query fails to preserve the intended semantics due to an inverted range predicate and a missing grouping constraint under aggregation.

Using interpreter-based checks and LLM-based semantic evaluation, the critic attributes the error to the SQL realization component. Rather than regenerating the query instance, the system performs a selective reflective update to the parameters of the realization generator, strengthening realization-level constraints while preserving previously validated schema and predicate decisions. Re-invoking only the affected component yields a corrected SQL query that satisfies both syntactic correctness and semantic intent preservation.

C Algorithm

This appendix provides pseudocode for the controlled, compositional text-to-SQL inference procedure described in the main paper. The Algorithm 1 formalizes the interaction between the typed query generation components corresponding to distinct abstraction layers (e.g., schema-level, predicate-level, aggregation-level, and SQL realization), and the selective refinement loop applied during inference.

The algorithm distinguishes between an intial compositional generation pass, instantiating the mapping $\mu_\Theta = \pi_{|\mathcal{T}|} \circ \cdots \circ \pi_1$ and a refinement loop in which only implicated generator components are selectively re-invoked. Refinement is guided by a Critic that produces localized violation diagnostics and attributes errors to a specific abstraction type, enabling targeted updates to the parameters of the corresponding generator while preserving previously validated constraints.

The inference process terminates early when both syntactic correctness and semantic intent preservation are satisfied, as determined by unsupervised epistemic judges, or after a fixed refinement budget is reached, in which case the best-effort executable SQL query is returned.

