SOTHIS-SQL: Anchoring Multi-Candidate Text-to-SQL in Execution Feedback

Anonymous ACL submission

Abstract

Text-to-SQL on complex schemas is unreliable on a single pass, so recent systems generate multiple SQL candidates and let voting filter out errors. Yet voting alone is not enough, because the multi-candidate recipe has three coupled weaknesses: 1) sampling more from a single generator produces increasingly redundant candidates, 2) existing pipelines apply one generic correction to every non-clean execution result, while runtime errors, timeouts, and empty results each indicate a different distance from correctness, and 3) existing selectors rely on a single angle such as result-majority voting or pairwise SQL comparison, missing what other angles would have caught. We present SOTHIS-SQL, which addresses all three weaknesses. A difficulty-smoothing RL recipe trains Sothis-32B to generate diverse executable SQL candidates, paired with a generalist LLM that fills in gaps left by the specialist. An execution-grounded lifecycle classifies each outcome and applies targeted repair before candidates re-enter the pool. A confidence-gated hybrid selector combines execution-result agreement with pairwise SQL-form judgment, escalating only near-tied cases to a deterministic structural check. SOTHIS-SQL reaches 75.88% on BIRD dev and 91.20% on SPIDER test. Two of three generalist pairings surpass Agentar-Scale-SQL, the strongest published multi-candidate system on BIRD dev. Our code is available at sothis-sql-C841.

1 Introduction

Real-world databases expose persistent limits of LLM-based text-to-SQL: on BIRD (Li et al., 2023), the strongest leaderboard system, GPT-5.5-xhigh, reaches only 72.55% execution accuracy on the test set.[1] Schema heterogeneity, implicit constraints, and ambiguous value references generate errors that no single model reliably avoids (Yu et al., 2018; Lei et al., 2025; Lin et al., 2020; Wang et al., 2025c).

[1]Official BIRD leaderboard

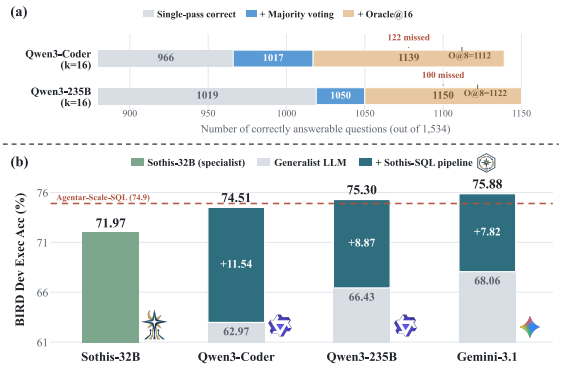

Figure 1: (a) Coverage-to-accuracy gap on BIRD dev. (b) SOTHIS-SQL paired with three generalist LLMs, with the red dashed line marking Agentar-Scale-SQL.

To mitigate this, recent work converges on a common recipe: sample multiple SQL candidates per question and let voting over execution results filter individual errors (Wang et al., 2023; Pourreza et al., 2025a; Liu et al., 2026; Wang et al., 2025b).

Figure 1(a) decomposes each single-generator candidate set into what majority voting recovers and what remains reachable, exposing three structural limits. These three limits trace to a common root: execution signals are systematically underused across training, repair, and selection.

L1: Diminishing-return sampling. A single generator's candidate set hits diminishing returns as sampling grows (Lee et al., 2025a; Pourreza et al., 2025a; Liu et al., 2026): on BIRD dev, doubling the candidate budget from k=8 to k=16 adds only 27–28 reachable questions per generator[2]. New samples largely repeat what the pool already contains, because each generator's outputs concentrate on the narrow region it was trained to prefer. Existing multi-candidate systems address this at inference time by adding more prompts, decoding routes, or parallel experts (Liu et al., 2026; Pourreza et al., 2025a; Wang et al., 2025b; Li et al.,

[2]A question is reachable at k (i.e., counted by Oracle@k, the bar length in Figure 1(a)) if at least one of its k sampled SQLs returns the correct execution result.

2025a), leaving the training objectives that shape each individual source's output distribution largely untouched.

L2: Wasted execution signal. The database itself returns rich diagnostic information whenever a candidate fails to produce a clean result (Mao et al., 2024; Xu et al., 2025; Cen et al., 2025; Shinn et al., 2023), yet existing pipelines apply one generic correction to every non-clean execution result. A runtime error exposes a syntactic or type mismatch, a timeout points to an inefficient join or subquery, and an empty result indicates a semantic predicate drift, each pointing to a different distance from correctness. Current pipelines either discard these candidates entirely or feed them to a single corrector with only a coarse label.

L3: Voting–oracle gap. Even when the correct query is already in the candidate set, voting still misses it (Pourreza et al., 2025a; Li et al., 2025a): 122 and 100 questions across the two single-generator pools in Figure 1(a) have correct SQL in the pool yet are misclassified by majority voting. The root cause is that selection requires joint judgment in two spaces: in the *result space*, different SQL statements may return identical outputs despite different logic, and in the *logic space*, the same correct intent admits many surface forms. Existing selectors operate in only one space (Pourreza et al., 2025a; Liu et al., 2026; Li et al., 2025a), leaving the other as a blind spot.

We address these three limits with SOTHIS-SQL, which redesigns candidate construction, repair, and selection together. **For L1**, SOTHIS-SQL trains Sothis-32B with a difficulty-smoothed two-stage RL recipe that broadens its candidate distribution while keeping each sample reward-validated, and pairs it with a generalist LLM for complementary coverage. **For L2**, an execution-grounded lifecycle classifies each outcome into a type (runtime error, timeout, or empty result) and applies targeted repair before re-admitting the candidate. **For L3**, a hybrid selector fuses tuple-level execution consensus with pairwise SQL quality assessment, and escalates only low-margin cases to a deterministic structural-support check. Our contributions are:

- A **difficulty-smoothed two-stage RL recipe** that trains Sothis-32B to produce a broad yet reward-validated candidate distribution (§3.3).
- An **execution-grounded lifecycle** with type-specific repair for runtime errors, timeouts, and empty results (§3.4).
- A **confidence-gated hybrid selector** that sums tuple-level consensus and pairwise SQL preference, with a deterministic AST gate as a zero-cost tie-breaker (§3.5).

SOTHIS-SQL reaches 75.88% on BIRD dev and 91.20% on SPIDER test. Across three generalist partners (Figure 1(b)), every pairing improves over its generalist baseline, and two surpass Agentar-Scale-SQL, the strongest published multi-candidate system on BIRD dev.

2 Related Work

Prompting and training-oriented generators. Early Text-to-SQL work studies schema-aware modeling, value grounding, intermediate representations, constrained decoding, and semantic evaluation (Bogin et al., 2019; Wang et al., 2020; Lin et al., 2020; Zhong et al., 2020; Gan et al., 2021; Qi et al., 2022). Recent LLM-based systems improve a single generator through prompt design, task decomposition, task alignment, and self-correction (Gao et al., 2024; Pourreza and Rafiei, 2023; Qu et al., 2024; Xie et al., 2024; Lee et al., 2025b). Another line adapts SQL-oriented models through synthetic data, open-source specialization, self-taught reasoning, execution-driven bootstrapping, reward modeling, self-play fine-tuning, and execution-grounded reinforcement learning (Yang et al., 2024; Li et al., 2024; He et al., 2025; Zhang et al., 2025a,b; Ma et al., 2025; Pourreza et al., 2025b; Yao et al., 2025).

Multi-candidate pipelines and selection. A second line of work moves beyond one-shot prediction by generating multiple candidates and organizing inference as a broader workflow. MAC-SQL uses multi-agent collaboration (Wang et al., 2025a), CHESS combines retrieval, schema pruning, generation, and verification (Talaei et al., 2024), Alpha-SQL formulates zero-shot generation as search (Li et al.), and Agentar-Scale-SQL emphasizes test-time orchestration (Wang et al., 2025b). Closer to selection, MCS-SQL aggregates candidates from multiple prompts (Lee et al., 2025a), PET-SQL applies two-round refinement with cross-consistency (Li et al., 2025c), CHASE-SQL and XiYan-SQL couple multi-path generation with pairwise candidate comparison as a final ranker after execution clustering (Pourreza et al., 2025a; Liu et al., 2026), and DeepEye-SQL introduces confidence-aware, execution-guided adjudication (Li et al., 2025a). Execution feedback has

also been used for question rewriting, refinement, test-driven self-correction, and decomposed automatic repair (Mao et al., 2024), while AST-level ranking has been explored for retrieval-augmented SQL generation (Shen et al., 2024).

Positioning. Most multi-candidate Text-to-SQL systems expand the candidate set through prompting, search, or multiple generators, then apply a final voter, comparator, or verifier. SOTHIS-SQL instead anchors every stage in execution feedback: an RLVR-trained specialist that produces broad yet executable candidates (L1), a typed-outcome lifecycle with re-execution before re-admission (L2), and a hybrid selector combining result-level, pairwise, and structural signals (L3).

3 Method

3.1 Problem Formulation

Given an input $x{=}(q, e, d)$ with question q, optional evidence e, and database d, SOTHIS-SQL predicts a SQL query y from a shared candidate pool $\mathcal{C}(x){=}\{c_1, \ldots, c_n\}$ populated by complementary generators: a SQL specialist trained with verifiable execution rewards and an off-the-shelf generalist LLM, detailed in §3.3. Let $\mathcal{E}(c, d)$ denote the execution outcome of candidate c on d. The pool is refined based on execution signals (§3.4), and the final query is selected via a confidence-gated cascade over the refined pool (§3.5).

3.2 System Overview

Figure 2 summarizes SOTHIS-SQL. Before generation, the system loads an offline schema prompt for every question and injects cached value hints. The schema prompt extends standard `CREATE TABLE` DDL with per-column comments and question-conditioned example values, so that ambiguous columns and unfamiliar value vocabularies are resolved before the generator sees them. Full construction details are in Appendix B.4.

3.3 Source-Complementary Candidate Construction

Pools drawn from a single generator are bounded by that generator's training distribution: sampling more candidates from the same model does not reliably surface hypotheses outside what its distribution prefers (L1, §1) (Lee et al., 2025a; Pourreza et al., 2025a). We therefore build the initial pool from two sources with complementary failure modes: a task-specialized generator G_{sp} and an external strong LLM G_{ge} serving as a generalist.

Specialist. G_{sp} is the task-specialized source of SQL hypotheses, trained on a code-pretrained base with reinforcement learning from verifiable execution rewards (RLVR), a natural choice for SQL where each candidate's correctness is decidable by execution. Our training-side design choices are a *difficulty-smoothed data mixture* and a *two-stage DAPO schedule*.

Difficulty-smoothed data synthesis. Two design choices shape the training signal. On the reward side, we replace the binary executable-match reward with a three-valued scalar that additionally rewards executable-but-wrong rollouts, distinguishing partial progress from outright failure. On the data side, we form a difficulty-smoothed corpus by combining multiple SQL sources, filtering out noisy gold queries, and pruning prompts that the baseline already solves trivially, so that training compute concentrates on instances where the policy still has room to improve. Reward values, filter stack, and pruning thresholds are in Appendix C.

Two-stage DAPO with threshold-gated entropy. We adopt DAPO (Yu et al., 2025), which builds on group-relative policy optimization for verifiable reasoning (Shao et al., 2024), as the policy-gradient backbone and augment it with a threshold-gated entropy regularizer that fires when policy entropy $H(\pi_\theta)$ falls below a target, counteracting entropy collapse without paying constant regularization:

$$\mathcal{L}_{\text{total}}(\theta) = \mathcal{L}_{\text{DAPO}}(\theta) + w_{\text{ent}}\, \mathcal{L}_{\text{ent}}(\theta), \qquad (1)$$

$$w_{\text{ent}} = \begin{cases} \beta, & H(\pi_\theta) < H_{\text{target}}, \\ 0, & \text{otherwise.} \end{cases} \qquad (2)$$

Training proceeds in two stages, with the objective shared. An *exploration* stage keeps rollout space wide on hard prompts, and an *exploitation* stage resumes from its checkpoint and anneals rollout width while preserving the dynamic-sampling constraint. Hyperparameters, execution sandbox, and the DAPO surrogate are in Appendix C.

