INTERNALIZING EXECUTION FOR TEXT-TO-SQL WITH PREDICTIVE RELATIONAL VERIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Execution feedback helps Text-to-SQL systems identify and revise incorrect query choices, but obtaining it typically requires interaction with a database engine. We investigate whether models can provide this feedback by predicting intermediate query results during SQL construction. On the same BIRD questions, we compare schema-only SQL generation with prediction of the corresponding gold SQL's results on MiniDBs constructed using that SQL. Models can accurately predict these results yet struggle to generate correct queries for the same questions, suggesting a gap between execution capability and use. We introduce SIMSQUIRREL, a family of 4B, 8B, and 14B models trained to use execution predictions during SQL construction. Within a single continuous response, each model generates local SQL, predicts its result relation, and selects CONTINUE or FIX to guide query construction without external execution feedback. To supply data for these predictions, we construct diagnostic MiniDBs without reference SQL, selecting linked rows within the context budget. We train this behavior by first developing execution prediction through supervised fine-tuning and reinforcement learning, then optimizing complete Text-to-SQL trajectories. During trajectory training, identical rewards on a single database yield zero group-relative advantages. Our counterexample-guided dynamic sampling therefore attempts to distinguish candidates from the reference query on shared synthesized databases, rescores each group, and retains groups with differing rewards for policy optimization. On BIRD dev, SIMSQUIRREL-14B achieves 73.86% execution accuracy, exceeding Qwen3-14B by 23.99 percentage points and the strongest baseline in our comparison by 3.06 points. Training-stage ablations show that, at 8B, the complete method exceeds Text-to-SQL reinforcement learning alone by 12.79 points.

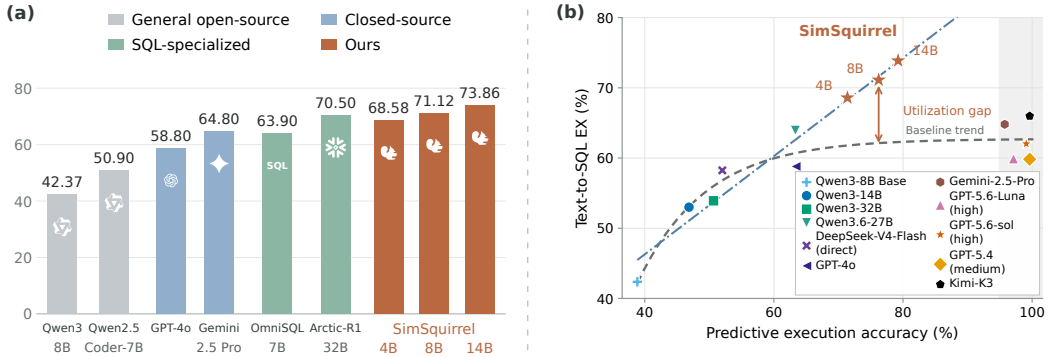


Figure 1: **(a)** BIRD dev execution accuracy for selected baselines and SIMSQUIRREL, colored by model category (Table 1; Appendix B.2). Arctic-R1 denotes Arctic-Text2SQL-R1. Our best model achieves 73.86% EX, exceeding all baselines shown. **(b)** Predictive execution and Text-to-SQL accuracy on the same BIRD subset; each point represents one model. Copper stars mark SimSquirrel. The gray curve fits all baselines; the blue dash-dotted line fits the six baselines with predictive execution accuracy below 70% and our three models. The arrow marks the descriptive gap between SimSquirrel-8B and the baseline curve.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

1 INTRODUCTION

Text-to-SQL translates a natural-language question into an executable query over a relational database. Generating correct SQL on benchmarks such as BIRD (Li et al., 2023) requires selecting appropriate joins and filters while preserving the intended aggregation granularity. Execution results help systems revise or select queries (Deng et al., 2025; Luo et al., 2026). Code models also learn from execution feedback and environment trajectories (Gehring et al., 2024; FAIR CodeGen team et al., 2025), and can be trained to predict program outputs for verification (Maimon et al., 2026).

Obtaining execution feedback requires a compatible database environment and appropriate access permissions. In enterprise deployments and benchmarks using hosted databases, exploratory queries can incur compute charges, while production policies may restrict unverified queries. Given prepared relational data, predictive execution can estimate local results without submitting intermediate queries to the online database. These considerations motivate our question: *can SQL execution be internalized as a model capability and used while a query is still being constructed?*

We compare predictive execution and Text-to-SQL on the same BIRD dev examples to control for sample selection. For execution prediction, each model receives gold SQL and a MiniDB, constructed by tracing the query's joins and filters to source rows and adding rows that satisfy or violate local predicates. Predictions are scored against SQLite execution on this MiniDB. For Text-to-SQL, baselines receive schema-only inputs, and generated queries are scored on the original databases.

Across the evaluated models (Yang et al., 2025; OpenAI, 2024; Comanici et al., 2025), predictive execution accuracy is positively associated with Text-to-SQL accuracy, but Text-to-SQL accuracy levels off among the strongest execution predictors (Figure 1b). GPT 5.4 (OpenAI, 2026a), for example, reaches 99.62% predictive execution accuracy but 59.83% Text-to-SQL accuracy. If a model can predict the result of a correct query, can it use this capability to construct correct queries more reliably? We call this mismatch the *gap between execution capability and use*. Figure 1b illustrates this gap through SimSquirrel's higher Text-to-SQL accuracy relative to the baseline trend at comparable predictive execution accuracy; this cross-model comparison remains descriptive.

Some systems use execution feedback to refine or select completed SQL candidates (Deng et al., 2025; Luo et al., 2026). Other methods execute intermediate SQL (Dai et al., 2026b;a) or Python operations (Wang et al., 2026) and return engine results to guide subsequent reasoning.

We introduce SIMSQUIRREL to train predictive execution and use predicted relations as feedback within a single continuous response. Result prediction needs relational data, but full databases can exceed context budgets, a constraint also addressed by context selection and compression (Talaei et al., 2024; Deng et al., 2025). We construct a diagnostic MiniDB without reference SQL, selecting linked rows that expose differences in filtering and join multiplicity. The model generates local SQL on this MiniDB, predicts its result relation, and selects CONTINUE or FIX to extend or correct the query before producing the final SQL, without external execution feedback during generation.

We first train execution prediction with supervised fine-tuning and reinforcement learning (RL), then optimize complete Text-to-SQL trajectories using final-query execution rewards. Single-database binary rewards can produce all-zero or all-one groups with zero relative advantages. Before discarding such groups, our counterexample-guided dynamic sampling attempts to distinguish their candidates from the reference query on shared synthesized databases, rescores each group, and retains groups with differing rewards for policy optimization.

On BIRD development, SIMSQUIRREL-14B achieves 73.86% execution accuracy, exceeding Qwen3-14B by 23.99 percentage points and Databricks-RLVR-32B, the strongest baseline in Table 1, by 3.06 points. Training-stage ablations show downstream gains from both stages; at 8B, the complete two-stage method exceeds Text-to-SQL RL alone by 12.79 points (Figure 3).

Our contributions are:

- We introduce SIMSQUIRREL models at 4B, 8B, and 14B scales. We train execution prediction through supervised fine-tuning and RL, then optimize complete Text-to-SQL trajectories using final SQL execution rewards to optimize intermediate decisions. Within a single response, the models generate local SQL, predict its result relation, and use CONTINUE/FIX decisions to guide query construction without external execution feedback.

- We construct diagnostic MiniDBs to supply relational data within the model's context budget. At inference, the builder selects linked rows and required columns using the question, schema, and database contents without reference SQL. These fixed contexts retain data for checking filters, join multiplicities, and aggregation.
- We introduce counterexample-guided dynamic sampling for Text-to-SQL RL. Before discarding a group with identical execution rewards and zero relative advantages, we attempt to synthesize databases that distinguish candidates from the reference query, rescore all candidates on these shared databases, and retain groups with differing rewards.