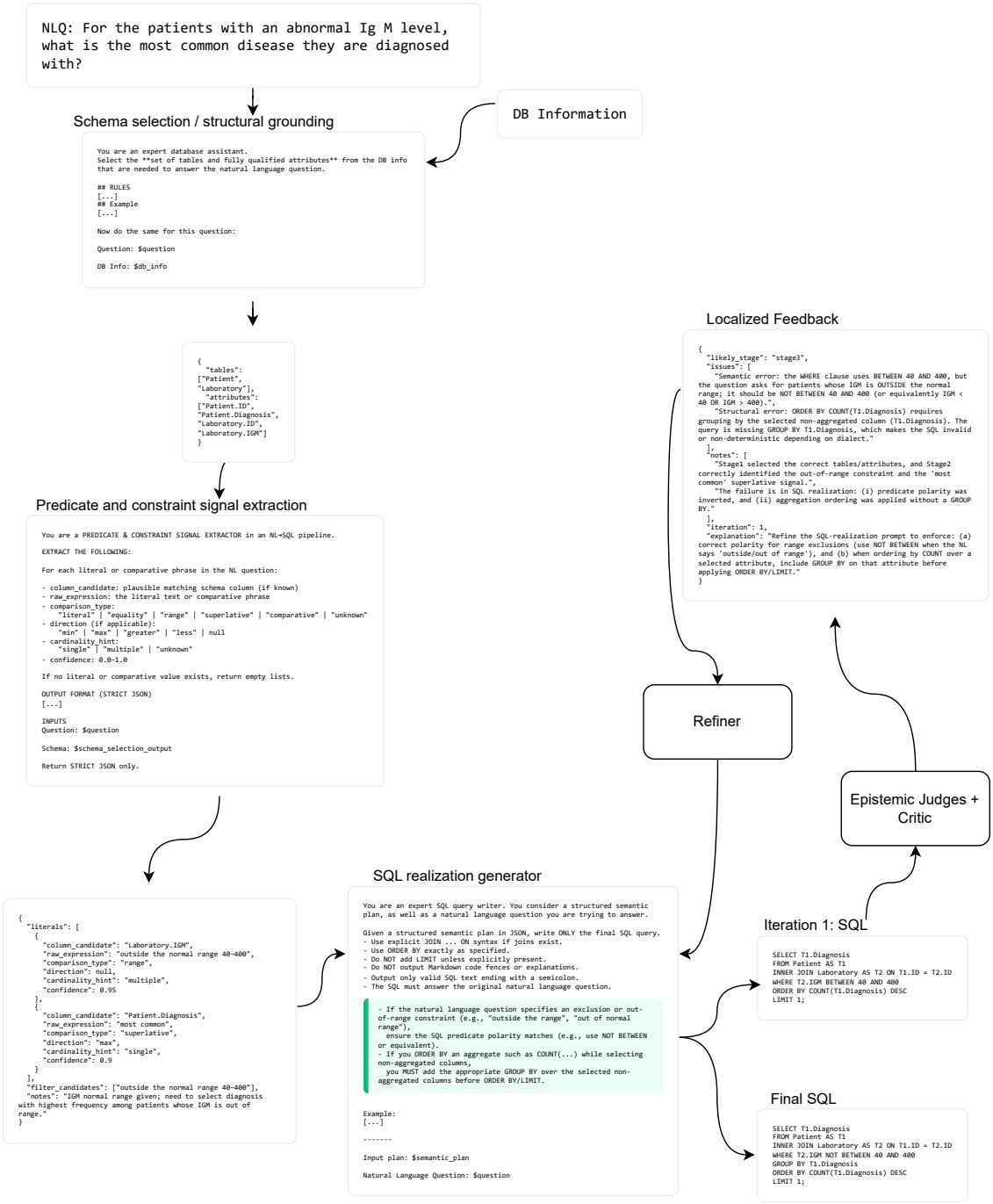

Figure 3: **Illustrative example of selective reflective refinement in the text-to-QL pipeline.** The diagram shows an end-to-end inference trace for a query involving an out-of-range laboratory constraint and a superlative aggregation. An initial compositional pass produces an incorrect SQL realization. Epistemic judges and a critic localize the failure to the SQL realization component, triggering a targeted refinement of its generation parameters. The refined realization enforces correct predicate polarity and aggregation grouping, yielding a syntactically and semantically correct final query.

Algorithm 1 Controlled compositional text→SQL with selective reflective refinement

Require: NL question q, relational database $\mathcal{DB}$, max refinements T=3
Require: Typed generators $\{\pi_t\}_{t\in\mathcal{T}}$ with parameters $\Theta = \{\theta_t\}_{t\in\mathcal{T}}$

Initialize state $s \leftarrow (q, DB)$

▷ Initial compositional generation: $\mu_\Theta = \pi_{|\mathcal{T}|} \circ \cdots \circ \pi_1$
$s \leftarrow \pi_{\text{SCH}}(s; \theta_{\text{SCH}})$
$s \leftarrow \pi_{\text{VAL}}(s; \theta_{\text{VAL}})$
$s \leftarrow \pi_{\text{AGG}}(s; \theta_{\text{AGG}})$
$s \leftarrow \pi_{\text{PRED}}(s; \theta_{\text{PRED}})$
$\widehat{Q} \leftarrow \pi_{\text{SQL}}(s; \theta_{\text{SQL}})$

for $i \leftarrow 1$ to T **do**
▷ Unsupervised evaluation via epistemic judges (interpreter + LLM)
  $r \leftarrow eval(\widehat{Q}, DB, q)$
  syn $\leftarrow Pass_{\text{SYN}}(r)$ ▷ parses/executes under DB
  sem $\leftarrow Pass_{\text{SEM}}(r)$ ▷ intent preserved: $[[\widehat{Q}]]_{DB} \approx Ans(q, DB)$
  if syn $\wedge$ sem **then**
    return $\widehat{Q}$
  end if

▷ Critique: produce localized violation triples (t, c, Q')
  $\Gamma_{DB} \leftarrow BuildContextProxy(DB)$
  $\mathcal{V} \leftarrow f_{\text{CRITIC}}(r, \Gamma_{DB}, \widehat{Q})$
▷ $\mathcal{V} = \{(t, c, Q')\}$ where $t \in \mathcal{T}$, $c \in \mathcal{C}_t$, $Q' \subseteq \widehat{Q}$

▷ Localize to the implicated abstraction type
  $t^\star \leftarrow \ell(\mathcal{V})$ ▷ $t^\star \in \mathcal{T}$

▷ Reflect-refine: update only the parameters of the implicated generator
  $\theta_{t^\star} \leftarrow Reflect(\theta_{t^\star}; \mathcal{V}, r)$
▷ Maximize backward preservation: for all $t \neq t^\star$, keep θ_t unchanged

▷ Selective restart: re-run only the implicated component (and downstream realization)
  $s \leftarrow \pi_{t^\star}(s; \theta_{t^\star})$
  $\widehat{Q} \leftarrow \pi_{\text{SQL}}(s; \theta_{\text{SQL}})$
end for
return $\widehat{Q}$ ▷ budget exhausted: return best-effort SQL

D Prompts

D.1 Stage 1: Table and Attribute Selection Prompt

Purpose. This prompt performs schema-level interpretation by selecting the minimal set of tables and fully qualified attributes required to answer the input question. It emits schema constraints that determine the permissible join structure and attribute footprint for downstream reasoning stages.