Generalist and pool union. G_{ge} is an off-the-shelf general-purpose LLM that provides semantic coverage for long-tail phrasing and implicit constraints, prompted with 5-shot in-context demonstrations while G_{sp} uses its task-tuned prompt format. G_{sp} contributes high-consensus clusters of execution-equivalent rewrites, while G_{ge} contributes broader coverage of cases that G_{sp} may miss. At inference time, both generators are sampled under a fixed

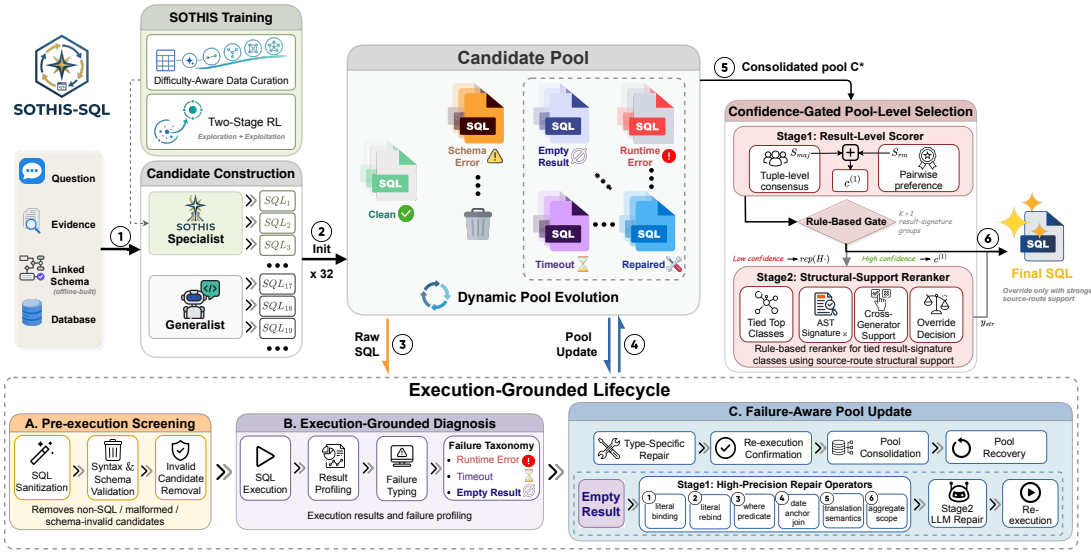

Figure 2: Overview of SOTHIS-SQL. A SQL specialist and a generalist LLM populate a shared candidate pool (①②; L1). An execution-grounded lifecycle screens, executes, and applies type-specific repair through Blocks A–C (③④; L2). A confidence-gated selector scores the consolidated pool and escalates only low-margin ties to a deterministic structural-support reranker (⑤⑥; L3).

budget, using multiple prompt routes within each source, and the resulting hypotheses are merged into a single pool. The construction objective is reduced error correlation rather than diversity alone: candidates from different sources should expose different failure modes, and downstream stages retain source and route metadata as provenance signals.

3.4 Execution-Grounded Candidate Lifecycle

Execution is used not only as a final verifier but also as the organizing signal for pool management. The lifecycle is organized into three blocks (Figure 2): (A) Pre-execution Screening, (B) Execution-Grounded Diagnosis, and (C) Failure-Aware Pool Update. The key design choice is to front-load deterministic screening, spend repair budget on execution-near hypotheses, and recycle unresolved failures as negative feedback only when the pool collapses. Appendix D details the high-level procedure, repair path, the empty-result operator catalog, and the post-repair rules.

Pre-execution screening. We first apply rule-based cleanup to common LLM artifacts and then filter the pool with lightweight syntax checks and `EXPLAIN`-backed schema validation. Candidates that fail this stage are removed from the pool before execution. This front-loaded filter is deliberate: obviously malformed SQL (such as comment-prefixed output, missing `FROM`/`SELECT` clauses, bracket or quote mismatches, or references to nonexistent tables and columns) should not consume downstream repair budget. Sanitization rules and validator details are fully specified in Appendix D.1.

Execution-grounded diagnosis. Surviving candidates are executed on the database and classified by a four-way taxonomy (`clean`, `runtime`, `timeout`, `empty`), where `empty` denotes valid queries returning zero rows, `runtime` indicates an engine exception, and `timeout` is caught by a per-query wall-clock budget. The taxonomy is load-bearing because different classes require structurally different repairs.

Failure-aware repair and pool consolidation. Every accepted rewrite is *immediately re-executed* before it can re-enter the pool, and each path is shaped by what the database outcome reveals. A runtime error pinpoints a concrete schema or syntax issue, and the engine's exception text is fed verbatim to the corrector. A timeout indicates a semantically plausible candidate with an inefficient query plan, so the rewrite preserves intent and restructures join order or subquery shape. An empty result indicates that the SQL ran but missed the answer (often value-binding errors or over-restrictive predicates), and six high-precision structural operators try local recovery before a diagnosis-conditioned LLM fallback. After repair, a deterministic projection-normalization pass rewrites noisy `SELECT` lists into cleaner forms, for example, dropping unrequested numbered variants such

as `Email2`/`Email3` or canonicalizing known column bundles. Full operator definitions and post-repair rules are in Appendix D.3–D.4.

Pool recovery. If post-repair filtering exhausts the pool, a recovery generator receives condensed negative feedback from unresolved candidates, rule-filtered SQLs, and matched failure tags, then regenerates a replacement pool (Appendix D.4).

3.5 Confidence-Gated Hybrid Selection

Given the consolidated pool $\mathcal{C}^{(*)}$ (Figure 2, right), the selector operates in two stages that correspond to the two boxes in the selection panel. **Stage 1** (Result-Level Scorer) computes a single coarse score $S_{\text{coarse}}{=}S_{\text{maj}}{+}S_{\text{rm}}$ per candidate and takes its argmax as the default winner $c^{(1)}$. A rule-based confidence gate then decides whether $c^{(1)}$ is sufficiently reliable. If not, **Stage 2** (Structural-Support Reranker) checks whether another tied class has strictly stronger cross-source structural support and overrides the default winner only in that case.

Stage 1: Result-level scorer. Let $\mathcal{C}^{+}{=}\{c \in \mathcal{C}^{(*)} \mid \delta(c){=}\texttt{clean}\}$ be the successfully executed candidates. Stage 1 combines two complementary result-level signals over $\mathcal{C}^{+}$ and picks the argmax of their sum.

Tuple-level consensus S_{maj}. We decompose each candidate's result set into cell-level triples $\mathcal{T}(c){=}\{(i, col_j, v_{ij})\}$, count the cross-pool frequency of each triple $f(t){=}|\{c'{\in}\mathcal{C}^{+} \mid t \in \mathcal{T}(c')\}|$, and average these frequencies with `null`-valued cells down-weighted:

$$S_{\text{maj}}(c) = \frac{1}{|\mathcal{T}(c)|} \sum_{t \in \mathcal{T}(c)} f(t)\, \mathbf{1}[v(t){\neq}\texttt{null}]. \quad (3)$$

Empty-result candidates receive $S_{\text{maj}}{=}0$ and are thus excluded from the consensus (derivation in Appendix E.1).

Position-debiased pairwise preference S_{rm}. We partition $\mathcal{C}^{+}$ into equivalence classes by result signature $\sigma(c){=}\text{frozenset}(\text{rows of } c)$.[3] We pick one representative per class and query an LLM judge on every pair of representatives in *both orderings* to cancel position bias (Zheng et al., 2023; Wang et al., 2024). Each representative accumulates a wins-minus-losses score broadcast to class members (Appendix E.3).

Stage 1 then returns

$$c^{(1)} = \arg\max_{c \in \mathcal{C}^{+}} \big[S_{\text{maj}}(c) + S_{\text{rm}}(c)\big], \quad (4)$$

[3]Result-signature grouping lets syntactically different SQL queries share evidence when they return the same tuple set.

with single-signal variants (*freq*, *majority voting*, *majority-refine*, *RM*) used in the RQ5 ablation defined in Appendix E.2. Besides the default merged score, the implementation also materializes an auxiliary consensus-refine statistic used only by the confidence gate. Appendix E gives the implementation-level definition.

Rule-based confidence gate. The gate is a strict ambiguity detector over Stage-1. Let $K{=}|[\mathcal{C}^{+}]_{\sigma}|$ be the number of result-signature classes. The gate fires if and only if the top-2 gaps under S_{coarse} (merge), an auxiliary consensus-refine score S_{ref} (refine), and S_{rm} (RM) are *all* zero, while $1{<}K{\leq}K_{\max}$ and the largest class has size $\leq T_{\max}$ (values in Appendix E). The zero-margin conjunction ensures escalation only when every Stage-1 signal is simultaneously tied, and the cardinality bounds exclude pools that are too fragmented or already dominated, both of which are well-handled by $c^{(1)}$. Empirically, this restricts Stage 2 to a small fraction of ambiguous questions.

Stage 2: Structural-support reranker. When the gate fires, a deterministic, LLM-free reranker is applied to the result-signature classes tied at the maximum Stage-1 coarse score. Let $\mathcal{G}_{\text{top}}$ denote this tied set, and let $\hat{c}_g$ be the representative candidate of class $g \in \mathcal{G}_{\text{top}}$. Each candidate is parsed into an AST-level structural signature $\kappa(c)$ capturing its logical skeleton: table set, projection patterns, projected/predicate columns, aggregation/order markers, and a coarse limit bucket (full feature list in Appendix E.5) (Shen et al., 2024). The reranker then measures *cross-source support*: how many distinct provenance groups $\text{srcgrp}(c)$ (generator–route pairs, with model-level fallback) contribute a candidate sharing the same structural signature, so that convergence from multiple independent sources outweighs support concentrated within one generator. For each tied class, let

$$\begin{aligned} \mathcal{S}_g &= \{c{\in}\mathcal{C}^{(*)} \mid \kappa(c){=}\kappa(\hat{c}_g)\}, \\ \text{supp}(g) &= \big|\{\text{srcgrp}(c) \mid c \in \mathcal{S}_g\}\big|, \\ g^{*} &= \arg\max_{g \in \mathcal{G}_{\text{top}}} \text{supp}(g), \end{aligned} \quad (5)$$

where $\mathcal{S}_g$ is the support set of candidates sharing the same structural signature as representative $\hat{c}_g$.

Final decision. Let $g^{(1)}$ be the result-signature class containing $c^{(1)}$. The structural branch overrides the Stage-1 winner only when $\texttt{gate}(x)$ holds and a competing tied class receives strictly stronger support, i.e. $\text{supp}(g^{*}) > \text{supp}(g^{(1)})$. Otherwise

$y{=}c^{(1)}$. The only LLM calls used in selection are those already required to compute S_{rm} in Stage 1. Algorithm 2 in Appendix E gives full procedure.

4 Experimental Setup

We evaluate on BIRD (Li et al., 2023) (1,534 dev questions, 95 databases) as the primary benchmark and SPIDER (Yu et al., 2018) (2,147 test questions) for generalization. All ablations run on BIRD dev. We report execution accuracy (EX) with a difficulty breakdown on BIRD. Evaluation-protocol details, including benchmark-specific execution matching rules, are in Appendix B.2.

All main runs use the default enriched schema context. Schema-variant ablations are reported in Appendix B.4.3. The specialist is Sothis-32B (Appendix C). Generalist partners are Qwen3-235B-Instruct-2507 (Team, 2025), Gemini-3.1-Pro-Preview, and Qwen3-Coder-30B-A3B-Instruct. Each question receives up to 32 candidates across generators. Serving, timeout, decoding, prompt-template, and backbone-matched baseline details are in Appendices B.1, F, and B.3.

5 Results and Analysis

We organize our study around five RQs. RQ1 (§5.1) compares SOTHIS-SQL against published systems. RQ2–RQ5 are ablations isolating each design decision on BIRD dev: difficulty-smoothed data synthesis (§5.2), complementary candidate construction (§5.3), failure-aware repair (§5.4), and confidence-gated selection (§5.5).

5.1 RQ1: How Does SOTHIS-SQL Compare to Prior Systems?

SOTHIS-SQL outperforms all baselines on both benchmarks and generalizes across generalist partners (Table 1). On BIRD dev, Sothis-32B + Gemini-3.1 Pro reaches 75.88 EX, +0.98 above Agentar-Scale-SQL (74.90). Over the best single-pass generalist (Gemini-3.1 Pro), SOTHIS-SQL improves Moderate by +9.48 and Challenging by +12.42. All three pairings improve over both their specialist alone ($\geq$ +2.54 EX) and their corresponding generalist baseline ($\geq$ +7.82 EX), and their ordering tracks each generalist's standalone strength, supporting partner-robust pipeline gains. On SPIDER test, the same configuration transfers without retuning to 91.20 EX, above all baselines.

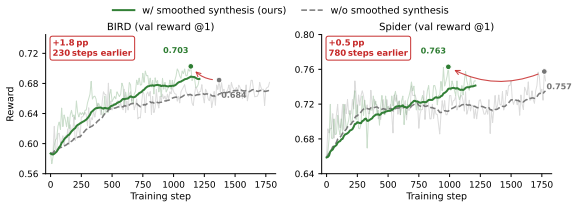

Figure 3: Training-source validation reward@1 during 7B specialist RL, with and without difficulty-smoothed data synthesis.