2 Related Work

Execution guidance for Text-to-SQL and context selection. Recent Text-to-SQL systems access a database during generation rather than producing SQL in a single pass. ReFoRCE and SIRIUS use execution to refine, rerank, or select candidate queries (Deng et al., 2025; Luo et al., 2026). ReEx interleaves intermediate SQL execution with decoding, while VET represents reasoning as Python operations executed step by step against the database (Dai et al., 2026b; Wang et al., 2026). A probing and verification approach probes concrete records to resolve schema and value ambiguity before applying an executable verification checklist; related work studies credit assignment in trajectories that use tools (Tian & Zhang, 2026; Dai et al., 2026a). These methods receive feedback after an executable probe or query is available. Methods for selecting context address a related prerequisite: CHESS retrieves schema elements and values relevant to the question, ReFoRCE combines table compression with iterative column exploration, and bidirectional retrieval improves schema linking (Talaei et al., 2024; Deng et al., 2025; Nahid et al., 2026). However, schema retrieval does not identify the rows needed to test competing local operations. We train a model to predict local SQL results and use these predictions to guide subsequent query construction. Our diagnostic MiniDB supplies the linked data needed for these predictions and remains fixed during generation. Database execution remains the reference for observed outcomes; our contribution is a feedback interface used during query construction.

Learning to model execution. Execution grounding has also been studied as a learned capability. CRUXEval separates prediction of inputs and outputs from code generation and shows that gains on generation benchmarks need not transfer to execution reasoning (Gu et al., 2024). NExT teaches models to reason over runtime traces, and RLEF optimizes code models from execution feedback (Ni et al., 2024; Gehring et al., 2024). CWM learns over program actions and observable states, while AgentWorld predicts observations across heterogeneous agent environments (FAIR CodeGen team et al., 2025; Zuo et al., 2026). Other work trains models to predict program outputs for verification (Maimon et al., 2026). We apply this perspective to relational programs, where the predicted state is a structured relation with rows, cardinality, duplicates, nulls, and ordering. Our evaluation separately measures SQL generation accuracy and execution prediction accuracy.

Semantic supervision across instances. Execution accuracy on one database instance can accept semantically different queries that happen to return the same result. Distilled test suites reduce this ambiguity by selecting generated databases with high query coverage (Zhong et al., 2020), and SpotIt uses bounded formal verification to find a database that distinguishes two queries (Klopfenstein et al., 2025). SynSQL constructs databases conditioned on the question and schema without access to gold SQL and shows that static evaluation can mask errors across Text-to-SQL systems (Habibollah & Rafiei, 2026). We use synthesized databases to rescore rollout groups with identical execution rewards before filtering them during training. Each group shares the same test set, and groups whose rewards differ after rescoring enter policy optimization.

3 Method

3.1 Task formulation

Let x be a natural language question, S a relational schema, and D a database instance that conforms to S. We use $\mathrm{Exec}_\tau(q, D)$ to denote the canonical result of executing SQL query q on D, where τ specifies whether row order and duplicate multiplicity matter. Text-to-SQL generates a query y from x, S, and the permitted database context. It succeeds when $\mathrm{Exec}_\tau(y, D) = \mathrm{Exec}_\tau(y^\star, D)$ for

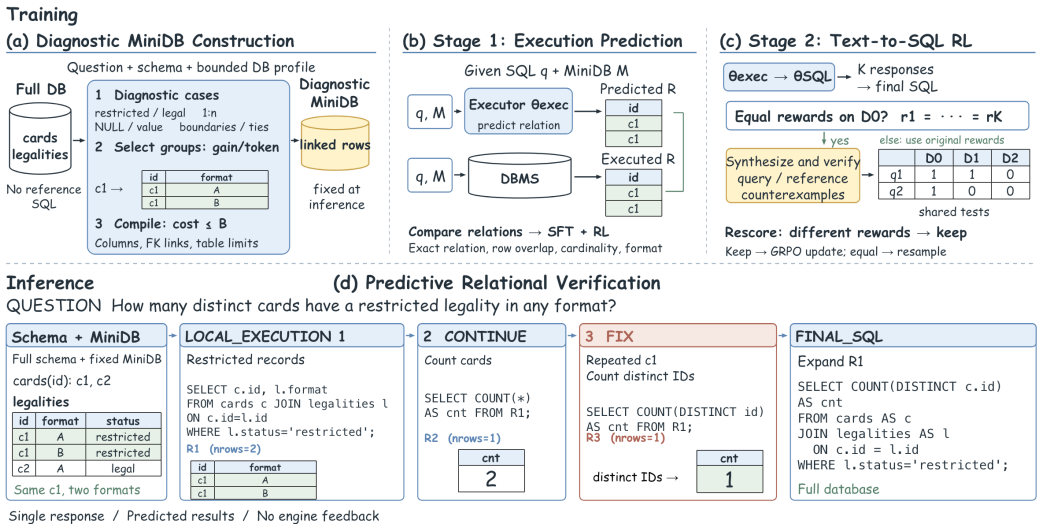


Figure 2: Overview of SIMSQUIRREL. (a) Diagnostic MiniDB construction selects linked row groups by coverage gained per token within budget, without reference SQL. (b) Training compares predicted and executed relations. (c) Stage 2 initializes from the trained executor. D_0 is the original database; D_1, D_2 are shared synthesized databases. Entries indicate agreement (1) or disagreement (0) with reference results. Groups with identical original rewards are rescored and retained only if rewards differ. (d) Local SQL, predicted relations, and CONTINUE/FIX decisions guide final SQL generation in one continuous response without engine feedback.

a reference query $y^\star$. Predictive execution instead receives a fixed query q and a visible relational instance M, produces a relation $\widehat{R}$ without invoking a database engine, and is evaluated against $R = \mathrm{Exec}_\tau(q, M)$. Text-to-SQL must infer a program from language; predictive execution holds the program fixed and measures whether the model can derive its result.

3.2 METHOD OVERVIEW

Our method has two training stages and uses predicted execution to guide SQL construction. In the first stage, we construct a diagnostic MiniDB D_x^{mini} from D. This small database retains linked rows that make consequential local operations produce different observable results. We train a predictive executor on D_x^{mini} with verifiable rewards derived from relations computed by a database engine. In the second stage, we initialize a Text-to-SQL model from the trained executor and use counterexample-guided dynamic sampling: groups with identical execution rewards are rescored on shared synthesized databases before filtering. During inference, the model generates local SQL, predicts its result using D_x^{mini} and earlier predicted relations, and selects CONTINUE or FIX for the next step. Figure 2 summarizes both training stages and inference.

3.3 BUDGETED CONSTRUCTION OF DIAGNOSTIC MINIDBS

The full database D is usually too large to include in the model input. We construct a diagnostic MiniDB that keeps the data needed to check SQL decisions. Value matching can retrieve constants mentioned in the question, but it does not preserve one-to-many joins, duplicate multiplicities, `NULL` contrasts, threshold boundaries, or ordering ties.

Candidate data and budget. For a Text-to-SQL instance x, let $\mathcal{B}_x$ denote the candidate linked row groups. Each group $b \in \mathcal{B}_x$ contains the rows, key columns, and foreign-key connections that preserve one relational configuration. For $A \subseteq \mathcal{B}_x$, the MiniDB $D_x^{\mathrm{mini}}(A)$ is the table-wise union of the selected groups, with duplicate rows removed and required links retained. Group b has serialization cost $c(b)$; a feasible selection satisfies $\sum_{b \in A} c(b) \le B$.

Ideal objective. Let $y_x^\star$ be the reference SQL. Model generations and controlled semantic perturbations provide a finite candidate set $\mathcal{Y}_x$. The following nonempty subset disagrees with $y_x^\star$ on D:

$$\mathcal{Y}_x^- = \{\widehat{y} \in \mathcal{Y}_x : \mathrm{Exec}_\tau(\widehat{y}, D) \neq \mathrm{Exec}_\tau(y_x^\star, D)\}. \tag{1}$$

The selection objective counts how many of these differences remain visible on the MiniDB:

$$J_x(A) = \frac{1}{|\mathcal{Y}_x^-|} \sum_{\widehat{y} \in \mathcal{Y}_x^-} \mathbb{I}\big[\mathrm{Exec}_\tau(\widehat{y}, D_x^{\mathrm{mini}}(A)) \neq \mathrm{Exec}_\tau(y_x^\star, D_x^{\mathrm{mini}}(A))\big],$$
$$A_{\mathrm{sem}}^\star \in \arg\max_{A \subseteq \mathcal{B}_x} J_x(A) \quad \text{s.t.} \quad \sum_{b \in A} c(b) \leq B. \tag{2}$$

Here, $J_x(A)$ is the fraction of known SQL errors that remain observable on the MiniDB. An exhaustive search for $A_{\mathrm{sem}}^\star$ executes every candidate for every selection under consideration. Adding rows may also change whether two SQL queries agree, so J_x is neither guaranteed to be monotone nor submodular.