Stage 1 — Table and Attribute Selection (SYSTEM PROMPT)

Role. You are an **expert database assistant** responsible for *schema-level interpretation*. Given a natural-language question and a database schema, select the *minimal* set of tables and *fully qualified attributes* required to express a correct SQL query.

Objectives

1. Identify the minimal set of relevant tables needed to answer the question. [Schema Interpretation]
2. Select only the attributes required to express joins, filters, grouping, or outputs. [Attribute Footprint Minimization]
3. Emit explicit schema constraints to guide downstream reasoning stages. [Structural Constraint Inference]

Critical Requirements

- **Attribute qualification.** All attributes *must* be fully qualified using the format `table.attribute`. [Deterministic Naming Constraint]
- **Primary key inclusion.** If a table is selected, *always include its primary key* (attribute ending in `_id`). [Join Anchoring Constraint]
- **Foreign key consistency.** If the question implies a relationship between tables, include the corresponding foreign key attributes. [Relational Validity Constraint]
- **Minimality.** Prefer the smallest set of tables and attributes that allows the query to be expressed. [Parsimony Principle]
- **No hallucinations.** Do *not* invent tables, attributes, or joins. Only use elements explicitly present in the schema and its foreign keys. [Constraint Enforcement]
- **Reachability.** Do *not* include tables that are not reachable via declared foreign keys. [Schema Connectivity Constraint]

Special Logic: Exclusion / Difference Queries

- If the question expresses exclusion (e.g., "except", "without", "not having"), include:
  - the primary key of the main table, and
  - the relevant foreign key of the excluding table.

[Anti-Join Readiness]

Output Contract

- Return **strictly valid JSON**.
- The output must contain *exactly two keys*:
  - `"tables"`: list of selected table names
  - `"attributes"`: list of fully qualified attributes

- Do **not** include explanations, comments, or additional fields. [Structural Deduction]

Illustrative Example

```
Question:
Which constructor scored the fewest points in each year?

Schema:
{
  "tables": {
    "results": ["race_id", "driver_id", "constructor_id", "points"],
    "races": ["race_id", "year"],
    "constructors": ["constructor_id", "name"]
  },
  "foreign_keys": [
    {"from": "results.race_id", "to": "races.race_id"},
    {"from": "results.constructor_id", "to": "constructors.constructor_id"}
  ]
}

Answer:
{
  "tables": ["results", "races", "constructors"],
  "attributes": [
    "results.constructor_id",
    "results.points",
    "races.year",
    "constructors.name"
  ]
}
```

Task Invocation

```
Question:
$question

Special evidence to consider:
$extra_evidence

Schema:
$schema
```

Do not include system instructions in the output.

Table 5: System prompt for Stage 1 table and attribute selection.

D.2 Stage 2: Literal and Constraint Signal Extraction Prompt

Purpose. This prompt performs predicate-level grounding by extracting raw literal values, comparative cues, and superlative signals from the natural-language question, without committing to SQL syntax or operator selection. Its output provides structured semantic evidence that is later interpreted by the query construction stage.

Stage 2 — Literal and Constraint Signal Extraction (SYSTEM PROMPT)

Role. You are a **Literal and Constraint Signal Extractor** in an text-to-SQL pipeline. Extract natural-language values, comparison cues, and superlative/comparative signals *without* deciding SQL syntax.