5.2 RQ2: Does Difficulty-Smoothed Data Synthesis Help Specialist RL Training?

We isolate the effect of difficulty-smoothed data synthesis (§3.3) at 7B scale: only the data synthesis procedure is toggled, with all other training settings held fixed (Appendix C).

Figure 3 shows that difficulty-smoothed synthesis reaches a higher peak validation reward *and* reaches it earlier on both benchmarks: peak 0.703 on BIRD (+1.8 pp, 230 steps earlier) and 0.763 on SPIDER (+0.5 pp, 780 steps earlier). Smoothing concentrates the training corpus on prompts where the baseline is genuinely uncertain, so each RL batch carries a more informative reward signal throughout training. We then apply the same recipe to train Sothis-32B (specialist for all main runs); the 32B curve and additional diagnostics are in Appendix C and C.5.

5.3 RQ3: Does Complementary Candidate Construction Help?

RQ3 asks whether candidate construction makes correct SQL both *reachable* (measured by Oracle@k) and *selectable* (proxied by majority voting), rather than merely increasing the sample count. We compare variants on BIRD dev under a matched candidate budget: each hybrid variant produces $N_s{=}16$ specialist samples (task-specific prompt) and $N_g{=}16$ generalist samples (5-shot ICL prompt), and single-source variants sample one generator $N_s{+}N_g$ times. All variants share the same lifecycle and selector.

Scaling Qwen3-Coder alone (V6) reaches a higher Oracle than scaling Sothis-32B alone (V7) (80.18 vs. 78.42), but its majority vote is lower (71.38 vs. 73.14). The two strengths complement each other: the generalist often reaches a correct SQL without enough within-pool support, while Sothis-32B has lower reachability but stronger consensus among its sampled candidates. Holding the generalist fixed, replacing the second generalist

Table 1: Main results. Execution accuracy on BIRD dev (1,534 questions, with difficulty breakdown) and SPIDER 1.0 test (2,147 questions, gold execution). †: source paper. §: public leaderboard. Unmarked: our runs.

		BIRD				SPIDER
Method	Model	Simple ↑	Moderate ↑	Challenging ↑	Exec. ↑	Exec. ↑
GPT-4o†	GPT-4o	–	–	–	61.9	83.2
Qwen3-Coder	Qwen3-Coder-30B	69.73	54.31	47.59	62.97	85.10
Qwen3-235B	Qwen3-235B	72.54	58.41	53.10	66.43	86.21
Gemini-3.1 Pro	Gemini-3.1 Pro	74.27	60.78	51.72	68.06	88.02
DIN-SQL§	GPT-4	–	–	–	50.72	85.3
DAIL-SQL§	GPT-4	–	–	–	54.76	86.6
MCS-SQL†	GPT-4 8K	–	–	–	63.4	89.6
OmniSQL†	OmniSQL-32B	–	–	–	67.0	89.8
OpenSearch-SQL†	GPT-4o	–	–	–	69.3	87.1
Arctic-Text2SQL-R1†	Arctic-Text2SQL-R1-32B	–	–	–	70.5	88.7
DeepEye-SQL	Qwen3-235B	76.22	64.87	59.31	71.12	87.61
Reasoning-SQL†	Qwen2.5-Coder-14B	–	–	–	72.29	81.43
CHASE-SQL†	Gemini 1.5	–	–	–	73.01	87.6
XiYan-SQL†	XiYanSQL-32B + GPT-4o	–	–	–	73.34	89.65
DeepEye-SQL†	Qwen3-Coder-30B	–	–	–	73.53	89.8
LongData-SQL§	–	–	–	–	74.32	–
Agentar-Scale-SQL§	Agentar-32B + Gemini	–	–	–	74.90	–
Sothis	Sothis-32B	76.22	67.67	58.62	71.97	89.99
Sothis-SQL	Sothis-32B + Qwen3-Coder-30B	79.24	68.97	62.07	74.51	89.85
Sothis-SQL	Sothis-32B + Qwen3-235B	80.00	69.40	64.14	75.30	90.41
Sothis-SQL	Sothis-32B + Gemini-3.1 Pro	**80.54**	**70.26**	**64.14**	**75.88**	**91.20**

Table 2: Candidate construction ablation (RQ3). *5-shots*: k=1 greedy decoding with 5 in-context demonstrations. *matched*: k=N_s+N_g samples drawn from the single source under the same total budget as Hybrid.

#	Variant	k	Oracle@k ↑	Maj Voting ↑	Final Exec. ↑
V1	Qwen3-coder (5-shots)	1	-	-	66.56 ↓7.95
V2	Qwen3-235B (5-shots)	1	-	-	69.30 ↓5.21
V3	XiYanSQL-32B	1	-	-	64.28 ↓10.23
V4	Agentar-32B	1	-	-	70.66 ↓3.85
V5	Sothis-32B	1	-	-	71.97 ↓2.54
V6	Qwen3-Coder only	N_s+N_g	80.18	71.38	72.08 ↓2.43
V7	Sothis-32B only	N_s+N_g	78.42	73.14	73.27 ↓1.24
V8	Qwen3-235B+Qwen3-Coder	N_s+N_g	80.52	72.26	72.41 ↓2.10
V9	XiYanSQL-32B+Qwen3-Coder	N_s+N_g	83.25	72.69	72.88 ↓1.63
V10	Agentar-32B+Qwen3-Coder	N_s+N_g	82.59	74.05	73.92 ↓0.59
V11	**Our (Sothis-32B+Qwen3-Coder)**	N_s+N_g	**82.66**	**74.12**	**74.51**

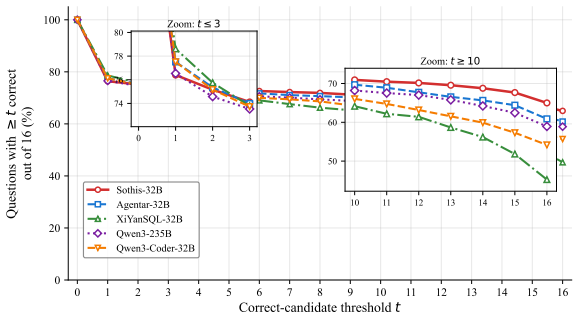

Figure 4: Correct-candidate concentration in single-source 16-sample pools. Each curve plots the fraction of questions with at least t execution-correct candidates. The point t=1 is Oracle@16, while a larger t indicates stronger support for voting and selection.

(V8: Qwen3-235B+Coder, 72.41) with our RL-trained specialist (V11) adds +2.10 EX and lifts Oracle by +2.14pp, isolating the value of having an SQL-specialized, RL-trained source in the pool.

V8–V10 replace Sothis-32B with concurrent specialists. Their Oracle ceilings are comparable (80.52–83.25 vs. 82.66), but Sothis-32B gives the best majority voting and final EX, and Figure 4 shows why. XiYanSQL-32B starts higher at t=1 (78.62% vs. 76.40%) but collapses to 45.24% at t=16, while Sothis-32B stays at 70.93% at $t \geq 10$ and 64.99% at t=16. Thus, Sothis-32B supplies fewer isolated hits and more high-support correct clusters, which follows from the RLVR training in §3.3: rewards depend only on execution correctness, so the trained specialist tends to sample multiple SQL rewrites that share the same correct execution result, anchoring majority voting once merged with a broad generalist.

5.4 RQ4: Does Failure-Aware Repair Beat Unified Correction?

The full type-specific lifecycle outperforms every ablated variant by 0.13–0.71pp (Table 3). We fix the default *Hybrid pool* (Sothis-32B + Qwen3-Coder unless noted otherwise) with exact-result majority voting as the selector, isolating the lifecycle alone.

The most informative row is *coarse LLM repair* (−0.32), which routes non-clean candidates through a generic corrector that sees only the failed SQL. It underperforms even dropping them outright (−0.26): without the typed signal, the corrector treats a timeout from a costly join the same as a silently empty predicate, and produces plausibly-

Table 3: Lifecycle ablation (RQ4) with selector held at exact-result majority voting.

Variant	Exec. ↑	Oracle@k ↑
w/o lifecycle	74.32 ↓0.71	81.49
w/o pre-execution screening	74.90 ↓0.13	82.40
w/o failure typing (drop failures)	74.77 ↓0.26	81.55
w/o failure typing (coarse LLM repair)	74.71 ↓0.32	81.81
w/o post-repair filtering	74.77 ↓0.26	82.14
SOTHIS-SQL (full lifecycle)	**75.03**	**82.27**

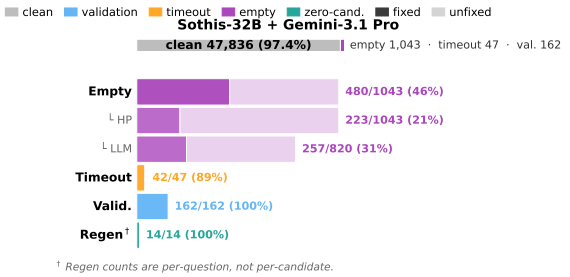

Figure 5: Per-mechanism repair rates on the Sothis-32B + Gemini-3.1 Pro pool (49,088 candidates on BIRD dev). Per-model breakdowns in Appendix Fig. 9.

clean but still-wrong rewrites that outvote the correct ones. The typed dispatch in §3.4 routes each outcome to a structurally different repair path, which is what the coarse-LLM ablation removes.

Fig. 5 attributes the +0.71pp gain to where each lifecycle component is actually doing work. The three components consume non-overlapping slices of the candidate population. Pre-execution screening is responsible for the near-zero runtime-error bar: schema-mismatch candidates are filtered before reaching execution-grounded diagnosis, so almost no runtime errors survive to the typed-repair stage. The cross-check is the second row of Table 3: removing screening exposes 92 runtime-error candidates to the back-end repair path, resulting in a −0.13pp drop in Final Exec. The cost-oriented prompt for timeouts (§3.4) resolves nearly all timeout candidates, since such candidates typically differ from correct ones only in join efficiency. Two-stage empty-result repair is the hardest category: the high-precision (HP) structural operators (Stage 1) and the LLM fallback (Stage 2) together recover close to half of all empty-result candidates, with the remainder dominated by value-binding and projection mismatches.

5.5 RQ5: Does Confidence-Gated Selection Beat Single-Signal Selectors?

Two complementary signals already account for nearly all the gap between single-signal selectors and the full system, while a deterministic gate provides a zero-cost robustness check on the residual ties (Table 4). We fix the Hybrid pool with Gemini-3.1 Pro as a generalist (the configuration that achieves the headline 75.88 EX) and the full failure-aware lifecycle, varying only the selector. Selector variants and RM training are detailed in Appendix E.2 and E.4.

Table 4: Selector ablation (RQ5). Variant definitions in Appendix E.2. *merge* = $S_{\text{maj}} + S_{\text{rm}}$ (§3.5).

Selector	Simple ↑	Moderate ↑	Challenging ↑	Exec. ↑
freq	79.24	70.04	63.45	74.97 ↓0.91
majority voting	79.57	70.04	62.07	75.03 ↓0.85
majority-refine	79.35	69.61	63.45	74.90 ↓0.98
RM	79.24	69.40	58.62	74.32 ↓1.56
merge	80.32	70.04	64.14	75.68 ↓0.20
SOTHIS-SQL (merge + gate)	**80.54**	**70.26**	**64.14**	**75.88**

Most of the gain comes from *merge*, the simple sum $S_{\text{maj}}+S_{\text{rm}}$. The three voting variants (freq, majority, majority-refine) cluster within 0.13pp of each other at 74.9–75.0, indicating that result-level consensus on its own has saturated. RM alone is the weakest single signal at 74.32, because pairwise preference ranks representatives within a cluster of execution-equivalent candidates rather than discriminating across clusters. Merge combines the two without learned weighting and lifts EX to 75.68 (+0.65 over majority voting).

The gate fires only when the top-two gaps in all three scores (merge, consensus-refine, RM) vanish simultaneously *and* multiple result-signature classes are present within size limits (§3.5). The gate then reverts to AST-level structural support without additional LLM calls. The resulting +0.20 EX affects only a small minority of dev questions and is Pareto-improving: same compute, higher accuracy, no parameters tuned across pools.

6 Conclusion

We address three structural limitations of multi-candidate Text-to-SQL voting (L1–L3) by anchoring every stage of the pipeline in execution feedback. SOTHIS-SQL pairs an RLVR-trained SQL specialist with a generalist LLM, refines the resulting pool through an execution-grounded lifecycle with type-specific repair, and selects the final SQL with a hybrid selector that resolves most questions by pooled result-level evidence and escalates only low-margin cases to a deterministic structural check. SOTHIS-SQL reaches 75.88% on BIRD dev and 91.20% on SPIDER test. Two of three generalist pairings surpass Agentar-Scale-SQL, and all three improve over their generalist baselines.