Stable diagnostic cases. For analysis, we replace repeated pairwise execution with a finite library of diagnostic cases. Each case records a concrete data configuration, such as a repeated join key, an unmatched row, a `NULL`, a threshold boundary, or a complete tie group. Given a reference query and a candidate error, let $\mathcal{U}_x(\widehat{y})$ contain the cases that are *stable* for candidate $\widehat{y}$. A case is stable if every admissible MiniDB containing its required rows, columns, and links keeps the outputs of $\widehat{y}$ and $y_x^\star$ different. Let $C_x(b)$ be the cases instantiated by group b and $C_x(A) = \bigcup_{b \in A} C_x(b)$. The surrogate counts a candidate error once at least one of its stable cases is retained:

$$F_x(A) = \frac{1}{|\mathcal{Y}_x^-|} \sum_{\widehat{y} \in \mathcal{Y}_x^-} \mathbb{I}[\mathcal{U}_x(\widehat{y}) \cap C_x(A) \neq \emptyset]. \tag{3}$$

Theorem 1 (Diagnostic coverage)**.** *If the compiler retains the rows, columns, and links required by every covered stable case, then for every feasible A,*

$$F_x(A) \leq J_x(A). \tag{4}$$

The function F_x is also normalized, monotone, and submodular over linked row groups.

Every error counted by F_x is also counted by J_x, so diagnostic coverage is a computable lower bound on preserved execution feedback. The surrogate optimum and the compiler output are

$$A_F^\star \in \arg\max_{A \subseteq \mathcal{B}_x} F_x(A) \quad \text{s.t.} \quad \sum_{b \in A} c(b) \leq B, \qquad \widehat{A} = \mathrm{Compiler}(x, S, D, B). \tag{5}$$

The two approximation gaps have different sources. Replacing execution with case coverage yields $A_F^\star$ in place of $A_{\mathrm{sem}}^\star$, while a greedy search produces $\widehat{A}$ in place of $A_F^\star$. For the idealized objective, this search starts from $A_0 = \emptyset$ and repeatedly adds the feasible group with the largest marginal coverage per token:

$$b_t \in \arg\max_b \frac{F_x(A_{t-1} \cup \{b\}) - F_x(A_{t-1})}{c(b)}. \tag{6}$$

In practice, a question-conditioned specification procedure identifies relevant diagnostic cases from the question, schema, and a bounded profile of D. A deterministic compiler then selects linked row groups by uncovered case gain per token while enforcing table limits and foreign-key closure. These additional constraints fall outside the simplified objective, so we do not claim the standard approximation ratio for the complete builder. Appendix A proves the theorem and analyzes both tightness and the failure of uniform sampling on rare cases.

3.4 Learning predictive execution

Stage 1 develops predictive execution through supervised fine-tuning followed by reinforcement learning. Given a query q and relational instance M, a database engine computes the target $R = \mathrm{Exec}_\tau(q, M)$ offline. In both phases, the model receives the schema, M, and q. The engine output provides SFT targets and verifiable RL rewards but is excluded from the prompt.

Supervised fine-tuning. We serialize each target relation as a JSON object with `columns`, `rows`, and `nrows`. Under teacher forcing, we minimize the token-level negative log likelihood of the target response, masking all input tokens. Targets retain column order, duplicate rows, `NULL` values, and any required row order, so supervision covers both relation contents and output structure.

Reinforcement learning. We optimize the SFT model with GRPO (Shao et al., 2024) using execution-derived rewards. Let E indicate exact relation agreement, F_{row} and F_{col} denote multi-set F1 scores for rows and column names, N indicate a correct row count, and V indicate a valid relation prediction. We use

$$r_{\text{exec}}(\widehat{R}, R) = \lambda_E E + \lambda_F F_{\text{row}} + \lambda_C F_{\text{col}} + \lambda_N N + \lambda_V V. \tag{7}$$

Exact agreement checks column names and order, row count, and result rows under τ, preserving duplicates and any required row order. Both F1 terms preserve multiplicity but ignore order. Invalid responses receive zero reward. Appendix B.1 lists the nonnegative weights, which sum to one.

3.5 Predictive relational verification during query construction

We use the execution capability developed in Section 3.4 during query construction within a single continuous response. The model receives the question x, permitted evidence, schema S, and a fixed MiniDB D_x^{mini}, without external execution feedback during generation.

Local SQL and predicted relations. At step t, the model states a subgoal based on the question or evidence, generates a read-only local SQLite query q_t, and predicts its result. The query may read MiniDB tables and previously predicted relations $\widehat{R}_{<t} = (\widehat{R}_1, \ldots, \widehat{R}_{t-1})$, referenced as $R_1, \ldots, R_{t-1}$:

$$\widehat{R}_t = \text{Sim}_\theta\Big(q_t; D_x^{\text{mini}}, \widehat{R}_{<t}\Big). \tag{8}$$

The output format specifies columns, all result rows, and cardinality; the model is instructed to preserve SQLite semantics for duplicates, nulls, aggregation, ordering, and type conversion. The same model generates the SQL and predicted relations; no engine results are inserted into the response.

Continuing and correcting the plan. The first step establishes the initial plan. Before each subsequent step, the model checks prior SQL and predicted relations against the question and evidence, including filters, join multiplicity, grouping, selected measures, ordering, and result shape. It emits CONTINUE to add the next operation or FIX when it identifies a mismatch. A continuation reuses an earlier relation when it contains the required inputs. A fix states the mismatch and correction in the new subgoal and implements the correction in a new local query, reading MiniDB tables again when needed. In Figure 2(d), R_1 contains the same card ID in two formats, so the FIX block replaces `COUNT(*)` with `COUNT(DISTINCT id)`, changing the predicted count from two legality records to one distinct card. These decisions use predicted results and can themselves be incorrect.

Final SQL generation. The protocol requires the last local block to represent the final query's result shape and semantics on the MiniDB. The model then expands local relation references into one standalone read-only query over D, incorporating any corrections. Because the MiniDB is a non-exhaustive sample, missing rows do not establish absence from D; literals in query conditions must be justified by the question or evidence.

3.6 Counterexample-guided dynamic sampling

The final SQL produced by the verification protocol in Section 3.5 provides the outcome reward for Text-to-SQL training. We sample K responses per question and compare the outputs of their final queries $y_1, \ldots, y_K$ with the reference output on D. If all K rewards are identical, the group-relative advantages vanish. DAPO's dynamic sampling filters these groups, whose accuracy is zero or one, and samples replacements to fill the training batch (Yu et al., 2025). In Text-to-SQL, however, identical rewards on D can conceal differences across instances: queries may pass accidentally or fail for different reasons. We therefore test whether synthesized databases can distinguish these candidates before discarding the group.

For each candidate paired with the reference, a synthesis model attempts to construct a compact database on which their outputs differ, subject to a fixed synthesis budget per group. We validate each instance against schema and specified domain constraints, retaining it only if the reference

executes successfully and a database engine verifies the disagreement under τ. We then remove unnecessary rows while preserving these conditions and collect the deduplicated instances in a shared test set $\mathcal{T}_x$. Every candidate is scored on every database in this set, so a counterexample found for one query can also expose errors in the others. Let $e(y, D') = \mathbb{I}[\mathrm{Exec}_\tau(y, D') = \mathrm{Exec}_\tau(y^\star, D')]$, with candidate execution errors scored as zero. For $\mathcal{T}_x \neq \emptyset$, the reward combines original execution accuracy with the fraction of synthesized tests passed:

$$r_k = \alpha e(y_k, D) + \frac{1-\alpha}{|\mathcal{T}_x|} \sum_{D' \in \mathcal{T}_x} e(y_k, D'), \qquad 0 \leq \alpha < 1. \tag{9}$$

This rescoring determines whether the group enters policy optimization: we retain it only if the rewards now differ. If no valid counterexample is found or the rewards remain identical, we discard the group and continue sampling until the batch is filled. Groups with differing original rewards bypass synthesis and use $r_k = e(y_k, D)$. All retained groups use the standard GRPO advantage normalization,

$$A_k = \frac{r_k - \mathrm{mean}_j\, r_j}{\mathrm{std}_j\, r_j + \epsilon}. \tag{10}$$

Synthesized databases and reference queries remain hidden from the solver. Failure to find a counterexample does not establish query equivalence.