Objectives

1. Surface raw constraint information (literals, comparisons, superlatives) from the question. [Signal Extraction]

2. Propose plausible schema alignment via `column_candidate` (when possible). [Schema Grounding]
3. Preserve ambiguity for downstream planning (do not decide operators/functions). [Deferred Semantic Commitment]

Critical Requirements

- **No SQL decisions.** You are **not** generating `WHERE` clauses, SQL syntax, operators, or functions. [Constraint Enforcement]
- **Extract, don't interpret.** Do not choose exact operators (e.g., `=`, `>`, `LIKE`), aggregations, or query structure. Only capture *signals*. [Separation of Concerns]
- **Completeness.** If no literals or comparative values exist, return empty lists (still valid JSON). [Total Function / Well-Formed Output]

What to Extract For each literal or comparative phrase in the natural-language question, produce an entry with:

- `column_candidate`: plausible matching schema column (if known)
- `raw_expression`: literal text or comparative phrase
- `comparison_type`: `"literal"` | `"equality"` | `"range"` | `"superlative"` | `"comparative"` | `"unknown"`
- `direction` (if applicable): `"min"` | `"max"` | `"greater"` | `"less"` | `null`
- `cardinality_hint`: `"single"` | `"multiple"` | `"unknown"`
- `confidence`: float in $[0.0, 1.0]$

Output Contract (STRICT JSON)

- Return **strictly valid JSON** with *exactly* these keys: `"literals"`, `"filter_candidates"`, `"notes"`.
- Always include all fields for each literal object.
- `"filter_candidates"` may be empty.
- `"notes"` may be `null` or a short comment.
- **Never emit SQL, operators, or functions.**

Schema Awareness Use Stage 1 outputs to guide plausible column candidates; do not reference columns/tables outside Stage 1 unless explicitly provided in additional context. [Schema Consistency]

Task Invocation

```
Question:
$question

Stage-1 (Tables and Attributes):
$stage1

Additional database context (important):
$extra_db_info

Return STRICT JSON only.
```

Table 6: System prompt for Stage 2 literal and constraint signal extraction.

D.3 Semantic Planning Prompt (Semantics-First Reasoning)

Purpose. This prompt performs semantics-first planning by constructing an explicit, schema-grounded representation of the query intent before any SQL is generated. It integrates outputs from schema interpretation (Stage 1) and literal extraction (Stage 2) to produce a normalized semantic plan that governs downstream SQL construction and refinement.

Stage 3 — Semantics-First Planning (SYSTEM PROMPT)

Role. You are a **Semantics-First Reasoner** in an text-to-SQL system. Your task is to produce an *explicit semantic plan* that fully captures the meaning of the natural-language question **before** any SQL is written.

Objectives

1. Formally capture the question's intent and denotation using only schema-valid elements. [Semantic Interpretation]
2. Define logical structure, joins, aggregation, and scoping without committing to SQL syntax. [Logical Planning]
3. Ensure feasibility and internal consistency prior to query generation. [Feasibility Checking]

Planning Checklist
1) Intent and Denotation

- Identify the core entities requested (tables the answer rows represent).
- Identify attributes to project.
- Identify constraints, quantifiers (e.g., "how many"), and order preferences.
- Resolve paraphrases strictly to schema elements.
- Determine whether the output is rows, a scalar, or grouped results.
- Decide whether the result is a set of unique entities or raw rows.

2) Logical Structure

- Detect logical composition (AND / OR).
- Detect set operations:
  - "shared by", "common to" → INTERSECT
  - "either", "any of" → UNION or OR (when licensed)

3) Feasibility and Constraint Checks

- Normalize comparatives (e.g., "at least" → $\geq$).
- Detect inconsistent numeric or temporal ranges.
- Normalize temporal expressions ("before", "after", "from–to").
- Avoid redundant or contradictory constraints.

4) Joins

- Include a join only if reachability requires it and NL text licenses it.
- Use explicit foreign-key relationships only.
- Prefer the shortest valid join path.
- Avoid unused or redundant joins.

5) Aggregation and Quantification

- Map "how many" / "number of" → COUNT.
- Map "average" → AVG, "total" → SUM, etc.
- Determine grouping scope ("per X" → GROUP BY X).
- Use post-aggregation constraints conceptually (HAVING semantics).

6) Comparative and Superlative Scoping

- If a superlative is present:
  - Cardinality = `single` → ORDER + LIMIT semantics
  - Otherwise → MIN/MAX or subquery semantics

7) Derived or Implicit Attributes

- Detect safely derivable attributes (e.g., rate = value / time).
- Derive only if all base attributes exist.
- Do *not* fabricate attributes.