7 Limitations

SOTHIS-SQL requires both a domain-specialized RL-trained model and a strong generalist LLM. Deployments that lack either will not benefit equally. The empirical residual headroom of the lifecycle focuses on *empty-result* candidates whose errors are dominated by value-binding and projection mismatches (Fig. 5), where our type-specific operators recover roughly half of these candidates, but the remainder remains a hard frontier. Finally, comparisons to published systems rely on their reported numbers and may differ in the retrieval pipeline, schema linking, or backbone. We therefore use the backbone-matched controls in RQ3 as the primary evidence for the design choices.

Ethics Statement

This work focuses on benchmark evaluation of Text-to-SQL systems and does not involve human subjects, private data, or dual-use concerns. All experiments use publicly available benchmarks and models. We use publicly available benchmarks and models according to their intended research use and applicable licenses or terms of use. Any released model checkpoints and derived artifacts will be distributed under licenses compatible with the applicable licenses and terms of the underlying resources.

References

Ben Bogin, Jonathan Berant, and Matt Gardner. 2019. Representing schema structure with graph neural networks for text-to-sql parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4560–4565.

Jipeng Cen, Jiaxin Liu, Zhixu Li, and Jingjing Wang. 2025. Sqlfixagent: Towards semantic-accurate text-to-sql parsing via consistency-enhanced multi-agent collaboration. In *Thirty-Ninth AAAI Conference on Artificial Intelligence, Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence, Fifteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2025, Philadelphia, PA, USA, February 25 - March 4, 2025*, pages 49–57. AAAI Press.

Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John H. Drake, and Qiaofu Zhang. 2021. Natural SQL: making SQL easier to infer from natural language specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, Findings of ACL, pages 2030–2042. Association for Computational Linguistics.

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

Mingqian He, Yongliang Shen, Wenqi Zhang, Qiuying Peng, Jun Wang, and Weiming Lu. 2025. Star-sql: Self-taught reasoner for text-to-sql. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 24365–24375. Association for Computational Linguistics.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025a. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 337–353.

Jimin Lee, Ingeol Baek, Byeongjeong Kim, Hyunkyung Bae, and Hwanhee Lee. 2025b. SAFE-SQL: self-augmented in-context learning with fine-grained example selection for text-to-sql. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025, Suzhou, China, November 4-9, 2025*, pages 19023–19035. Association for Computational Linguistics.

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. 2025. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

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

Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. Alpha-sql: Zero-shot text-to-sql using monte carlo tree search. In *Forty-second International Conference on Machine Learning*.

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. 2025b. Omnisql: Synthesizing high-quality text-to-sql data at scale. *Proc. VLDB Endow.*, 18(11):4695–4709.

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

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

Zhishuai Li, Xiang Wang, Jingjing Zhao, Sun Yang, Guoqing Du, Xiaoru Hu, Bin Zhang, Yuxiao Ye, Ziyue Li, Hangyu Mao, and Rui Zhao. 2025c. PET-SQL: A prompt-enhanced two-round refinement of text-to-sql with cross-consistency. In *Database Systems for Advanced Applications - 30th International Conference, DASFAA 2025, Singapore, Singapore, May 26-29, 2025, Proceedings, Part II*, Lecture Notes in Computer Science, pages 193–208. Springer.

Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. Bridging textual and tabular data for cross-domain text-to-sql semantic parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4870–4888.

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

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

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.

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

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in neural information processing systems*, 36:36339–36348.

Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wan, Hailong Li, Azalia Mirhoseini, Amin Saberi, and Sercan Ö. Arik. 2025b. Reasoning-sql: Reinforcement learning with SQL tailored partial rewards for reasoning-enhanced text-to-sql. *CoRR*, abs/2503.23157.

Jiexing Qi, Jingyao Tang, Ziwei He, Xiangpeng Wan, Yu Cheng, Chenghu Zhou, Xinbing Wang, Quanshi Zhang, and Zhouhan Lin. 2022. Rasat: Integrating relational structures into pretrained seq2seq model for text-to-sql. In *Proceedings of the 2022 conference on empirical methods in natural language processing*, pages 3215–3229.

Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-sql generation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 5456–5471.

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

Zhili Shen, Pavlos Vougiouklis, Chenxin Diao, Kaustubh Vyas, Yuanyi Ji, and Jeff Z Pan. 2024. Improving retrieval-augmented text-to-sql with ast-based ranking and schema pruning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 7865–7879.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*.

Qwen Team. 2025. Qwen3 technical report. *CoRR*, abs/2505.09388.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 7567–7578.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang,

Di Yin, Xing Sun, and Zhoujun Li. 2025a. MAC-SQL: A multi-agent collaborative framework for text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu Dhabi, UAE, January 19-24, 2025*, pages 540–557. Association for Computational Linguistics.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, and Zhifang Sui. 2024. Large language models are not fair evaluators. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 9440–9450. Association for Computational Linguistics.

Pengfei Wang, Baolin Sun, Xuemei Dong, Yaxun Dai, Hongwei Yuan, Mengdie Chu, Yingqi Gao, Xiang Qi, Peng Zhang, and Ying Yan. 2025b. Agentar-scale-sql: Advancing text-to-sql through orchestrated test-time scaling. *arXiv preprint arXiv:2509.24403*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Yihan Wang, Peiyu Liu, and Xin Yang. 2025c. 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.

Yuanzhen Xie, Xinzhou Jin, Tao Xie, Mingxiong Lin, Liang Chen, Chenyun Yu, Cheng Lei, Chengxiang Zhuo, Bo Hu, and Zang Li. 2024. Decomposition for enhancing attention: Improving llm-based text-to-sql through workflow paradigm. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, Findings of ACL, pages 10796–10816. Association for Computational Linguistics.

Wenbo Xu, Haifeng Zhu, Liang Yan, Chuanyi Liu, Peiyi Han, Shaoming Duan, and Jeff Z Pan. 2025. Ts-sql: Test-driven self-refinement for text-to-sql. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 2864–2889.

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.

Zhewei Yao, Guoheng Sun, Lukasz Borchmann, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. 2025. Arctic-text2sql-r1: Simple rewards, strong reasoning in text-to-sql. *CoRR*, abs/2505.20315.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Weinan Dai, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. 2025. DAPO: an open-source LLM reinforcement learning system at scale. *CoRR*, abs/2503.14476.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 3911–3921. Association for Computational Linguistics.

Jipeng Zhang, Haolin Yang, Kehao Miao, Ruiyuan Zhang, Renjie Pi, Jiahui Gao, and Xiaofang Zhou. 2025a. 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.

Yuhao Zhang, Shaoming Duan, Jinhang Su, Chuanyi Liu, and Peiyi Han. 2025b. Spft-sql: Enhancing large language model for text-to-sql parsing by self-play fine-tuning. *arXiv preprint arXiv:2509.03937*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

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

A Appendix Overview

This appendix is organized to mirror the evidence chain in the main paper. Appendix B first fixes the experimental protocol, baseline settings, and input schema context used across all runs. Appendix C then expands L1, covering specialist training, difficulty smoothing, the DAPO objective, the execution sandbox, and training diagnostics. Appendix D expands L2, giving the execution-grounded candidate lifecycle, its diagnostic taxonomy, repair operators, and additional failure-breakdown evidence. Appendix E expands L3, including tuple-level consensus, score-family ablations, position-debiased pairwise preference, the pairwise reward model, and structural-signature features. Algorithms 1 and 2 are placed inside their corresponding L2 and L3 sections. Appendix F lists representative prompt templates, and Appendix G collects extended discussion. The appendix figures are marked at the point where they support the main experimental claims: Fig. 6 and Fig. 7 support L1 training, Fig. 8 adds the critic-reward diagnostic for RQ2, and Fig. 9 expands the L2 failure analysis.

B Experimental Protocol and Input Context

This section collects the implementation, evaluation, baseline, and input-context details that are summarized in compact form in §4 of the main paper.

B.1 Serving, Timeouts, and Decoding Profiles

Open-weight models (Sothis-32B, Qwen3-235B-Instruct-2507, Qwen3-Coder-30B-A3B-Instruct) are served through OpenAI-compatible endpoints with vLLM 0.11 (Kwon et al., 2023). Gemini-3.1-Pro-Preview is accessed through its hosted API. Query execution uses two timeout budgets: 120 seconds during the candidate lifecycle, and 30 seconds during selection-time re-execution. The tighter selection-time budget rejects candidates whose earlier acceptance depended on a transient long run, while the lifecycle budget is loose enough to avoid penalizing expensive but correct queries during repair. Each question receives 16 candidates from the specialist and 16 from the generalist, for a total of 32 candidates per question. Within each source, samples are drawn with a fixed temperature profile and multiple prompt routes; route and decoding metadata are attached to every candidate as provenance signals consumed by the selector.

B.2 Evaluation Protocol Details

All results are reported under each benchmark's official evaluation protocol.[4] BIRD adopts the stricter criterion: beyond matching the set of returned rows, the element order *within* each row (i.e., column order) must also match the gold result. SPIDER ignores column order and only requires value equivalence on each row. The BIRD difficulty breakdown (simple / moderate / challenging) follows the official BIRD release.

B.3 Backbone-Matched Baseline Configurations

The "backbone-matched" baselines in Table 1 all share the same scaffold as SOTHIS-SQL (schema linking, prompt routes, candidate budget) and differ only in the candidate sources or the selector:

- **Qwen3 self-consistency.** Samples 16 SQL candidates from Qwen3-235B-Instruct-2507 (Team, 2025) and resolves them by majority voting only, with no lifecycle and no hybrid selector.
- **Sothis-only pipeline.** Uses only the specialist generator Sothis-32B for all 32 candidates, then runs the full execution-grounded lifecycle and hybrid selector.
- **Qwen3-only pipeline.** Uses only the generalist Qwen3-235B-Instruct-2507 (Team, 2025) for all 32 candidates, then runs the full lifecycle and hybrid selector.
- **SOTHIS-SQL (full).** Pairs Sothis-32B with the generalist partner (16+16 candidates) and runs the full lifecycle and hybrid selector.

The prompted single-model rows at the top of Table 1 (Qwen3-Coder, Qwen3-235B, Gemini-3.1 Pro, Sothis-32B) use the same linked-schema prompt under 5-shot prompting but remove the lifecycle and selector entirely, reporting the single-pass EX of each backbone.

B.4 Schema Context Construction

We formalize the question-specific schema context introduced in §3.3. Given a question q and a target database d, the runtime always loads an offline schema prompt

$$s = \Phi(q, d), \tag{6}$$

where s is a prompt-ready schema description (table DDLs, column names, foreign-key edges, inline

[4] We use the public scripts distributed by the benchmark organizers without modifying correctness criteria.

per-column comments, and short per-column examples). Optionally, the runtime may also inject a question-level value-hint block

$$h = \Psi(q, d), \tag{7}$$

when such hints are provided by an external cache. The main runs reported in this paper always use the offline schema prompt s; value hints h are supported by the implementation but are optional rather than assumed. When available, they are forwarded together with the question and evidence as structured grounding context (q, e, s, h).[5] This design keeps the online pipeline lightweight and makes candidate generation reproducible across runs.

B.4.1 Schema Representation Format

We adopt an *enriched DDL* representation that augments the standard `CREATE TABLE` skeleton with two additional information channels: (i) **column comments** that disambiguate cryptic or abbreviated column names, and (ii) **question-conditioned example values** that ground the model's understanding of the data distribution. Unless otherwise specified, this subsection describes the default enriched schema context (DES) used by our main pipeline. Listing 1 shows a representative fragment of our enriched schema format; Listing 2 shows the same tables under a plain DDL baseline (Wang et al., 2025b) that retains only structural metadata.