4 Experiments

4.1 Experimental setup

Baselines. We compare against closed-source models, general-purpose open models, and SQL-specialized models, including OmniSQL (Li et al., 2025), CodeS (Li et al., 2024), Arctic-Text2SQL-R1 (Yao et al., 2025), and Databricks-RLVR (Ali et al., 2025). Table 1 reports model sizes and execution accuracy. Appendix B.2 distinguishes published scores from our evaluations.

Datasets. We evaluate on all 1,534 BIRD development questions (Li et al., 2023) and the SQLite subset of Spider 2.0-Lite (Lei et al., 2024), comprising 135 questions across 30 databases. Stage 1 uses 44,519 execution-SFT examples and 6,397 execution-RL examples; Stage 2 reuses 6,554 training questions. Both evaluation sets are excluded from training and model selection. Data partitions, sampling, and MiniDB construction are described in Appendix B.3.

Evaluation Metrics. We report execution accuracy (EX) on the original databases, using OmniSQL's evaluator for Spider 2.0 SQLite and counting invalid SQL and execution failures as incorrect. Execution-prediction accuracy separately evaluates the predicted relation for a supplied query and MiniDB. Appendix B.4 specifies the comparators and MiniDB diagnostic metrics.

Implementation Details. We initialize SimSquirrel-8B from Qwen3-8B (Yang et al., 2025). Execution SFT and RL train result prediction, followed by Text-to-SQL RL over complete trajectories. Our evaluation uses one final SQL per question, without voting or candidate selection; the solver predicts intermediate relations without execution feedback during decoding. Training and decoding configurations appear in Appendix B.1.

4.2 Main results and ablations

Main results. In Table 1, SimSquirrel-8B obtains 71.12% EX on BIRD dev, compared with 42.37% for Qwen3-8B, a difference of 28.75 percentage points. Among models of similar size, its score exceeds those of OmniSQL-7B and Arctic-Text2SQL-R1-7B by 7.22 and 2.22 points, respectively. The larger Arctic-Text2SQL-R1-32B and Databricks-RLVR-32B models score 70.50% and 70.80%, respectively, within 0.62 points of the 8B model. On Spider 2.0 SQLite, SimSquirrel-14B reaches 21.48% EX, compared with 6.67% for Qwen3-14B.

Table 1: Execution accuracy (EX) on BIRD dev and Spider 2.0 SQLite. Appendix B.2 distinguishes our evaluations from published baseline scores. Reasoning (✓) indicates whether reasoning is enabled at inference.

Model	Size	Reasoning	BIRD dev	Spider 2.0 SQLite
Closed-source Models				
GPT-4o (OpenAI, 2024)	–		58.80	17.0
Gemini 2.5 Pro (Comanici et al., 2025)	–	✓	64.80	24.44
GPT-5.6-luna (high) (OpenAI, 2026b)	–	✓	59.91	21.48
Larger Models (≥30B)				
OmniSQL-32B (Li et al., 2025)	32B	✓	64.50	14.8
Arctic-ExCoT-32B (Zhai et al., 2025)	32B	✓	68.32	–
Arctic-Text2SQL-R1-32B (Yao et al., 2025)	32B	✓	70.50	16.3
Databricks-RLVR-32B (Ali et al., 2025)	32B	✓	70.80	–
Small & Medium Open-source Models (4B–14B)				
Qwen3-4B (Yang et al., 2025)	4B		41.26	2.22
Llama-3.1-8B-Instruct (Grattafiori et al., 2024)	8B		42.00	1.5
Qwen3-8B (Yang et al., 2025)	8B		42.37	2.22
Qwen3-14B (Yang et al., 2025)	14B		49.87	6.67
Qwen2.5-7B-Instruct (Qwen et al., 2025)	7B		46.90	5.2
Qwen2.5-Coder-7B-Instruct (Hui et al., 2024)	7B		50.90	4.4
SFT CodeS-7B (Li et al., 2024)	7B		57.17	–
OmniSQL-7B (Li et al., 2025)	7B	✓	63.90	13.3
Arctic-Text2SQL-R1-7B (Yao et al., 2025)	7B	✓	68.90	15.6
Arctic-Text2SQL-R1-14B (Yao et al., 2025)	14B	✓	70.10	16.3
SIMSQUIRREL (Qwen3-4B)	4B		68.58	14.81
SIMSQUIRREL (Qwen3-8B)	8B		**71.12**	17.78
SIMSQUIRREL (Qwen3-14B)	14B		**73.86**	21.48

Ablation studies. We examine execution training, MiniDB construction, and the Stage 2 reward using Qwen3-8B. We compare Stage 2 initialized from the base model with Stage 2 initialized from the trained executor to assess the contribution of execution pre-training to SQL generation. Appendices C.1 and C.2 give the comparison settings and metric definitions.

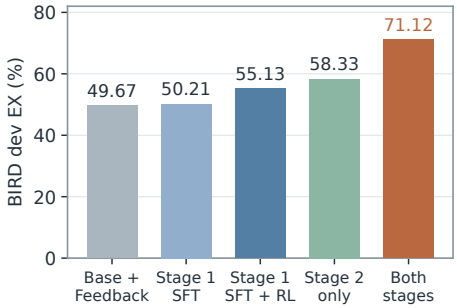
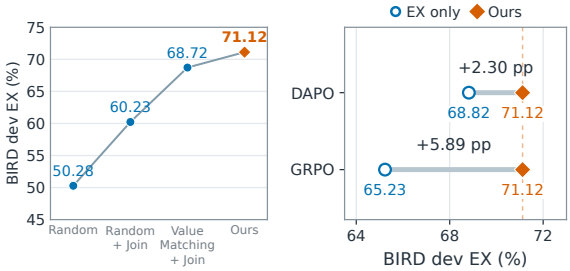


Figure 3: BIRD dev EX across training configurations of Qwen3-8B, with the same inference protocol and MiniDBs.

Training stages. Figure 3 uses the same feedback protocol and MiniDBs for all configurations, including Qwen3-8B without additional training (Base + Feedback). Execution SFT raises EX from 49.67% to 50.21%, and execution RL further improves it to 55.13%. Stage 2 alone reaches 58.33%, whereas the complete method reaches 71.12%. The full method therefore exceeds Stage 1 alone by 15.99 points and Stage 2 alone by 12.79 points. Section 4.3 examines how these changes relate to execution prediction accuracy.

MiniDB construction. Figure 4a compares four construction methods using SimSquirrel-8B. Random samples complete rows independently; Random + Join adds rows linked to random seeds through primary–foreign key relations. Value Matching + Join instead selects seeds by matching question and evidence values, with the same join-completion rules. These comparisons are intended to distinguish the effects of seed selection and relational completion. Ours uses the frozen complete-system construction and reaches 71.12% EX.

Stage 2 reward. With execution accuracy on the original database as the only reward, DAPO reaches 68.82% EX and GRPO reaches 65.23% (Figure 4b). Ours reaches 71.12%, improving on these baselines by 2.30 and 5.89 points. It replaces standard DAPO dynamic sampling with the procedure in Section 3.6: groups with identical original rewards are rescored on shared synthesized databases and retained when the updated rewards distinguish candidates. Rescoring gives these groups nonzero relative advantages.



(a) MiniDB construction. (b) Stage 2 reward.

Figure 4: BIRD dev EX for (a) MiniDB construction with SimSquirrel-8B and (b) Stage 2 reward variants. Segments show gains over EX-only baselines.

Diagnostic quality and cost. Figure 5 complements downstream EX with measures of query-disagreement preservation, mutation detection, predicate and column coverage, and token savings, using SimSquirrel-8B. These metrics distinguish compact contexts that retain diagnostic cases from contexts that merely contain fewer rows. Predicate coverage requires both satisfying and non-satisfying rows for each covered predicate. Appendix C.2 defines all six metrics and their scoring conventions.

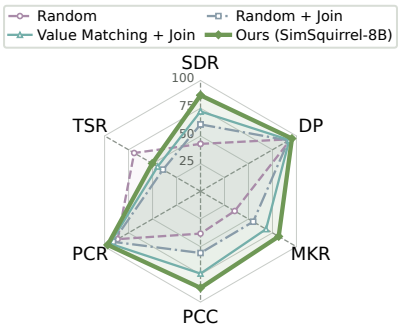


Figure 5: MiniDB metrics (0–100%; higher is better). Token savings are relative to a 6,000-token reference.