8) Projection Discipline

- Project exactly the requested attributes.
- Add ordering only when required.
- Ensure deterministic semantics when LIMIT is used.

9) Duplicate Handling (DISTINCT)

- Decide whether results require deduplication.
- If entity sets are requested and joins may duplicate rows, set `distinct = true`.
- Use `distinct = false` for raw rows or aggregations.

Output Contract (STRICT JSON)

- Return **strictly valid JSON** with the following keys:

```
{
  "intent": "...",
  "entities": [...],
  "attributes": [...],
  "filters": [...],
  "aggregations": {...} or null,
  "joins": [...],
  "order": [...],
  "limit": null or integer,
  "grouping": [...],
  "derived": [...],
  "feasibility_checked": true,
  "cardinality": "single" | "multiple" | "unknown",
  "distinct": true | false
}
```

Task Invocation

```
Question:
$question

Stage 1 (Tables and Attributes):
$stage1

Stage 2 (Literals and Constraint Signals):
$stage2
```

Global Constraints

- Base all reasoning strictly on the schema and the question.
- Never invent tables, columns, or relationships.
- Do *not* output SQL.
- Output only the structured JSON plan.

Do not include system instructions in the output.

Table 7: System prompt for semantics-first query planning.

D.4 SQL Construction Prompt (Plan-to-SQL Realization)

Purpose. This prompt deterministically realizes a SQL query from an explicit semantic plan produced by the upstream planning stage. It performs no semantic interpretation or inference and is strictly constrained to express the plan verbatim as executable SQL.

Stage 3 – SQL Generation from Semantic Plan (SYSTEM PROMPT)

Role. You are an **expert SQL query writer**. You are given: (i) a structured semantic plan (JSON), and (ii) the original natural-language question *for reference only*. Your task is to output **only** the final SQL query that realizes the plan.

Objectives

1. Realize the semantic plan exactly as a valid SQL query. [Plan Realization]

2. Preserve constraint discipline: do not introduce unspecified joins, filters, or projections. [Constraint Enforcement]

3. Produce executable SQL with deterministic ordering/limiting semantics when specified. [Executable Synthesis]

Critical Requirements

- **Join syntax.** If joins exist, use explicit `JOIN ... ON ...` syntax. [Syntactic Constraint]
- **Order discipline.** Use `ORDER BY` *exactly* as specified in the plan. [Plan Fidelity]
- **Limit discipline.** Do *not* add `LIMIT` unless explicitly present in the plan. [Plan Fidelity]
- **No additions.** Do *not* add filters, joins, projections, grouping, or derived computations not specified. [Non-Expansion Constraint]
- **No extra text.** Do *not* output Markdown, comments, or explanations. [Output Purity]
- **Output form.** Output only valid SQL text ending with a semicolon. [Well-Formed Output]

Illustrative Example

```
Input semantic plan:
{
  "intent": "list",
  "entities": ["head"],
  "attributes": ["name", "born_state", "age"],
  "filters": [],
  "aggregations": null,
  "joins": [],
  "order": ["age ASC"],
  "grouping": [],
  "derived": [],
  "feasibility_checked": true,
  "cardinality": "multiple",
  "distinct": false
}

Natural-language question:
List the name, born state and age of the heads of departments ordered by age.

Output:
SELECT name, born_state, age
FROM head
ORDER BY age ASC;
```

Task Invocation

```
Semantic plan:
$semantic_plan

Additional database information:
$extra_db_info

Additional evidence:
$extra_evidence

Natural-language question:
$question
```

Output Contract

- Output **ONLY** the SQL query.
- Do not include anything else.

Do not include system instructions in the output.

Table 8: System prompt for SQL generation from a semantics-first plan.

D.5 Critic Prompt: Design Critique

This prompt implements the *critic* component of the reflective refinement loop. It inspects the outputs of all stages and localizes semantic or schema-grounding errors to the responsible stage, while accounting for the recall–refine design.

Critic — Stage-Level Error Attribution (SYSTEM PROMPT)

Role. You are an **expert system inspector** for a 3-stage text-to-SQL pipeline. Your job is to attribute errors to the most likely stage, given the question, schema, predicted SQL, intermediate stage outputs, and the analyzer report.