Listing 1: Enriched schema format used by SOTHIS-SQL (excerpt from the `financial` database).

```
CREATE TABLE account (
    account_id integer, -- example: [1,
    2]
    district_id integer, -- location of
    branch, example: [18, 1]
    frequency text, -- example:
    ['POPLATEK PO OBRATU', 'POPLATEK
    MESICNE']
    `date` date, -- example:
    ['1995-03-24', '1993-02-26']
    PRIMARY KEY (account_id),
    CONSTRAINT fk_account_district_id
    FOREIGN KEY (district_id)
    REFERENCES district (district_id)
);

CREATE TABLE district (
    district_id integer, -- location of
    branch, example: [1, 2]
    A2 text, -- district_name, example:
    ['Hl.m. Praha', 'Benesov']
    A3 text, -- region, example: ['east
    Bohemia', 'Prague']
    A4 text, -- number of inhabitants,
    example: ['1204953', '88884']
    A5 text, -- no. of municipalities
    with inhabitants < 499, example:
    ['0', '80']
    PRIMARY KEY (district_id)
);
```

Listing 2: Plain DDL schema baseline (same tables, no comments or examples).

```
CREATE TABLE account (
    account_id integer,
    district_id integer,
    frequency text,
    `date` date,
    PRIMARY KEY (account_id),
    CONSTRAINT fk_account_district_id
    FOREIGN KEY (district_id)
    REFERENCES district (district_id)
);

CREATE TABLE district (
    district_id integer,
    A2 text,
    A3 text,
    A4 text,
    A5 text,
    PRIMARY KEY (district_id)
);
```

Design rationale. The enriched format addresses three failure modes commonly observed when LLMs operate on plain DDL alone:

1. **Ambiguous column semantics.** Columns such as `A2`–`A5` in the `district` table carry no self-explanatory meaning. The comment annotation (e.g., `- district_name`) resolves this ambiguity without requiring the model to infer semantics from data alone. A comment is emitted only when the human-authored column description differs from the column name itself; otherwise the comment is suppressed to avoid redundancy.

2. **Value-space grounding.** Example values expose the actual vocabulary stored in the database. This helps the model produce correct literal predicates in `WHERE` clauses without hallucinating non-existent values. Up to six values per column are shown, drawn from two complementary sources: (a) a distinct non-null sample of values from the database, and (b) question-relevant values retrieved via BM25 from a pre-built database-content index.

3. **Structural completeness.** Both formats preserve primary-key and foreign-key constraints verbatim, ensuring that join-path reasoning is

[5] The offline schema cache is keyed by question identifier. Optional value hints are keyed by question and database identifier so that all generators see the same side information when hints are enabled.

available regardless of the enrichment level. Column identifiers containing reserved words or special characters are wrapped in backticks.

Construction pipeline. The schema $s{=}\Phi(q, d)$ is assembled offline by the following steps:

1. Load the database metadata (table names, column names, column comments, column types, primary keys, foreign keys) from the schema catalogue.
2. For each column, sample up to k distinct non-null values from the SQLite database (truncated to 40 characters for long strings).
3. Tokenize the question q (augmented with any external knowledge e) into n-grams ($n{\leq}8$) and retrieve matching database cell values via a Lucene-based BM25 index built over all column contents.
4. Merge sampled and retrieved values per column (deduplicated, capped at six), and emit the enriched DDL with inline comments and example annotations.

When optional value hints are enabled, the hint set $h{=}\Psi(q, d)$ is the subset of retrieved values whose substring-match score against the question exceeds 0.85, surfaced separately in the prompt as explicit value-link hints.

B.4.2 Schema Variant Ablations

In addition to DES, we construct eight controlled schema variants to isolate the effects of schema scope and metadata richness. All eight variants are serialized in the same enriched-DDL template so that the ablation changes the schema content rather than the prompt format.

Scope. We consider two schema scopes:

1. **Full.** The full variant keeps all tables and columns in the target database together with their primary-key and foreign-key constraints.
2. **Pruned.** The pruned variant first performs question-conditioned schema linking and then retains only the linked tables and columns. This is an instance-specific schema selection step rather than generic prompt truncation.

Metadata levels. Under each scope, we instantiate four metadata configurations:

1. **Plain.** This is the minimal variant in the ablation family. It keeps only the basic schema serialization under our template and does not add question-conditioned retrieved values, value descriptions, or value-range annotations.
2. **Retrieved examples.** This variant augments columns with question-conditioned example values from the embedding-based value-retrieval pipeline. Concretely, an LLM first extracts a compact set of salient keywords from the question and evidence. These keywords are embedded and used to retrieve semantically related cell values from a per-database value index on a column-wise basis. The retrieved values are then inserted into the example field of the corresponding columns under a fixed budget.
3. **Retrieved examples + value meaning.** This variant additionally exposes human-authored description fields when they are available. In our implementation, this bundle may include expanded column names, short column descriptions, and value descriptions, so the model sees both retrieved example values and natural-language explanations of what a column or its values mean.
4. **Retrieved examples + value meaning + value range.** This is the richest variant. In addition to retrieved examples and description fields, it adds coarse value-space constraints. For numeric columns, we provide minimum/maximum-style range information. For low-cardinality categorical columns, we expose small valid-value sets.

Relation to the default schema. These eight variants should not be conflated with the default enriched schema context. DES uses question-conditioned lexical retrieval over database contents via n-gram expansion and BM25 matching, then merges the retrieved values with sampled column values. By contrast, the ablation variants with *retrieved examples* use an LLM-guided keyword-extraction stage followed by embedding-based value retrieval, which changes both the retrieval mechanism and the resulting example set. Similarly, the *plain* variant in this ablation family is not the same as the pure DDL baseline in Listing 2; it is simply the least enriched member within our unified ablation template.

B.4.3 Schema Variant Results and Prompt Size

Table 5 reports all schema-variant results on the BIRD dev set, computed over all 1,534 questions. For Qwen3-Coder, the strongest controlled

variant is F+E, which leads that family on all three metrics: 71.45 Final EX, 71.25 majority-voting EX, and 78.16 Oracle@k. For Qwen3-235B, DES remains the strongest overall setting at 71.77 Final EX, 71.45 majority-voting EX, and 77.38 Oracle@k, while the strongest controlled variants are more fragmented: F+E leads Final EX (71.58) and Oracle@k (77.31), whereas F+E+M reaches the best majority-voting score within that family (71.19). Across both backbones, adding *retrieved examples* consistently helps the corresponding *plain* schemas, whereas further adding *value meaning* and *value range* does not produce stable gains and often slightly degrades the final metric. From the joint perspective of stability across backbones and absolute effectiveness, DES remains the most reliable default choice: it is best on Qwen3-235B and stays close to the best controlled variant on Qwen3-Coder without requiring the richer retrieval-based variant stack.

Table 6 complements the accuracy comparison with prompt-side schema length. The full variants are expensive, ranging from 2.9k to 4.5k tokens on average, while the pruned variants fall between roughly 326 and 481 tokens. In other words, pruning reduces the average schema length by nearly an order of magnitude. However, this compression does not translate into a consistent EX improvement in Table 5. The default enriched schema context occupies a middle point at roughly 1.95k tokens on average while remaining competitive with, and in one case stronger than, the richer full variants with embedding-retrieved examples. Taken together, the results indicate that better question-conditioned value grounding matters more than simply adding more schema-side metadata or aggressively minimizing prompt length, and they further support retaining DES as the primary setting in the main system.

C L1: Specialist Training and Complementary Candidate Construction

This section expands L1 in §3.3 with the training data composition, the combined training objective (DAPO + threshold-gated entropy), the training-time execution sandbox, and the two-stage DAPO configuration used for the SOTHIS-SQL specialist generator. We train the specialist under reinforcement learning with verifiable rewards (RLVR): the environment supplies reliable supervision by executing sampled SQL programs, but the reward is observed only after a complete response is produced. This terminal-only signal makes policy collapse a practical failure mode. As the model overfits easy prompts, its response distribution can become too sharp, triggering entropy collapse and reducing the sampling diversity needed to cross discontinuous execution-reward boundaries on harder tasks. In group-relative training, this appears as all-correct or all-incorrect rollout groups, where the in-group advantage is zero and the prompt contributes no useful gradient. Our recipe therefore balances exploration and exploitation from two sides: we reshape the training data to reduce zero-variance prompts, and we use a dynamic-sampling DAPO schedule with entropy control to keep policy updates well conditioned.

C.1 Training Data

The specialist is trained on a union of three Text-to-SQL sources converted into a single prompt format: the BIRD training split, the SPIDER training split, and a sample from SynSQL (Li et al., 2025b). Each example is serialized with the default enriched schema context (DES) described in §B.4.1, so the training prompt exposes table structure, column descriptions, and representative values in the same format used at inference time.

Data construction and cleaning. We apply a two-stage cleaning pipeline before RL. First, the execution environment drops examples whose gold SQL fails to execute, times out, or returns an empty result on the paired database. This avoids introducing zero-reward targets that would inflate the number of zero-variance prompts during RL. Second, we use a multi-model voting filter with DeepSeek-v3, GPT-5, and Qwen3-235B to check semantic consistency from multiple angles, including schema selection, join structure, and predicate constraints. Examples that fail this cross-model consistency check are removed from the training pool, reducing noisy supervision before policy optimization. The resulting examples, together with the SynSQL sample, are consolidated into a single DAPO training set. Before DAPO training, we reserve 500 examples from the training sources as a validation split for monitoring training dynamics; these examples are excluded from policy updates. Table 7 summarizes the final composition.

Difficulty smoothing. The difficulty distribution of the raw training pool directly affects the qual-

Table 5: Schema-variant results on BIRD dev (%). DES: default enriched schema. F: full, P: pruned, E: retrieved examples, M: value meaning, R: value range.

	Qwen3-Coder			Qwen3-235B		
Variant	Final EX ↑	Maj ↑	Oracle@k ↑	Final EX ↑	Maj ↑	Oracle@k ↑
DES	70.99	70.53	77.71	**71.77**	**71.45**	**77.38**
F	69.82	69.49	76.79	70.93	70.73	76.66
F+E	**71.45**	**71.25**	**78.16**	71.58	71.06	77.31
F+E+M	71.06	71.12	77.05	71.25	71.19	77.25
F+E+M+R	70.99	70.73	77.64	70.66	70.80	76.92
P	70.14	69.17	75.68	71.12	71.06	75.75
P+E	71.06	70.80	76.92	71.25	71.12	76.01
P+E+M	70.34	70.14	76.47	70.80	70.53	75.81
P+E+M+R	70.86	70.60	76.47	71.06	70.60	76.14

Table 6: Prompt-side schema size of the same variants, measured in Qwen3-Coder tokens.

Variant	Avg.	Min.	Max.	p50	p90	p95	p99
DES	1951	383	4654	2051	2970	4641	4654
F	2858	519	6172	3174	5188	6172	6172
F+E	3148	649	6449	3216	5855	6417	6433
F+E+M	3821	672	7765	3419	7128	7733	7749
F+E+M+R	4521	837	10348	3982	7859	10316	10332
P	326	30	2444	287	521	684	1401
P+E	361	30	2610	318	603	744	1487
P+E+M	411	49	2730	336	689	842	1569
P+E+M+R	481	52	3561	432	789	941	1954

Source	**Rows**	**Share**
SynSQL sample	20,484	75.4%
BIRD train	4,726	17.4%
SPIDER train	1,972	7.3%
Total (train)	27,182	100%
Validation (train sources)	500	–

Table 7: Specialist training data composition after execution filtering and multi-model semantic-consistency filtering. SynSQL is subsampled from the public release. The validation split is reserved from the training sources before DAPO optimization.

ity of policy gradients. Very easy prompts mostly reinforce behavior that the model can already exploit, providing limited training signal. Moderately hard prompts are the most useful for RL because they still admit occasional successes while forcing the policy to explore beyond its current high-probability outputs. We estimate prompt difficulty by sampling a baseline model for 10 independent rollouts and grouping examples by the number of correct rollouts: $0/10$ denotes prompts the baseline never solves, while $10/10$ denotes prompts solved in every rollout. The raw pool is strongly imbalanced, with heavy mass at the easy end and a long tail of hard prompts (Fig. 6). We reshape the data distribution by smoothing overrepresented pass-rate buckets and removing the trivially solved $10/10$ bucket, so RL batches concentrate on prompts with informative rollout variation. The $10/10$ bin is retained in the figure to show that these prompts are intentionally emptied by the smoothing procedure rather than omitted from the axis.

C.2 DAPO Objective, Dynamic Sampling, and Threshold-Gated Entropy

The total training objective combines the DAPO policy loss with a threshold-gated entropy regularizer:

$$\mathcal{L}_{\text{total}}(\theta) = \mathcal{L}_{\text{DAPO}}(\theta) + w_{\text{ent}}\, \mathcal{L}_{\text{ent}}(\theta). \quad (8)$$

Reward. Each sampled SQL receives a scalar execution reward from the training-time sandbox of §C.3, taking one of three values: $R{=}1$ when the SQL is executable and its row set matches the gold, $R{=}0.1$ when it is executable but the rows do not match, and $R{=}0$ when it is unparsable or triggers a runtime or timeout error. The small partial-credit value of 0.1 for executable-but-wrong rollouts is important in the zero-variance-prompt regime: when all n rollouts fail to match the gold, the partial-credit term nonetheless separates "plausibly executable" rollouts from broken ones, keeping the group advantage non-degenerate and producing a usable gradient that a pure $\{0, 1\}$ reward would have lost. An additional soft penalty on overlong generations is applied using a length buffer of 1024 tokens and penalty factor 1.0.