4.3 From execution prediction to SQL generation

Figure 6 separates improvements in execution prediction from their use in SQL generation. Stage 1 raises execution prediction accuracy from 32.82% to 76.44%, but BIRD dev EX increases only from 49.67% to 55.13%. Stage 1 trains result prediction for supplied SQL, without directly optimizing which local queries to construct from the question. Even a correctly predicted relation can belong to a query that does not answer the question; generation also requires checking whether the chosen operations satisfy its requirements. Stage 2 optimizes complete generation trajectories using the execution reward of the final SQL. Adding Stage 2 raises EX to 71.12% while largely preserving prediction accuracy at 76.23%, consistent with learning to use predicted relations during query construction. Without Stage 1, Stage 2 reaches only 35.62% prediction accuracy and 58.33% EX. These results support developing execution prediction before optimizing its use; the comparison also includes the additional compute used by Stage 1. Each metric uses a fixed evaluation set across configurations, which share the inference protocol and MiniDB construction (Appendix C.1).

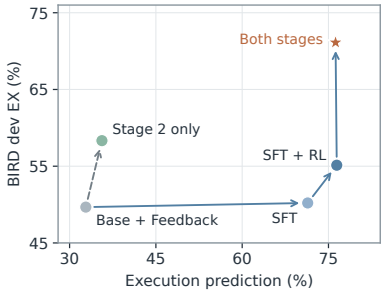


Figure 6: Execution prediction and BIRD dev EX across training stages. The dashed arrow shows training Stage 2 directly from Base + Feedback.

5 Conclusion

We introduced SimSquirrel, which predicts intermediate SQL results on diagnostic MiniDBs to guide query construction without external execution feedback. Shared synthesized databases distinguish candidates with identical original execution rewards during training. SimSquirrel-8B achieves 71.12% EX on BIRD dev. Ablations show higher EX from combining execution training and trajectory optimization, with little change in prediction accuracy during the second stage. The multi-database configuration exceeds EX-only DAPO by 2.30 points, although this comparison changes both reward values and training-group selection.

AI USE STATEMENT

We used AI tools to polish the language of the manuscript and assist with code development. All final writing decisions and technical content were verified by the authors.

REFERENCES

Alnur Ali, Ashutosh Baheti, Jonathan Chang, Ta-Chung Chi, Brandon Cui, Andrew Drozdov, Jonathan Frankle, Abhay Gupta, Pallavi Koppol, Sean Kulinski, Jonathan Li, Dipendra Misra, Krista Opsahl-Ong, Jose Javier Gonzalez Ortiz, Matei Zaharia, and Yue Zhang. A state-of-the-art SQL reasoning model using RLVR, 2025. URL https://arxiv.org/abs/2509.21459.

Gheorghe Comanici et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025. URL https://arxiv.org/abs/2507.06261.

Yaxun Dai, Baolin Sun, Junying Wang, Pengfei Wang, Yingqi Gao, Xuemei Dong, Mengdie Chu, Xiang Qi, and Pingfu Chao. Every step counts: Step-level credit assignment for tool-integrated text-to-SQL, 2026a. URL https://arxiv.org/abs/2605.04719.

Yaxun Dai, Wenxuan Xie, Xialie Zhuang, Tianyu Yang, Ziyi Liu, Haiqin Yang, Yiying Yang, Yuhang Zhao, Pingfu Chao, and Wenhao Jiang. ReEx-SQL: Reasoning with execution-aware reinforcement learning for text-to-SQL. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 824–847. Association for Computational Linguistics, 2026b. doi: 10.18653/v1/2026.acl-long.35. URL https://aclanthology.org/2026.acl-long.35/.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://huggingface.co/deepseek-ai/DeepSeek-V4-Flash.

Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. ReFoRCE: A text-to-SQL agent with self-refinement, consensus enforcement, and column exploration, 2025. URL https://arxiv.org/abs/2502.00675.

FAIR CodeGen team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, Juliette Decugis, Zacharias V. Fisches, François Fleuret, Fabian Gloeckle, Alex Gu, Michael Hassid, Daniel Haziza, Badr Youbi Idrissi, Christian Keller, Rahul Kindi, Hugh Leather, Gallil Maimon, Aram Markosyan, Francisco Massa, Pierre-Emmanuel Mazaré, Vegard Mella, Naila Murray, Keyur Muzumdar, Peter O'Hearn, Matteo Pagliardini, Dmitrii Pedchenko, Tal Remez, Volker Seeker, Marco Selvi, Oren Sultan, Sida Wang, Luca Wehrstedt, Ori Yoran, Lingming Zhang, Taco Cohen, Yossi Adi, and Gabriel Synnaeve. CWM: An open-weights LLM for research on code generation with world models, 2025. URL https://arxiv.org/abs/2510.02387.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning, 2024. URL https://arxiv.org/abs/2410.02089.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. The Llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 16568–16621. PMLR, 2024. URL https://proceedings.mlr.press/v235/gu24c.html.

Mohammadamin Habibollah and Davood Rafiei. SynSQL: Synthesizing relational databases for robust evaluation of text-to-SQL systems, 2026. URL https://arxiv.org/abs/2604.27261.

Binyuan Hui, Jian Yang, Zeyu Cui, et al. Qwen2.5-Coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

Kimi Team et al. Kimi K3: Open frontier intelligence, 2026. URL https://arxiv.org/abs/2607.24653.

Rocky Klopfenstein, Yang He, Andrew Tremante, Yuepeng Wang, Nina Narodytska, and Haoze Wu. SpotIt: Evaluating text-to-SQL evaluation with formal verification, 2025. URL https://arxiv.org/abs/2510.26840. Accepted at ICLR 2026.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows, 2024. URL https://arxiv.org/abs/2411.07763.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. CodeS: Towards building open-source language models for text-to-SQL, 2024. URL https://arxiv.org/abs/2402.16347.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. OmniSQL: Synthesizing high-quality text-to-SQL data at scale, 2025. URL https://arxiv.org/abs/2503.02240.

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

Leo Luo, Haining Xie, Siqi Shen, Zhipeng Ma, Rui Ling, Hang Xu, Hefeng Jiang, Dingwei Chen, Yang Li, Peng Chen, and Jie Jiang. SIRIUS-SQL: Anchoring multi-candidate text-to-SQL in execution feedback, 2026. URL https://arxiv.org/abs/2606.01246.

Gallil Maimon, Ori Yoran, Felix Kreuk, Michael Hassid, Gal Cohen, Pierre Chambon, and Yossi Adi. Self-execution simulation improves coding models, 2026. URL https://arxiv.org/abs/2604.03253.

Md Mahadi Hasan Nahid, Davood Rafiei, Weiwei Zhang, and Yong Zhang. Rethinking schema linking: A context-aware bidirectional retrieval approach for text-to-SQL. In *Findings of the Association for Computational Linguistics: EACL 2026*, pp. 4516–4546. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-eacl.236. URL https://aclanthology.org/2026.findings-eacl.236/.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. NExT: Teaching large language models to reason about code execution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 37929–37956. PMLR, 2024. URL https://proceedings.mlr.press/v235/ni24a.html.

OpenAI. GPT-4o system card, 2024. URL https://arxiv.org/abs/2410.21276.

OpenAI. Introducing GPT-5.4, 2026a. URL https://openai.com/index/introducing-gpt-5-4/.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition, 2026b. URL https://openai.com/index/gpt-5-6/.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026. URL https://qwen.ai/blog?id=qwen3.6-27b.

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

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework, 2024. URL https://arxiv.org/abs/2409.19256.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. CHESS: Contextual harnessing for efficient SQL synthesis, 2024. URL https://arxiv.org/abs/2405.16755.

Yuan Tian and Tianyi Zhang. PV-SQL: Synergizing database probing and rule-based verification for text-to-SQL agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 25827–25845. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.1286. URL https://aclanthology.org/2026.findings-acl.1286/.

Dongyu Wang, Jingyu Li, Lan Zhang, Ganggang Yu, and Liang Huang. VET: Verifiable execution tracing for reliable text-to-SQL generation. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 30867–30883. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.1544. URL https://aclanthology.org/2026.findings-acl.1544/.