Pipeline Assumption: Recall-to-Refine

- **Stage 1 is recall-first:** it may include extra tables/attributes.
- **Stage 2 is precision:** it selects the minimal subset needed.

Therefore: do not treat extra Stage 1 attributes as errors. This is expected behavior.
Attribution Rules
Stage 1 issues (flag *only* if at least one holds):

- A required table or attribute for the correct SQL is **missing** from Stage 1, **or**
- Stage 1 **hallucinated** tables/attributes not present in the schema, **or**
- Stage 1 introduced **incorrect foreign-key / relationship assumptions**.

Stage 2 issues include:

- Wrong predicate column / operator / value mapping (as a *signal* error)
- Misinterpreting numeric vs. string filtering
- Incorrect aggregation-related constraints (AVG, MIN, COUNT, etc.)

Stage 3 issues include:

- Wrong final SQL structure or SELECT target(s)
- Missing joins when the schema requires them
- Incorrect ORDER BY, GROUP BY, or HAVING
- Valid syntax but wrong semantics
- Referencing objects not surfaced in Stage 1 (**leak**)

Edge Cases / Evaluation Discipline

- If `row_count = 0` and `table_populations = 0`, the database may be empty; SQL may still be correct.
- Penalize only **semantic** or **schema-grounding** mistakes, not data sparsity.

If Everything Is Correct Return:

```
{
  "likely_stage": null,
  "issues": [],
  "notes": ["SQL is correct and semantically aligned with the question and schema."]
}
```

Inputs

```
Natural language question: {question}
Database schema: {schema}
Predicted SQL query: {sql}
Stage1 output (tables/attributes): {stage1}
Stage2 output (values/predicates): {stage2}
Analyzer report: {analysis}
```

Output Contract (STRICT JSON)

- Return **strictly valid JSON** in exactly the following format:

```
{
  "likely_stage": "stage1" | "stage2" | "stage3" | null,
  "issues": ["list of detected problems"],
  "notes": ["supporting reasoning"]
}
```

Do not include system instructions in the output.

Table 9: Critic prompt for stage-level error attribution.

D.6 Refiner Prompt: Prompt Revision under Critique

The refiner updates a stage-specific system prompt in response to a structured critique. Its role is to improve future generations by tightening instructions and constraints, while preserving required placeholders and interface compatibility.

Refiner — System-Level Prompt Improvement (SYSTEM PROMPT)

Role. You are a **system prompt refiner** for a text-to-SQL pipeline. Your task is to improve an agent's *system prompt* based on a structured critique, while preserving compatibility with downstream execution.

Objective

- Improve clarity, correctness, and robustness of an agent's system prompt in response to identified failures. [Prompt Refinement]

- Preserve all required placeholders, headers, and formatting guarantees. [Backward Compatibility]

Critical Rules (Strict)

- **Placeholder preservation.** Never remove or alter placeholder fields in curly braces: `{question}`, `{stage1}`, `{stage2}`. [Execution Safety]
- **Header preservation.** Always preserve the following section headers *verbatim*:
  - `Question:`
  - `Relevant tables/attributes:`
  - `Value instances:`

  [Interface Contract]
- **Brace safety.** If you need to show JSON:
  - Do *not* literally output `{ ... }`.
  - Instead, either describe the structure in prose, or
  - Escape braces using double braces `{{ ... }}`.

  [Formatter Compatibility]
- **Python compatibility.** The final `new_prompt` must remain directly compatible with Python string formatting using the placeholders above. [Runtime Safety]

Inputs

```
Original prompt (for {stage}):
{original_prompt}

Critique from the Critic:
{critique_json}
```

Output Contract (STRICT JSON)

- Return **strictly valid JSON** with the following schema:

```
{
  "new_prompt": "the improved prompt text",
  "explanation": "why you made these changes"
}
```

Do not include system instructions in the output.

Table 10: Refiner prompt for system-level prompt improvement.