GRPO advantage and dynamic sampling. For prompt x with n sampled responses $\{y_i\}_{i=1}^{n}$ from π_θ, GRPO normalizes the scalar reward within the group:

$$A_i = \frac{R(x, y_i) - \mu}{\sigma + \epsilon}, \quad (9)$$

where μ and σ are the in-group mean and standard deviation of $R(x, y_k)$, and

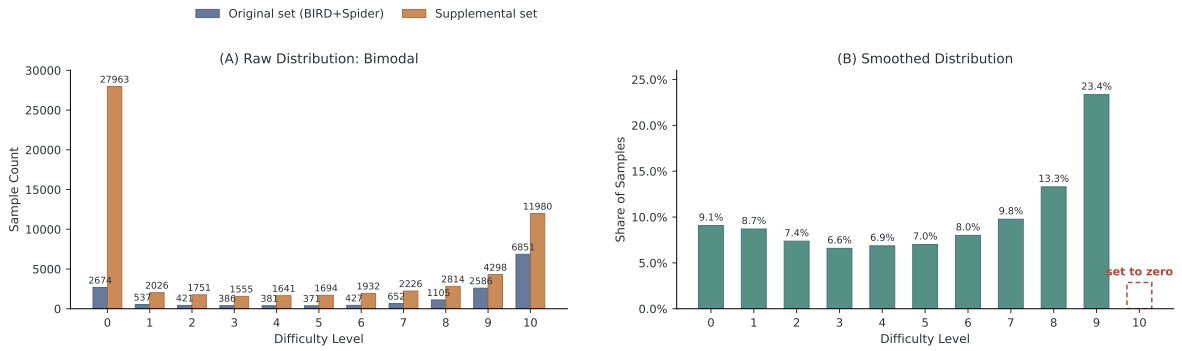

Figure 6: Difficulty distribution before and after smoothing. The x-axis is the number of correct baseline rollouts out of 10. Smoothing redistributes training mass over informative buckets and sets the trivially solved $10/10$ bucket to zero.

A_i is broadcast to every token in y_i. Let $\rho_{i,t}(\theta)=\pi_\theta(y_{i,t}|x, y_{i,<t})/\pi_{\theta_{\text{old}}}(y_{i,t}|x, y_{i,<t})$ denote the token-level importance ratio. Policy optimization uses the per-token asymmetric-clipped surrogate (Yu et al., 2025):

$$\mathcal{L}_{\text{DAPO}}(\theta) = \frac{\nVdash[\text{Var}_i R(x, y_i)>0]}{n} \sum_{i=1}^{n} \ell_i(\theta), \tag{10}$$

$$\ell_i(\theta) = \min\Big(\rho_{i,t}\, A_i, \quad \text{clip}(\rho_{i,t},\, 1-\epsilon_l,\, 1+\epsilon_h)\, A_i\Big). \tag{11}$$

The leading indicator implements *dynamic sampling*. Prompts whose rollouts produce all-correct or all-incorrect groups are discarded from the gradient update, and up to e_{max} extra rollout batches are drawn until a training batch of non-degenerate prompts is accumulated. This rejection-sampling loop is especially important under sparse execution feedback: it adaptively spends additional inference on prompts whose initial rollouts do not provide relative advantage, so the optimizer sees batches with a higher signal-to-noise ratio. We use ϵ_l=0.2 and ϵ_h=0.28, and disable KL entirely.

Threshold-gated entropy regularizer. Beyond the standard DAPO components above, we add a simple but effective gate against entropy collapse. Let $H(\pi_\theta) = -\sum_a \pi_\theta(a \mid s_t) \log \pi_\theta(a \mid s_t)$ be the average policy entropy over the current rollout, and let $\mathcal{L}_{\text{ent}}(\theta)$ denote its empirical estimate:

$$\mathcal{L}_{\text{ent}}(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{a \in \mathcal{V}} \pi_\theta(a \,|\, s_t) \log \pi_\theta(a \,|\, s_t). \tag{12}$$

The gating coefficient is a step function of $H(\pi_\theta)$:

$$w_{\text{ent}} = \beta \cdot \mathbf{1}\big[H(\pi_\theta) < H_{\text{target}}\big], \tag{13}$$

This means the entropy bonus pays zero regularization tax whenever the policy already maintains sufficient diversity, and only injects a fixed gradient β once the rollouts start collapsing. Unlike a constant entropy coefficient, this "activate-on-collapse" schedule preserves the execution-reward signal during normal optimization and intervenes only at the boundary of entropy failure. The target entropy is stage-specific (Table 8), and we set β=5×10^{-3}. In practice β is warmed up with step Δ=10^{-4} to avoid abrupt gradient spikes when the gate first opens.

C.3 Training-Time Execution Sandbox

Because the reward requires running every rollout against a real database at every optimization step, we wrap SQL execution in a reproducible training-time sandbox. Each rollout and its gold SQL are executed on the target SQLite database through a read-only connection, their result sets are compared as unordered sets, and the three-level reward rule above is applied. A 60-second wall-clock budget is enforced per query: timeouts and runtime errors are both mapped to reward 0, while executable-but-wrong queries receive the partial credit 0.1.

C.4 Specialist DAPO Configuration

Table 8 summarizes the specialist DAPO configuration. Our training recipe follows a two-stage schedule sharing the same objective: an exploration stage with a wide rollout (n=32, temperature 1.2, H_{target}=0.30) that helps collect sparse successful trajectories on hard prompts, and an exploitation stage that resumes from the exploration checkpoint and anneals the sampling width (n=16, temperature 0.9, H_{target}=0.15). The first stage uses broad sampling and a higher entropy target to prevent

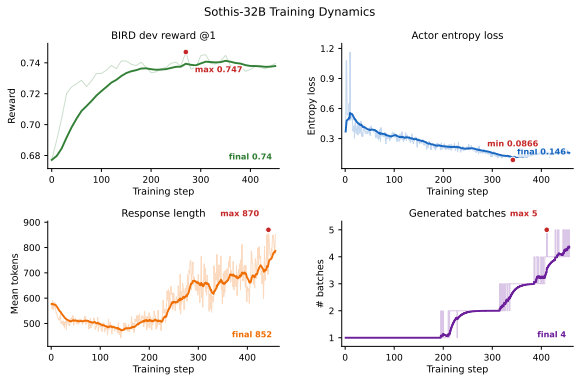

Figure 7: Sothis-32B training dynamics under the two-stage DAPO recipe. Validation reward improves and stabilizes; entropy loss decays without immediate collapse; response length first contracts and later recovers as SQL completions become more detailed; generated batches increase when dynamic sampling needs extra rollouts to construct non-degenerate updates.

	Exploration	**Exploitation**
Base model	OmniSQL-32B	
Adv. estimator	GRPO	
Rollout n	32	16
Rollout temperature	1.2	0.9
Clip ratio low/high	0.2 / 0.28	
KL regularization	disabled	
Entropy target H_{target}	0.30	0.15
Entropy coef. β	5×10^{-3}	
Entropy warm-up Δ	10^{-4}	
Max prompt / response	6000 / 5120	
Overlong buffer	1024 tokens, factor 1.0	
Dyn. sampling retries	up to 10 batches	
Train / gen / mini bsz	128 / 384 / 128	
Learning rate	1×10^{-5}	
LR warmup steps	10	
Weight decay	0.1	
Gradient clip	1.0	
Epochs	20	10
Hardware	4×8 GPUs	

Table 8: Two-stage specialist training configuration. Exploitation resumes from the exploration checkpoint and anneals the sampling temperature and rollout width while preserving dynamic sampling.

early homogenization as previously challenging prompts become easy. The second stage applies a simulated-annealing style constraint: it lowers the sampling temperature, reduces the rollout width, and removes the entropy target so that optimization shifts from broad state-space exploration toward reinforcing verified high-confidence solutions with less rollout variance.

Figure 7 summarizes these dynamics. The reward curve stabilizes near its final value rather than peaking and immediately collapsing. The entropy-loss curve decreases as the policy becomes more confident, while the generated-batch curve rises later in training, indicating that dynamic sampling increasingly has to search for rollout groups with non-zero relative advantage as the model solves more prompts consistently.

C.5 Critic Reward under Difficulty-Smoothed Data

Figure 8 reports the critic reward during specialist RL under the same two configurations compared in Fig. 3. With difficulty-smoothed data synthesis the critic reward reaches a higher steady-state mean (0.641 vs. 0.604, +3.8 pp) and maintains a visibly larger variance band throughout training, consistent with a non-collapsing policy that continues to explore novel trajectories after the baseline has plateaued. We stop training at step 1,210 for the smoothed run because the critic curve has entered a stable regime.

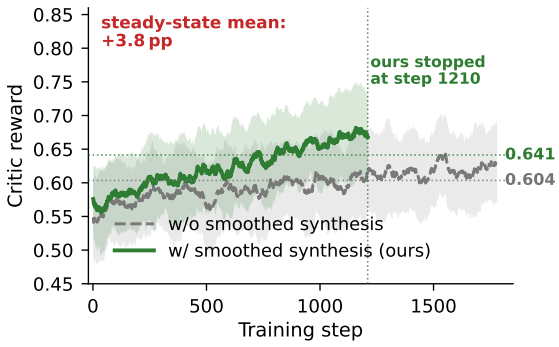

Figure 8: Critic reward during specialist RL.

D L2: Execution-Grounded Candidate Lifecycle

Algorithm 1 gives the full lifecycle pseudocode before the block-level details.

This section expands §3.4 of the main paper and follows the three blocks of the Execution-Grounded Lifecycle panel in Figure 2: **(A) Pre-execution Screening**, **(B) Execution-Grounded Diagnosis**, and **(C) Failure-Aware Pool Update**. Block A applies deterministic sanitization and lightweight syntax/schema validation, removing malformed candidates from the pool before execution. Block B executes every surviving candidate and tags each with one of four diagnostic labels {`clean`,`runtime`,`timeout`,`empty`}. Block C dispatches each non-`clean` candidate to a dedicated type-specific repair path, re-executes

Algorithm 1 Execution-Grounded Candidate Lifecycle

Require: input $x = (q, e, d)$, linked context (s, h), generators $G_{\text{sp}}, G_{\text{ge}}$
Ensure: pool $\mathcal{C}^{(*)}$

$\mathcal{C}^{(0)} \leftarrow G_{\text{sp}}(q, e, s, h) \cup G_{\text{ge}}(q, e, s, h)$ ▷ candidate construction
$(\mathcal{C}^{(1)}, \mathcal{R}_{\text{phase2}}) \leftarrow \mathsf{Validate}(\mathcal{C}^{(0)})$ ▷ pre-execution screening
for all $c \in \mathcal{C}^{(1)}$ **do**
  $\delta(c) \leftarrow \mathsf{Exec}(c)$ ▷ $\delta \in \{\texttt{clean}, \texttt{runtime}, \texttt{timeout}, \texttt{empty}\}$
end for
for all c with $\delta(c) \neq \texttt{clean}$ **do**
  if $\delta(c) = \texttt{empty}$ **then**
    $c' \leftarrow R_{\text{op}}(c;\ \texttt{operators})$ ▷ high-precision empty-result operators first
    if c' still returns empty **then**
      $c' \leftarrow R_{\text{empty_llm}}(c;\ q, e, s, h)$ ▷ diagnosis-conditioned Stage-2 fallback
    end if
  else
    $c' \leftarrow R_{\delta(c)}(c;\ q, e, s, h)$ ▷ runtime/timeout/validation repair
  end if
  if $c' \neq \varnothing$ **then**
    $\delta(c') \leftarrow \mathsf{Exec}(c')$ ▷ immediate re-execution
    replace c with c' if $\delta(c') = \texttt{clean}$, else mark c unresolved
  end if
end for
$\mathcal{C}^{(*)} \leftarrow \mathsf{NormalizeProjection}(\mathsf{Consolidate}(\mathcal{C}^{(1)}))$
if $\mathcal{C}^{(*)} = \varnothing$ **then**
  build condensed negative feedback $(\mathcal{B}, \rho)$ from unresolved and rule-filtered SQLs
  $\mathcal{C}^{(*)} \leftarrow G_{\text{rec}}(q, e, s; \mathcal{B}, \rho)$ ▷ zero-candidate recovery
end if
return $\mathcal{C}^{(*)}$

accepted rewrites immediately, normalizes projections deterministically, and regenerates only when the pool is fully exhausted.