An Yang et al. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Zhewei Yao, Guoheng Sun, Lukasz Borchmann, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. Arctic-Text2SQL-R1: Simple rewards, strong reasoning in text-to-SQL, 2025. URL https://arxiv.org/abs/2505.20315v1.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 113222–113244. Curran Associates, Inc., 2025. doi: 10.52202/085713-3775. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/a4277440d50f1f15d2cb4c14f7e0c0d2-Paper-Conference.pdf.

Bohan Zhai, Canwen Xu, Yuxiong He, and Zhewei Yao. Optimizing reasoning for text-to-SQL with execution feedback. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 19206–19218, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.982. URL https://aclanthology.org/2025.findings-acl.982/.

Ruiqi Zhong, Tao Yu, and Dan Klein. Semantic evaluation for text-to-SQL with distilled test suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 396–411. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.emnlp-main.29. URL https://aclanthology.org/2020.emnlp-main.29/.

Yuxin Zuo, Zikai Xiao, Li Sheng, Fei Huang, Jianhong Tu, Yuxuan Liu, Tianyi Tang, Xiaomeng Hu, Yang Su, Qingfeng Lan, et al. Qwen-AgentWorld: Language world models for general agents, 2026. URL https://arxiv.org/abs/2606.24597.

A Analysis of diagnostic MiniDB selection

A.1 Proof of Theorem 1

For each $\widehat{y} \in \mathcal{Y}_x^-$, define

$$f_{\widehat{y}}(A) = \mathbb{I}[\mathcal{U}_x(\widehat{y}) \cap C_x(A) \neq \emptyset], \qquad d_{\widehat{y}}(A) = \mathbb{I}[\mathrm{Exec}_\tau(\widehat{y}, D_x^{\mathrm{mini}}(A)) \neq \mathrm{Exec}_\tau(y_x^\star, D_x^{\mathrm{mini}}(A))]. \tag{11}$$

If $f_{\widehat{y}}(A) = 1$, the selected MiniDB contains a stable case for $\widehat{y}$. By the definition of stability and the compiler assumption in Theorem 1, $d_{\widehat{y}}(A) = 1$. Hence $f_{\widehat{y}}(A) \leq d_{\widehat{y}}(A)$ pointwise, and averaging over $\mathcal{Y}_x^-$ proves $F_x(A) \leq J_x(A)$.

The empty selection covers no case, hence $F_x(\emptyset) = 0$. If $A \subseteq A'$, then $C_x(A) \subseteq C_x(A')$; any error covered by A remains covered by A'. This proves monotonicity. For submodularity, fix $A \subseteq A'$ and $b \notin A'$. Adding b changes $f_{\widehat{y}}$ from zero to one only when $C_x(b)$ contains a stable case for $\widehat{y}$ that the current selection has not covered. The larger set A' may already cover errors left uncovered by A, so

$$f_{\widehat{y}}(A \cup \{b\}) - f_{\widehat{y}}(A) \geq f_{\widehat{y}}(A' \cup \{b\}) - f_{\widehat{y}}(A'). \tag{12}$$

Averaging these coverage indicators preserves the inequality and proves submodularity.

Tightness. The lower bound is exact on the specified candidate family if the case library is exhaustive: whenever $d_{\widehat{y}}(A) = 1$, the selected groups instantiate a case in $\mathcal{U}_x(\widehat{y})$. The main result does not assume exhaustiveness. Stability depends on the operator. For `EXCEPT`, `NOT EXISTS`, and `ORDER BY` with `LIMIT`, a case must contain enough rows to close the relevant comparison. Cases without this closure provide empirical evidence but no certificate.

A.2 Why uniform sampling misses rare diagnostic cases

Let $N = |\mathcal{B}_x|$ and let $\mathcal{U}_x = \bigcup_{b \in \mathcal{B}_x} C_x(b)$ be the cases available in the candidate groups. For $u \in \mathcal{U}_x$, define the groups that instantiate it and its prevalence as

$$\mathcal{B}_{x,u} = \{b \in \mathcal{B}_x : u \in C_x(b)\}, \qquad p_u = \frac{|\mathcal{B}_{x,u}|}{N}. \tag{13}$$

If a baseline samples k groups uniformly without replacement, then

$$\Pr[u \text{ is missed}] = \frac{\binom{N-|\mathcal{B}_{x,u}|}{k}}{\binom{N}{k}} \leq (1 - p_u)^k \leq e^{-kp_u}. \tag{14}$$

Let $p_{\min} = \min_{u \in \mathcal{U}_x} p_u$. A union bound gives

$$\Pr[\text{some diagnostic case is missed}] \leq |\mathcal{U}_x| e^{-kp_{\min}}. \tag{15}$$

A sufficient condition for covering all cases with probability at least $1 - \delta$ is

$$k \geq \frac{1}{p_{\min}} \log \frac{|\mathcal{U}_x|}{\delta}. \tag{16}$$

The rarest case controls the sample size required by uniform reduction. Our selector instead retrieves a linked group that instantiates the requested case.

A.3 Finite-sample interpretation

Under an i.i.d. error model, the finite candidate set in the main text is an empirical sample from a fixed population of plausible SQL errors. Let Π_x denote this population distribution and define

$$F_{\Pi_x}(A) = \Pr_{\widehat{y} \sim \Pi_x}[\mathcal{U}_x(\widehat{y}) \cap C_x(A) \neq \emptyset]. \tag{17}$$

For n i.i.d. training errors $\widehat{y}_1, \ldots, \widehat{y}_n \sim \Pi_x$, Equation 3 is the empirical objective

$$\widehat{F}_n(A) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}[\mathcal{U}_x(\widehat{y}_i) \cap C_x(A) \neq \emptyset]. \tag{18}$$

Let $\mathcal{A}_B = \{A \subseteq \mathcal{B}_x : \sum_{b \in A} c(b) \le B\}$. Fix this finite feasible family and the case mapping before sampling the errors. Hoeffding's inequality followed by a union bound gives, with probability at least $1 - \delta$,

$$\sup_{A \in \mathcal{A}_B} |\widehat{F}_n(A) - F_{\Pi_x}(A)| \le \sqrt{\frac{\log(2|\mathcal{A}_B|/\delta)}{2n}} =: \epsilon_n. \tag{19}$$

If the compiler returns an α-approximate maximizer of $\widehat{F}_n$ and $A^\star_\Pi$ maximizes F_{Π_x} over $\mathcal{A}_B$, then on the same event,

$$F_{\Pi_x}(\widehat{A}) \ge \alpha F_{\Pi_x}(A^\star_\Pi) - (1 + \alpha)\epsilon_n. \tag{20}$$

The bound separates optimization error from error-sample estimation error. It assumes a fixed candidate distribution and a case mapping chosen independently of the sample, unless the capacity of the mapping class is included in the analysis. The result does not require each test question to supply many i.i.d. errors.

B Experimental settings

B.1 Experimental settings

The 8B configuration uses verl (Sheng et al., 2024), eight H20 GPUs, full-parameter FSDP, BF16 computation, and gradient checkpointing. All stages use AdamW, with the main training parameters listed in Table 2.

Parameter	Stage 1 SFT	Stage 1 RL	Stage 2 RL
Training examples/questions	44,519	6,397	6,554
Learning rate	5×10^{-6}	10^{-6}	10^{-6}
Examples per SFT update	128	–	–
Questions generated/update	–	32 / 8 retained	8 / 4 retained
Responses per question	–	8	8
Rollout temperature / top-p	–	1.0 / 1.0	1.0 / 1.0
Prompt / response tokens	12,288 / 2,048	12,288 / 2,048	12,288 / 4,096
Optimizer updates	348 (1 epoch)	700	700

Table 2: Training parameters.

SFT uses a constant learning rate, AdamW betas $(0.9, 0.999)$, gradient clipping at 1.0, a 16,384-token sequence limit, and seed 20260729. One epoch draws 44,544 weighted samples in 348 updates.

Execution RL uses GRPO (Shao et al., 2024), DAPO filtering (Yu et al., 2025) with at most 20 generation batches, clipping bounds 0.2/0.28, and seed 20260714. Reward weights for exact match, row F1, column F1, cardinality, and format are 0.70/0.12/0.10/0.06/0.02; KL, entropy, and overlength penalties are disabled.

Figure 7 shows the entropy and policy-gradient loss during Stage 2 training of SimSquirrel-8B.