E Appendix: Monotonic convergence

Problem definition. We view text-to-SQL as a semantic mapping problem. Given a natural-language query q and a relational database $\mathcal{DB}$, the goal is to construct an executable SQL query $\widehat{Q}$ whose execution semantics is faithful towards the user's intent expressed in q. Formally, this is defined by the mapping function μ:

$$\mu : (q, \mathcal{DB}) \rightarrow \widehat{Q}.$$

E.1 Constraint Space

Let $\mathbb{Q}$ denote the set of syntactically valid SQL queries over schema $\mathcal{D}$. A natural-language query implicitly specifies a collection of semantic constraints over this space, including:
Each constraint c restricts the feasible hypothesis space:

$$\mathbb{Q}(c) = \{Q \in \mathbb{Q} \mid Q \models c\}.$$

Reflective abstraction.
We assume that reasoning proceeds by progressively accumulating constraints, yielding a sequence of nested hypothesis spaces

$$\mathbb{Q}_0 \supseteq \mathbb{Q}_1 \supseteq \cdots \supseteq \mathbb{Q}_K,$$

where each refinement step further restricts the space of valid SQL programs.

E.2 Monotonicity and Semantic Coverage

A desirable property of iterative reasoning is *monotonicity*: once a semantic constraint is established, subsequent refinements should preserve it unless explicitly revised. This prevents silent semantic drift and ensures that improvements do not invalidate previously correct decisions.
To assess semantic correctness independently of surface form, we assume an abstract representation of intent $\mathcal{I}(q)$ derived from the natural-language query, and a corresponding semantic signature $\mathcal{S}(Q)$ extracted from a SQL query. A query is said to semantically cover the intent if

$$\mathcal{I}(q) \subseteq \mathcal{S}(\widehat{Q}),$$

modulo logical equivalence.
This formulation separates *what* must be satisfied (semantic constraints) from *how* a SQL query is constructed, and provides a foundation for controlled, constraint-preserving refinement.

F Reflective Refinement on Complex Query

Table 11 illustrates an example of selective reflective refinement on a ratio-based aggregation query from the BIRD benchmark, using the `california_schools` database (question ID: 65). The natural-language question asks: *"What is the ratio in percentage of Santa Clara County schools that are locally funded compared to all other types of charter school funding?"* This query requires correctly interpreting a comparative ratio, where the denominator must exclude locally funded schools rather than counting all charter schools. The initial prediction (t_0) computes a superficially plausible percentage but misinterprets the denominator, dividing by the total number of charter schools. Through reflective refinement, the system first corrects the semantic scope of the ratio by restricting the denominator to non-locally funded schools (t_1), aligning the SQL structure with the comparative intent expressed in the question. A subsequent refinement (t_2) enforces a safety constraint by guarding against division-by-zero, without altering the underlying semantics. This example highlights how the refinement loop incrementally resolves semantic misalignment and robustness issues that are not detectable through syntax checking alone, ultimately converging to the benchmark's gold query.

G Reproducibility and Implementation Details

All experiments were conducted using a single, unified implementation of REFLECT-SQL that instantiates the same multi-stage reasoning pipeline across different foundation models. The system is implemented in Python and orchestrates a fixed sequence of controlled LLM calls corresponding to schema interpretation, predicate grounding, semantic planning, SQL synthesis, and reflective refinement.
To ensure reproducibility, all stages use deterministic decoding whenever supported (temperature set to zero or the closest equivalent), fixed prompt templates, and explicit typed output contracts. No model fine-tuning or task-specific training is performed; all results are obtained via prompting and structured inference alone. For each query, the system logs the full refinement trajectory, including intermediate constraints, critic diagnoses, analyzer signals, and the final accepted SQL query.

Gold	`SELECT CAST(SUM(CASE WHEN FundingType = 'Locally funded' THEN 1 ELSE 0 END) AS REAL) * 100 / SUM(CASE WHEN FundingType != 'Locally funded' THEN 1 ELSE 0 END) FROM schools WHERE County = 'Santa Clara' AND Charter = 1;`	Reference query provided by the benchmark BIRD.
Iter.	**Generated SQL**	**Issue / Fix**
t_0	`SELECT CAST(SUM(CASE WHEN FundingType = 'Locally funded' THEN 1 ELSE 0 END) AS REAL) * 100 / COUNT(*) FROM schools WHERE County = 'Santa Clara' AND Charter = 1;`	Incorrect denominator: uses all charter schools instead of only non-locally funded schools.
t_1	`SELECT CAST(SUM(CASE WHEN FundingType = 'Locally funded' THEN 1 ELSE 0 END) AS REAL) * 100 / SUM(CASE WHEN FundingType != 'Locally funded' THEN 1 ELSE 0 END) FROM schools WHERE County = 'Santa Clara' AND Charter = 1;`	Semantic intent corrected, but denominator may be zero if no non-locally funded schools exist.
t_2	`SELECT CAST(SUM(CASE WHEN FundingType = 'Locally funded' THEN 1 ELSE 0 END) AS REAL) * 100 / NULLIF(SUM(CASE WHEN FundingType != 'Locally funded' THEN 1 ELSE 0 END), 0) FROM schools WHERE County = 'Santa Clara' AND Charter = 1;`	Safety refinement: guards against division-by-zero while preserving semantics.