D.1 Block A: Pre-execution Screening

Before any execution or LLM correction, candidates are sanitized by a short set of deterministic preprocessors targeted at common LLM-generation artifacts: stripping `-/#` comment prefixes that precede the actual SQL, inserting a missing space after the `SELECT` keyword when the LLM emits `SELECT*`, and rejecting outputs whose first token is not a valid SQL leading keyword (`SELECT`, `WITH`, `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `DROP`). The sanitized SQL is then passed through dual validation. The syntax validator is a lightweight rule-based checker rather than a full SQL parser: it verifies a legal leading keyword, required `FROM` presence for `SELECT`, balanced parentheses outside string literals, balanced quotes, and a few common comma-related artifacts. The schema validator then combines simple table-name extraction with an `EXPLAIN`-based check that catches missing tables, missing columns, ambiguous columns, and other SQLite-side schema mismatches. Candidates failing either check are removed from the pool in Phase 2, with the specific failure subtype attached for diagnostics and later summary statistics.

D.2 Block B: Execution-Grounded Diagnosis

Block B executes every candidate that survives pre-execution screening and attaches a diagnostic label used by the downstream repair dispatcher. A candidate is marked `clean` when execution succeeds and returns a non-empty result, `empty` when execution succeeds but returns zero rows, `runtime` when SQLite raises an engine-level exception, and `timeout` when execution exceeds the wall-clock budget. The label is retained together with the original SQL, execution feedback, and source-route metadata, so Block C can choose a type-specific repair path rather than treating all failed candidates as interchangeable correction requests.

D.3 Block C: Failure-Aware Repair

Each non-`clean` candidate in the active post-Phase2 pool is routed to one of four repair paths. All four paths share a common prompt skeleton over question, evidence, schema, current SQL, failure type, and failure feedback. They differ only in the concrete feedback and

in `type_specific_instructions` (Appendix F). Every repaired SQL is re-executed immediately. The repaired candidate replaces the original only if re-execution is `clean` (or `clean` with a non-empty result for `empty` repairs); otherwise the original is moved out of the pool and consolidation proceeds.

Validation repair. The implementation contains a validation-repair path for candidates whose current state is still marked invalid at repair time. In the normal runtime used in our main experiments, however, Phase 2 removes validation failures from the pool before Phase 4 begins, so repair budget is spent almost entirely on execution-surviving failures. When validation repair is invoked, the feedback slot lists the specific syntax or schema violations surfaced by Block A (e.g., unknown column, missing table, mismatched alias), and the type-specific instructions emphasize repairing the structural error while preserving the original query intent.

Runtime repair. The feedback slot carries the SQLite engine's error string together with any residual validation context. The instructions direct the LLM to address the exact engine-reported root cause (type mismatch, invalid function call, ambiguous column), not to restructure the whole query.

Timeout repair. The feedback slot records the execution elapsed time against the wall budget. The instructions prioritize eliminating missing `JOIN` conditions, Cartesian products, redundant nested subqueries, and other expensive structural patterns while keeping the semantic intent intact.

Empty-result repair: two-stage cascade. Empty-result is the most load-bearing failure class because, unlike runtime or timeout failures, the query is formally well-formed. The failure is semantic rather than structural. We therefore apply the two-stage cascade shown inside Block C of Figure 2: front-load cheap, high-precision structural edits and only escalate to an LLM when structural edits do not apply.

Stage 1: high-precision operator catalogue. Before any LLM call, the candidate is passed through a catalogue of six deterministic operators, each targeting a specific empty-result mechanism observed repeatedly in the training split:

- **`literal_binding`**: matches empty results caused by case, prefix, or substring mismatch between a literal in the `WHERE` clause and the actual stored value, and rewrites the predicate with a probed alternative literal.
- **`literal_rebind`**: similar to `literal_binding` but probes the database for candidate replacement values and picks the closest match under a database-backed similarity check.
- **`where_predicate`**: rewrites over-restrictive `WHERE` shapes such as a chain of equality predicates that should have been a single `LIKE`, or a nested subquery that should have been a direct `JOIN`.
- **`translation_semantics`**: handles natural-language words with multiple SQL semantics (e.g., "average" → `AVG` vs. `SUM/COUNT`), replacing the wrong aggregate with the intent-consistent one.
- **`date_anchor_join`**: catches empty results from date-range predicates on the wrong table in a multi-table schema, and rewrites the predicate onto the joined table along the identified foreign key.
- **`aggregate_scope`**: corrects `GROUP BY` or `HAVING` scope mismatches, for example moving a predicate from `WHERE` to `HAVING` when the column is an aggregate.

Each operator independently emits zero or more candidate repairs, and the first repair whose re-execution returns a non-empty result is accepted. If no operator produces a non-empty result, the candidate escalates to Stage 2.

Stage 2: error-bank LLM fallback. Stage 2 delegates to an iterative error-bank repair loop. The first round probes the database to produce a structured diagnosis (which literal value is missing, which join path is cold, which aggregate scope is wrong) and feeds the diagnosis to a probe-guided repair prompt. If the repaired SQL still returns empty, the loop issues a baseline rewrite prompt that instructs the LLM to draft a completely new SQL for the question while treating the previous attempt as a negative example. We cap the loop at two rounds[6] so that Stage 2 remains bounded in LLM cost.

[6] The cap is a wall-clock and cost guard rather than a modeling assumption. In pilot runs, additional rounds mostly repeated the same value-binding mistakes.

D.4 Block C: Post-Repair Rule Filtering and Pool Recovery

After re-execution confirmation and consolidation, Block C applies a small set of deterministic projection rules that *rewrite* noisy `SELECT` lists into more canonical forms. Two representative rules are:

- *Numbered-column deduplication.* If a projection contains columns with numeric suffixes alongside the base column, and the question does not request all variants, the suffixed columns are dropped.
- *Canonical-bundle reordering.* If a projection contains a standard column bundle in a non-canonical order, the columns are reordered to the form typically written by a human SQL author.

These rules encode regularities that humans typically would not write into hand-authored SQL. LLMs, in contrast, sometimes produce verbatim-complete column lists or unordered bundles when they over-pattern-match the schema. Applying these adjustments before selection reduces noise in the pool-level consensus signals of §3.5 without discarding otherwise valid queries.

If post-repair consolidation leaves the pool empty for a question, the system invokes *pool recovery* (Figure 2, Block C, rightmost box). The regeneration prompt does not consume the full historical removed set. Instead, it receives a condensed negative-feedback block built from (i) unresolved or still-invalid candidates at the end of Phase 4, (ii) candidates filtered by the Phase 5 rules, and (iii) the matched failure tags ρ attached to those removals. This keeps the feedback specific to the failure modes that actually exhausted the pool.

D.5 Additional Lifecycle Diagnostics

Per-model failure breakdown. Figure 9 gives the per-mechanism candidate-level repair rates broken down by generalist pairing: Sothis-32B + Qwen3-Coder, Sothis-32B + Qwen3-235B, and Sothis-32B + Gemini-3.1 Pro. Across all three pools the qualitative shape of Fig. 4 in the main text is preserved: validation failures and zero-candidate regeneration each resolve 100% of their cases, timeouts are resolved at 86–94% by the dual-pool consistency check, and empty-result repair is the hardest category with total resolution rates in 46–52%. The L HP row isolates the contribution of the high-precision structural operators (Stage 1 on empty-result) and the L LLM row shows the additional residue recovered by the LLM fallback (Stage 2). Empty-result repair depends mildly on the partner generalist because stronger generalists produce fewer empty-result failures in the first place (empty counts 1,178 → 1,021 → 1,043), but the relative share recovered by the Stage 1/Stage 2 cascade remains comparable across pools.

E L3: Confidence-Gated Selection

Algorithm 2 gives the full selection procedure before the score definitions and derivations.

Algorithm 2 Confidence-Gated Hybrid Selection

Require: pool $\mathcal{C}^{(*)}$, input $x = (q, e, d)$, schema s
Ensure: final SQL y

$\mathcal{C}^{+} \leftarrow \{c \in \mathcal{C}^{(*)} \mid \delta(c) = \texttt{clean}\}$
for all $c \in \mathcal{C}^{+}$ **do**
  compute tuple-level consensus $S_{\text{maj}}(c)$
end for
Partition $[\mathcal{C}^{+}]_{\sigma}$ by result signature
compute $S_{\text{rm}}^{\text{dedup}}$ over class representatives
broadcast $S_{\text{rm}}^{\text{dedup}}$ to class members
$S_{\text{coarse}}(c) \leftarrow S_{\text{maj}}(c) + S_{\text{rm}}(c)$
compute auxiliary consensus-refine statistic $S_{\text{ref}}(c)$
$c^{(1)} \leftarrow \arg\max_c S_{\text{coarse}}(c)$
$g^{(1)} \leftarrow$ result-signature class containing $c^{(1)}$
$y \leftarrow c^{(1)}$ ▷ default
if $\texttt{gate}(x)$ **holds (§3.5) then**
  extract $\kappa(c)$ for c in the tied top classes
  $g^{*} \leftarrow$ tied-top class with highest cross-source support under κ
  if $\text{supp}(g^{*}) > \text{supp}(g^{(1)})$ **then**
    $y \leftarrow \hat{c}_{g^{*}}$ ▷ structural-support override
  end if
end if
return y

This section expands L3 in §3.5 with the full derivations that are only summarized in the main text.

Confidence-gate thresholds. The cardinality bounds in the confidence gate (§3.5) are set to $K_{\max}$=6 and $T_{\max}$=8 in all reported runs. The three score-gap thresholds are all zero, so the selector escalates only when the merged score, the auxiliary consensus-refine score, and the RM score all leave the top candidate unresolved. These values are chosen so that the gate excludes both *over-fragmented* pools (many tiny result-signature classes with no dominant candidate, where the

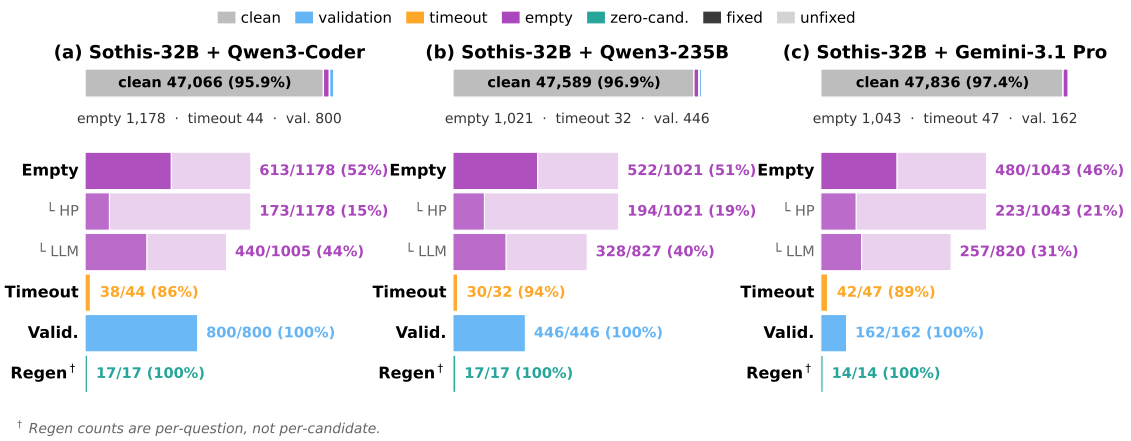

Figure 9: Per-model failure breakdown on BIRD dev.

structural-support prior becomes unreliable) and *over-concentrated* pools (one class already taking the majority of the pool, where $c^{(1)}$ is already the correct default). In practice the gate fires on roughly 5–10% of BIRD dev questions.[7]

E.1 Tuple-Level Consensus Derivation

Standard self-consistency for Text-to-SQL treats each candidate's execution result as an atomic signature and selects the one produced by the most candidates (Wang et al., 2025a). This whole-result voting fails when (i) no two candidates produce identical row sets, (ii) superficial differences such as column ordering or alias choice create spurious signature mismatches, or (iii) candidates that are partially correct should receive credit for the cells they do get right.

We address these limitations with a *tuple-level* consensus score. Given the set of successfully executed candidates $\mathcal{C}^+$, we first canonicalize each candidate's result into an order-independent representation $\mathrm{frozenset}(\mathrm{rows})$, then decompose it into cell-level triples:

$$\mathcal{T}(c) = \big\{(i, col_j, v_{ij}) \;\big|\; \text{row } i, \text{column } j \text{ of } \mathcal{E}(c,d)\big\}, \tag{14}$$

where $\mathcal{E}(c,d)$ denotes the execution result of candidate c on database d. The cross-pool frequency of a triple t measures how many candidates agree on that specific cell value:

$$f(t) = \big|\{c' \in \mathcal{C}^+ \mid t \in \mathcal{T}(c')\}\big|. \tag{15}$$

[7]This rate varies slightly with the generalist partner because candidate pools differ in the number and size of result-signature classes.