Inference configuration. Our evaluations use one solver response per question, with no voting or agent framework. Inputs contain the question, permitted evidence, and full schema; SimSquirrel additionally receives a reference-free diagnostic MiniDB. The BIRD feedback runner uses greedy decoding, context/output limits of 16,384/4,096 tokens, seed 20260729, and final/local SQL timeouts of 30/10 seconds.

Spider 2.0 SQLite. We use OmniSQL's released data and evaluator (Li et al., 2025). Baseline inputs retain its schema descriptions, example values, and external knowledge. SimSquirrel uses full schemas and fixed MiniDBs built by frozen Qwen3-8B without reference SQL, following the Stage 2 prompt in Appendix E. Qwen3 baselines and SimSquirrel use greedy decoding with native thinking disabled and output limits of 4,096 and 16,384 tokens, respectively. Gemini and GPT-5.6-luna use temperature 1 and a 16,384-token completion budget including reasoning.

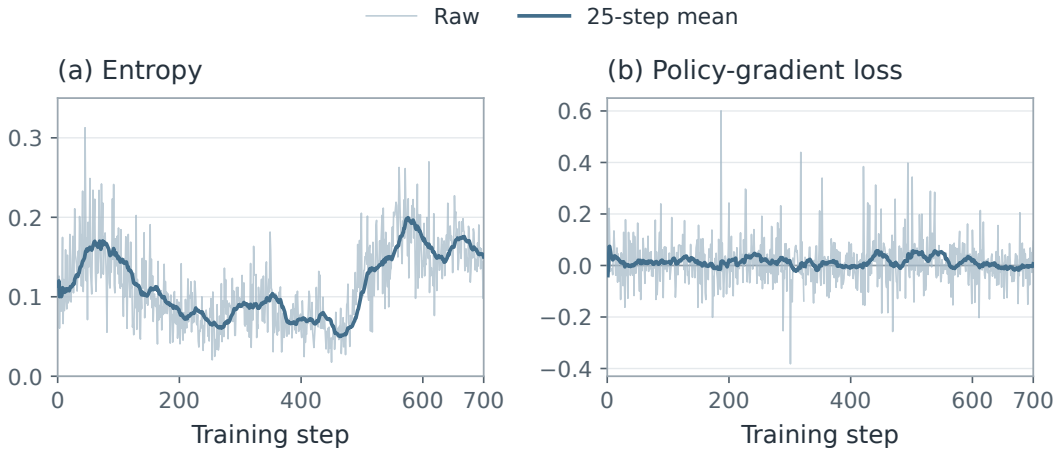


Figure 7: Stage 2 training curves for SIMSQUIRREL-8B.

B.2 BASELINES

The closed-source model group in Table 1 includes GPT-4o (OpenAI, 2024), Gemini 2.5 Pro (Comanici et al., 2025), and GPT-5.6-luna (high) (OpenAI, 2026b). General-purpose open baselines include Qwen3 at 4B, 8B, and 14B (Yang et al., 2025), Llama-3.1-8B-Instruct (Grattafiori et al., 2024), Qwen2.5-7B-Instruct (Qwen et al., 2025), and Qwen2.5-Coder-7B-Instruct (Hui et al., 2024).

SQL-specialized baselines include SFT CodeS-7B (Li et al., 2024), OmniSQL-7B/32B (Li et al., 2025), Arctic-ExCoT-32B (Zhai et al., 2025), Arctic-Text2SQL-R1-7B/14B/32B (Yao et al., 2025), and Databricks-RLVR-32B (Ali et al., 2025).

Figure 1(b) additionally includes Qwen3.6-27B (Qwen Team, 2026), DeepSeek-V4-Flash (DeepSeek-AI, 2026), and Kimi-K3 (Kimi Team et al., 2026).

Result sources. For BIRD dev, the closed-source and Qwen3 baseline scores are from our evaluations on all 1,534 questions; Qwen3 uses schema-only inputs with native thinking disabled. Llama-3.1-8B-Instruct, Qwen2.5-7B-Instruct, Qwen2.5-Coder-7B-Instruct, and OmniSQL-7B/32B scores are taken from the greedy-decoding evaluation in OmniSQL (Li et al., 2025). Scores for CodeS, Arctic-ExCoT, Arctic-Text2SQL-R1, and Databricks-RLVR are taken from their cited reports or the public BIRD leaderboard. For Spider 2.0 SQLite, we evaluate Qwen3, Gemini 2.5 Pro, and GPT-5.6-luna; the remaining baseline scores are from Table 7 of Yao et al. (2025).

Evaluation settings. Our single-response inference configuration is described in Appendix B.1. Published scores retain their source data versions and evaluation settings: OmniSQL uses schema descriptions augmented with database values, while the reported CodeS result uses beam search with four candidates and selects the first executable SQL. These entries provide comparisons with published systems; the controlled comparisons between training stages use our evaluation protocol.

B.3 DATASETS

We retain 6,554 training and 731 held-out ID questions from the BIRD training split (Li et al., 2023); all derived examples remain in their source partition. Stage 1 exports contain 44,519 SFT examples (20,004 local queries and 24,515 mutations), 6,397 reference-query RL examples, and 713 shared execution-validation tasks. SFT uses weighted sampling with replacement: local/mutation mass is $2:1$, reference/predicted predecessor mass is $3:1$ within local examples, and the weighted empty-result fraction is 15%.

Stage 1 reuses fixed oracle MiniDBs selected using reference SQL, targeting 12 real rows per table with a cap of 20. The reference-free Text-to-SQL builder uses frozen Qwen3-8B with a profile of up to 200 rows per table; its compiler limits are 16 tables, 64 columns and 14 rows per table before

projection, and an estimated 6,500-token MiniDB budget. The current feedback evaluator freezes the initial V1.5 artifacts without online expansion; builder context/output limits are 32,768/3,000 tokens.

B.4 METRICS

Execution accuracy (EX) evaluates final SQL on the original databases. The BIRD scorer compares unordered row sets. For Spider 2.0 SQLite, OmniSQL's scorer compares outputs with reference result files using each question's column-selection and ordering rules. Invalid SQL, execution failures, and timeouts count as incorrect. Stage 1 exact relation accuracy instead checks columns, cardinality, duplicate multiplicities, and required row order on the same 713 ID tasks. The Stage 2 reward uses a separate comparator preserving multiplicity and task-dependent ordering.

The MiniDB ablation additionally reports strict disagreement recall (SDR), disagreement precision (DP), mutation kill rate (MKR), predicate contrast coverage (PCC), provided-column recall (PCR), and token saving rate (TSR). Appendix C.2 defines these measures and their aggregation conventions. They evaluate diagnostic preservation, schema coverage, and context cost separately from final-query EX.

C ABLATION SETTINGS AND METRIC DEFINITIONS

C.1 COMPARISON SETTINGS

The training-stage comparison uses Qwen3-8B as the base model. Except for the schema-only configuration, all variants share the inference protocol and MiniDB construction. Execution prediction and BIRD dev EX are computed on their respective fixed evaluation sets; the two metrics do not share an evaluation denominator. Execution prediction uses the 713-task set described in Appendix B.4, whereas BIRD dev EX scores final SQL against the original development databases.

For the construction comparison, we keep SimSquirrel-8B fixed and change only the inference-time MiniDB construction. Random samples complete rows independently from each table without relational completion. Random + Join starts with random seed rows and supplements them through available primary–foreign key relations. Value Matching + Join matches cells against the question and permitted evidence, selects the corresponding rows, and uses the same completion rules as Random + Join. Ours uses the frozen construction implementation and configuration with SimSquirrel-8B. Reference SQL is used for post-hoc diagnostic evaluation, not as a construction input.

The Stage 2 comparison uses execution accuracy on the original database as the sole reward for two baselines: DAPO with standard dynamic sampling enabled and GRPO with it disabled. Ours disables standard DAPO dynamic sampling and instead rescores otherwise indistinguishable rollout groups on shared synthesized databases before deciding which groups to retain.

C.2 MINIDB QUALITY AND RESOURCE METRICS

We retain the notation of Section 3.3: D is the full database, $D_x^{\mathrm{mini}}(A)$ is the selected MiniDB, and $y_x^\star$ is the reference SQL. Results are compared using Exec_τ. Below, $|\cdot|$ denotes set size. Set elements retain their evaluation-instance identifiers, so counts pool across instances without merging identical queries or columns from different examples. Ratios are displayed as percentages in Figure 5.