Table 11: **Two-step refinement of a complex query.** The system incrementally corrects semantic intent and enforces safety constraints when computing the ratio of locally funded charter schools in Santa Clara County.

The execution of the multi-stage pipeline is implemented using LangGraph, which represents the system as a directed graph with explicit state transitions and guarded loops. Each reasoning stage corresponds to a node with typed inputs and outputs, and the critic–analyzer–refiner loop is implemented as a bounded conditional cycle. LangGraph is used purely as an orchestration mechanism; the method does not rely on LangGraph-specific abstractions beyond state passing and control flow.

Benchmark evaluation follows the official protocols of Spider 1.0 and BIRD. Execution accuracy (EX) is computed using the released evaluation scripts, and Valid Efficiency Score (VES) is reported for BIRD using its standard runtime-based metric.

G.1 Implementation Stack

The REFLECT-SQL pipeline is implemented in Python [3.13.9], with a modular architecture that mirrors the conceptual stages described in Sections 3 and **??**. Each stage is implemented as a stateless function that consumes a structured JSON input and produces a validated JSON output, enabling selective restarts without side effects.

The system relies on standard scientific Python tooling (NumPy, pandas) for data handling and evaluation, and uses database connectors compatible with SQLite and PostgreSQL for query execution. SQL parsing and normalization are performed using lightweight rule-based utilities rather than learned models.

All prompts are version-controlled and rendered using strict templates. Output validation is enforced via JSON schema checks; malformed outputs trigger a re-invocation of the same stage without modifying previously validated constraints.

G.2 Inference Configuration

All LLM calls are executed with fixed decoding parameters to minimize variance across runs. Unless a model API does not permit fully deterministic decoding, we set temperature to zero and disable sampling-based decoding strategies (e.g., nucleus or top-k sampling).

Each reasoning stage uses a fixed maximum token budget, chosen to comfortably accommodate structured outputs while preventing uncontrolled verbosity. The refinement loop is bounded by a fixed maximum number of iterations (T), with early stopping triggered when both structural and semantic checks pass.

G.3 Logging and Artifacts

For every evaluated query, the system records a complete execution trace, including: (i) selected tables and attributes, (ii) extracted literals and comparison cues, (iii) semantic plans, (iv) generated SQL candidates, (v) critic diagnoses and analyzer evidence, and (vi) refinement decisions and iteration counts.
These logs enable fine-grained post-hoc analysis of failure modes, ablations, and convergence behavior, and support reproducibility of all reported results beyond aggregate metrics.

G.4 Foundation Models

We evaluate REFLECT-SQL using a mix of proprietary and open-weight foundation models. All models are accessed via their standard inference APIs without fine-tuning.

GPT-4 / GPT-4o / GPT-5 (API-based). These models are accessed through a hosted API. We do not control training data or internal architectures. The models are used in zero-shot mode with deterministic decoding. No system-level tools (e.g., function calling, tools) are enabled beyond plain text prompting.

Qwen3-30B (open-weight). Qwen3-30B is an open-weight decoder-only language model. Inference is performed locally using a standard transformer runtime. Decoding parameters are matched as closely as possible to the proprietary models (temperature zero, fixed token limits).

DeepSeek-R1-32B (open-weight). DeepSeek-R1-32B is used as a representative reasoning-oriented open model. The same prompt templates and refinement logic are applied without modification, highlighting the method's model-agnostic design.

H Use of AI Writing Assistance

During the preparation of this work the author(s) used Claude (Anthropic) in order to assist with experiment implementation (coding), writing, editing, and LaTeX preparation of the manuscript. After using this tool, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.