Disagreement preservation: SDR and DP. Let $\mathcal{E}_{\mathrm{full}}$ be the set of query pairs that disagree on D, and $\mathcal{E}_{\mathrm{mini}}$ the set that disagree on the MiniDB and execute successfully on both databases. Strict disagreement recall (SDR) and disagreement precision (DP) are

$$\mathrm{SDR} = \frac{|\mathcal{E}_{\mathrm{full}} \cap \mathcal{E}_{\mathrm{mini}}|}{|\mathcal{E}_{\mathrm{full}}|}, \qquad \mathrm{DP} = \frac{|\mathcal{E}_{\mathrm{full}} \cap \mathcal{E}_{\mathrm{mini}}|}{|\mathcal{E}_{\mathrm{mini}}|}. \tag{21}$$

MiniDB execution failures remain in the SDR denominator but do not count as preserved disagreements. When each candidate is paired with $y_x^\star$ and all executions are valid, per-instance SDR equals $J_x(A)$ in Equation 2.

Mutation kill rate (MKR). Let $\mathcal{V}$ contain valid mutants and $\mathcal{K} \subseteq \mathcal{V}$ those distinguished from the reference on the MiniDB. Then

$$\mathrm{MKR} = \frac{|\mathcal{K}|}{|\mathcal{V}|}. \tag{22}$$

A valid mutant executes successfully and differs from $y_x^\star$ on D, placing it in $\mathcal{Y}_x^-$ when included in the candidate family. Membership in $\mathcal{K}$ requires successful, differing MiniDB executions. All methods use the same mutation generator and result comparator.

Predicate contrast coverage (PCC). Let $\mathcal{P}$ contain eligible single-table predicates from the reference query's top-level `WHERE` clause, and $\mathcal{P}_{\mathrm{cov}} \subseteq \mathcal{P}$ those with contrasting MiniDB rows. We define

$$\mathrm{PCC} = \frac{|\mathcal{P}_{\mathrm{cov}}|}{|\mathcal{P}|}. \tag{23}$$

Coverage requires the predicate's columns and both a satisfying and a non-satisfying row; the latter includes SQL `UNKNOWN` outcomes. PCC measures predicate-level contrast, not the stable-case objective $F_x(A)$ in Equation 3.

Provided-column recall (PCR). Let $\mathcal{C}_{\mathrm{gold}}$ be the table-qualified columns referenced by $y_x^\star$ and $\mathcal{C}_{\mathrm{mini}}$ those present in the MiniDB schema. Then

$$\mathrm{PCR} = \frac{|\mathcal{C}_{\mathrm{gold}} \cap \mathcal{C}_{\mathrm{mini}}|}{|\mathcal{C}_{\mathrm{gold}}|}. \tag{24}$$

PCR measures column availability, independently of whether the retained rows preserve query behavior.

Token saving rate (TSR). For total MiniDB and reference-context token counts T_{mini} and T_{ref},

$$\mathrm{TSR} = 1 - \frac{T_{\mathrm{mini}}}{T_{\mathrm{ref}}}. \tag{25}$$

Counts use the same tokenizer and serialization convention, with each instance's reference fixed across methods. TSR measures serialized context cost, corresponding to the resource constraint in Section 3.3. This cost need not equal $\sum_{b \in A} c(b)$ because serialization can remove shared content. TSR uses a fixed reference of 6,000 tokens. TSR excludes generation cost; negative values indicate a MiniDB context longer than the reference.

SDR and DP are undefined when their denominators are zero. The scorer returns fallback values of zero for MKR and one for PCC and PCR when the denominator is zero. Radar axes are interpreted individually, not through polygon area.

D CASE STUDY: CORRECTING JOIN MULTIPLICITY

BIRD dev question 235 asks, "How many molecules are carcinogenic and have the bromine element?" The raw trace below uses its existing MiniDB to illustrate how FIX corrects the counting unit before final SQL generation.

1 Join and filter — LOCAL_EXECUTION 1

Subgoal. Retrieve carcinogenic molecules and their matching bromine atoms.

LOCAL SQL

```
SELECT m.molecule_id, a.atom_id
FROM molecule AS m
JOIN atom AS a
  ON m.molecule_id = a.molecule_id
WHERE m.label = '+'
  AND a.element = 'br'
ORDER BY m.molecule_id, a.atom_id;
```

RESULT RELATION R_1

molecule_id	atom_id
TR028	TR028_4
TR028	TR028_5
TR076	TR076_15
TR452	TR452_6

Four atom matches; TR028 occurs twice.

2 Count joined rows — CONTINUE

Subgoal. Count qualifying molecules using the number of joined rows.

LOCAL SQL

```
SELECT COUNT(*) AS molecule_count
FROM R1;
```

RESULT RELATION R_2

molecule_count
4

This counts matching atoms.

3 Correct the counting unit — FIX

Subgoal. R_1 contains two matches for TR028, with atom IDs TR028_4 and TR028_5. The question asks for molecules, so count distinct molecule IDs in R_1.

REVISED LOCAL SQL

```
SELECT COUNT(DISTINCT molecule_id)
  AS molecule_count
FROM R1;
```

RESULT RELATION R_3

molecule_count
3

R_1 retains the IDs needed for deduplication; R_2 contains only a count.

Final SQL — Apply the corrected plan

```
SELECT COUNT(DISTINCT m.molecule_id)
  AS molecule_count
FROM molecule AS m
JOIN atom AS a
  ON m.molecule_id = a.molecule_id
WHERE m.label = '+'
  AND a.element = 'br';
```

FULL DATABASE CHECK

Row count	42
Distinct molecules	14
Gold result	**14**

The final query computes the answer on the full database.

All three intermediate relations match offline SQLite execution on the MiniDB. These checks validate the displayed results; no engine feedback is part of the trace.

E PROMPTS

We provide the prompts for Stage 1 and Stage 2 below.

Execution prediction (Stage 1)

```
Predict the result of the given SQLite query on the supplied database.
Return a JSON object containing columns, rows, and nrows.
Output only the JSON object, without Markdown code fences.

[SCHEMA]
<SCHEMA_DDL>

[DATABASE]
<MINIDB_TABLES_JSON>

[SQL]
<SQL_QUERY>
```

Feedback v2 (Stage 2)

```
Solve QUESTION using FULL_SCHEMA, MINI_DATABASE, and EVIDENCE.
Return 1–4 LOCAL_EXECUTION blocks, followed by FINAL_SQL.

Rules:
1. Each block addresses a requirement of QUESTION or EVIDENCE.
   State it in SUBGOAL and implement it as one read-only SQLite query.
   Block N produces RN and may read source tables or earlier relations.
   Reuse earlier relations when sufficient; combine operations when useful.

2. Simulate each query's complete output in RESULT_RELATION.
   Preserve SQLite semantics, including NULLs, duplicates, joins,
   grouping, ordering, LIMIT, and type conversion.

3. Check the SQL and simulated results against QUESTION and EVIDENCE.
   From block 2, use PLAN_ACTION: CONTINUE for the next operation,
   or FIX for a concrete mismatch. For FIX, explain the mismatch and
   correction in SUBGOAL and implement the correction in LOCAL_SQL.
   Make corrections in a local block before FINAL_SQL.

4. MINI_DATABASE is non-exhaustive: missing rows do not prove absence
   from the full database. Filter values and IDs must be justified by
   QUESTION or EVIDENCE, not solely by the sample.

5. The last block must produce the final query's result shape and
   semantics on MINI_DATABASE. FINAL_SQL must implement this plan as
   one standalone read-only SQLite query over the full database,
   expanding all RN references.
```

Feedback v2 (Stage 2, continued)

```
[DATABASE]
<DATABASE_ID>

[FULL_SCHEMA]
<FULL_SCHEMA>

[MINI_DATABASE]
<MINI_DATABASE>

[EVIDENCE]
<EVIDENCE>

[QUESTION]
<QUESTION>

[RESPONSE_FORMAT]
Return only the following blocks. Number local blocks consecutively
from 1 and omit PLAN_ACTION in block 1.

[LOCAL_EXECUTION N]
PLAN_ACTION: CONTINUE or FIX
SUBGOAL: ...
LOCAL_SQL:
SELECT ...
RESULT_RELATION: {"columns":[...],"rows":[[...],...],"nrows":N}

[FINAL_SQL]
SELECT ...
```