A candidate's consensus score then averages these frequencies over its own non-null triples:

$$S_{\mathrm{maj}}(c) = \frac{1}{|\mathcal{T}(c)|} \sum_{t \in \mathcal{T}(c)} f(t)\, \mathbf{1}[v(t) \neq \texttt{null}]. \tag{16}$$

Relationship to whole-result frequency counting. When every candidate's result set is either identical or completely disjoint, S_{maj} reduces to counting the number of candidates that share the same signature (scaled by the result size), recovering exact whole-result support counting as a special case. The advantage appears precisely when result sets overlap partially: a candidate that shares most cell values with the consensus receives a high score even if its overall row set differs due to, e.g., a single extra row or a different column alias.

Empty-result exclusion and null down-weighting. Candidates returning zero rows have $\mathcal{T}(c)=\varnothing$, so $S_{\mathrm{maj}}(c)=0$ by convention. This effectively excludes empty-result candidates from the consensus, which is the primary filtering mechanism: such candidates carry no execution evidence and should not compete with candidates that produce substantive output.

For non-empty candidates, the indicator $\mathbf{1}[v(t)\neq\texttt{null}]$ zeros out the frequency contribution of `null`-valued triples in the numerator, while the denominator $|\mathcal{T}(c)|$ still counts all triples including those with null values. The effect is that `null` cells *dilute* rather than inflate a candidate's score: a result set dominated by nulls receives a lower S_{maj} than one with the same number of non-null agreements. This is deliberate rather than dismissive of nulls: `null` values can reflect meaningful query logic (e.g., `LEFT JOIN` pro-

ducing `null` for unmatched rows), but because `null` cells are pervasive across structurally different queries, counting their agreement at full weight would inflate consensus scores without improving the selector's discriminative power.

E.2 Ablation Score Family: Freq, Majority Voting, Majority-Refine, and Merge

Table 4 in the main paper uses shorthand names for several selectors derived from the same clean pool $\mathcal{C}^+$. To avoid terminology ambiguity, we reserve *freq* for exact-result support counting, while *majority voting* denotes the tuple-level consensus score of Eq. 3, rather than exact-signature voting.

Exact-result support / *freq*. Let $\sigma(c)$ be the canonical result signature of candidate c. We define the exact-result support count as

$$N_\sigma(c) = \big|\{c' \in \mathcal{C}^+ \mid \sigma(c') = \sigma(c)\}\big|. \tag{17}$$

Its normalized frequency form is

$$S_{\text{freq}}(c) = \frac{N_\sigma(c)}{|\mathcal{C}^+|}. \tag{18}$$

Because the denominator is shared across candidates, ranking by S_{freq} is identical to ranking by N_σ. The *freq* selector is therefore

$$y_{\text{freq}} = \arg\max_{c \in \mathcal{C}^+} S_{\text{freq}}(c). \tag{19}$$

Tuple-level majority voting. Our *majority voting* ablation keeps only the tuple-level consensus signal:

$$y_{\text{vote}} = \arg\max_{c \in \mathcal{C}^+} S_{\text{maj}}(c). \tag{20}$$

This selector rewards candidates whose returned cells are repeatedly supported across the pool, even when no other candidate matches their full result signature exactly.

Majority-refine. The *majority-refine* statistic used in our implementation adds exact-result support on top of tuple-level consensus:

$$\begin{aligned} S_{\text{ref}}(c) &= N_\sigma(c) + S_{\text{maj}}(c), \\ y_{\text{ref}} &= \arg\max_{c \in \mathcal{C}^+} S_{\text{ref}}(c). \end{aligned} \tag{21}$$

Using the unnormalized count $N_\sigma(c)$ keeps this term on the same raw count scale as $S_{\text{maj}}(c)$. In the released selector code, the gate uses a closely related implementation-level quantity `maj_score_refine` produced by weighted majority voting; conceptually it serves the same role as a consensus-refine ambiguity check, even though its exact tie-break behavior is defined by the implementation rather than by the clean ablation formula above.

RM-only and *merge*. The pairwise preference ablation keeps only the position-debiased reward-model score:

$$y_{\text{rm}} = \arg\max_{c \in \mathcal{C}^+} S_{\text{rm}}(c). \tag{22}$$

Our coarse backbone, denoted *merge* in Table 4, fuses tuple-level consensus with pairwise Elo preference:

$$\begin{aligned} S_{\text{merge}}(c) &= S_{\text{maj}}(c) + S_{\text{rm}}(c), \\ y_{\text{merge}} &= \arg\max_{c \in \mathcal{C}^+} S_{\text{merge}}(c). \end{aligned} \tag{23}$$

The full system then applies the structural-support override (§3.5) only when the confidence gate fires.

E.3 Position-Debiased Elo Derivation

We first partition the successful pool $\mathcal{C}^+ = \{c \mid \delta(c) = \texttt{clean}\}$ into result-signature equivalence classes:

$$\begin{aligned} [\mathcal{C}^+]_\sigma = \big\{ &\{c \in \mathcal{C}^+ \,|\, \sigma(c) = s\} \\ &\big|\ s \in \mathrm{Im}(\sigma)\big\}, \end{aligned} \tag{24}$$

and pick one representative $\hat{c}_g$ per class. For every representative pair $(\hat{c}_i, \hat{c}_j)$ we build a comparison prompt containing the question, evidence, a schema summary restricted to tables referenced by either candidate, and the first k execution rows of each. Let $J_p(a, b) \in \{1, 0, \perp\}$ denote the judge's verdict for candidate a when it appears in position $p \in \{1, 2\}$ of a comparison against b ($\perp$ flags a parse failure). The position-debiased pairwise outcome for the ordered pair (i, j) is

$$\begin{aligned} \Delta_{i \succ j} = &\big[J_1(\hat{c}_i, \hat{c}_j) - J_2(\hat{c}_j, \hat{c}_i)\big]_+ \\ &+ \big[J_1(\hat{c}_j, \hat{c}_i) - J_2(\hat{c}_i, \hat{c}_j)\big]_-, \end{aligned} \tag{25}$$

where $[\cdot]_+$ and $[\cdot]_-$ denote the positive and negative parts. In practice we accumulate wins minus losses across all valid comparisons, obtaining the cumulative score

$$\begin{aligned} S_{\text{rm}}^{\text{dedup}}(\hat{c}_g) = \sum_{(\hat{c}_g, \hat{c}_h) \in \mathcal{P}} &\big(\mathbf{1}[J(\hat{c}_g) = \text{C}] \\ &- \mathbf{1}[J(\hat{c}_h) = \text{C}]\big), \end{aligned} \tag{26}$$

where C denotes the `Correct` verdict and $\mathcal{P}$ ranges over all $\binom{|[\mathcal{C}^+]_\sigma|}{2}$ valid representative pairs.

Pair label	Train		Val	
	Count	%	Count	%
`correct, correct`	7,298	9.2	788	75.6
`correct, incorrect`	28,166	35.3	127	12.2
`incorrect, correct`	28,166	35.3	127	12.2
`incorrect, incorrect`	16,170	20.3	–	–
Total	79,800	–	1,042	–

Table 9: Pairwise RM training data, all sourced from BIRD. Validation contains no `incorrect, incorrect` pairs by construction, as both candidates failing the execution gate indicate a degenerate pair that carries no preference signal.

The representative score is then broadcast back to class members:

$$S_{\mathrm{rm}}(c) = S_{\mathrm{rm}}^{\mathrm{dedup}}\big(\hat{c}_{g(c)}\big). \tag{27}$$

This deduplication-then-broadcast scheme keeps the number of LLM judge calls quadratic only in the number of distinct result signatures rather than in the raw pool size.

E.4 Pairwise Reward Model

The pairwise reward model used to compute S_{rm} in Eq. 4 is a fine-tuned Qwen2.5-Coder-32B model that, given two candidate SQLs together with their execution results, emits two independent correctness labels wrapped in `<sql1_judge>` and `<sql2_judge>` tags. Training has two stages: a supervised warm-start checkpoint trained on the same pairwise judgment format, followed by DAPO with a pairwise exact-match reward.

Data. We synthesize pairwise examples from question-level candidate pools. For each BIRD question we collect multiple candidate SQLs under several prompting strategies, execute each candidate on the target database to obtain an individual correctness tag, and then materialize ordered pairs $(\texttt{sql1}, \texttt{sql2})$ with label tuple $(y_1, y_2) \in \{\texttt{correct}, \texttt{incorrect}\}^2$. Table 9 summarizes the final train/validation split and its realized label distribution. The four-class distribution is deliberate: an effective pairwise judge must remain accurate when both candidates are wrong (20.3% of training pairs), not only when one is clearly better. The equal counts of `correct,incorrect` and `incorrect,correct` in the training split further indicate that both orderings are retained rather than collapsed to a single example per unordered pair.

Setting	Value
Rollout n	16
Rollout temperature	1.0
Clip ratio low / high	0.2 / 0.28
KL regularization	disabled
Entropy target H_{target}	0.15
Entropy coef. β	5×10^{-3}
Max prompt / response	16,384 / 1,024 tokens
Overlong buffer	128 tokens, factor 1.0
Train / gen / mini bsz	128 / 384 / 128
Learning rate	1×10^{-5}
LR warmup steps	10
Weight decay	0.1
Gradient clip	1.0
Total training steps	1,000

Table 10: Reward-model DAPO hyper-parameters. Longer prompts and shorter responses (relative to the specialist) reflect the pairwise judgment format: two SQLs with their execution results in, a pair of correctness tags out.

Supervised warm-start. We first fine-tune Qwen2.5-Coder-32B on the pairwise judgment format of Listing 8 with cross-entropy on the ground-truth `<sql1_judge>` and `<sql2_judge>` tags as targets. We use per-device batch size 1, gradient accumulation 2, learning rate 1×10^{-5} with cosine decay, and train for 3 epochs. This stage teaches the RM to emit a well-formed, parseable judgment before RL introduces reward sparsity.

Reinforcement learning with DAPO. We train the RM with DAPO (Yu et al., 2025) under a binary pairwise exact-match reward. Let $j_1(y)$ and $j_2(y)$ denote the predicted correctness tags parsed from `<sql1_judge>` and `<sql2_judge>`, and $j_1^\star, j_2^\star$ the corresponding gold tags. The reward is

$$R_{\mathrm{rm}}(y) = \mathbf{1}[j_1(y){=}j_1^\star] \cdot \mathbf{1}[j_2(y){=}j_2^\star]. \tag{28}$$

The indicator product means partial credit is not awarded: only trajectories that correctly label *both* SQLs receive reward 1, which forces the RM to reason about each SQL on its own merits rather than anchoring on the obviously wrong one. Table 10 lists the RM-specific hyper-parameters. The DAPO objective, dynamic sampling, and threshold-gated entropy follow §C.2.

E.5 Structural Signature Features

The AST-level structural signature $\kappa(c)$ used by the structural-support reranker concatenates the following features extracted from the parsed SQL:

- $\mathsf{tables}(c)$: sorted tuple of base table names in FROM/JOIN.

- proj_kinds(c): tuple of projection kinds (e.g., COUNT, SUM, AVG, plain column).
- proj_count(c): number of projected columns.
- has_distinct(c) and has_count_distinct(c): whether the query uses DISTINCT or COUNT(DISTINCT ·).
- has_group_by(c) and group_by_count(c): whether the query uses GROUP BY and how many grouping expressions it uses.
- has_order_by(c) and order_dirs(c): whether the query uses ORDER BY and, if so, the direction tuple.
- limit_bucket(c): a coarse LIMIT bucket with three values: no limit, limit equals 1, and limit greater than 1.
- proj_cols(c): projected source-column tuple from SELECT.
- pred_cols(c): set of columns appearing in WHERE predicates.

The structural-support reranker of §3.5 uses $\kappa(c)$ to measure *cross-source consensus*: for each candidate class tied in the coarse score, we count how many distinct source groups (generator–route identities, with model-level fallback) contributed a candidate with the same κ, and the class with the highest such count wins the override if its support is strictly stronger than the baseline tied winner. Unlike a structural clustering, we do not build per-cluster evidence summaries or invoke any LLM judge. The entire reranker is deterministic.

F Prompt Templates

This section lists representative prompt templates from the runtime pipeline. They are grouped by stage (generation, correction, and selection), and the section ends with the pairwise reward-model training prompt. Variable slots ({...}) are preserved, and role wrappers are normalized to a unified SYSTEM/USER presentation. Implementation-level variants are available in the code release.

F.1 Generation Prompts

The specialist generator uses two prompt variants: a schema-only variant (agentar_primary) and a few-shot variant (secondary_fewshot) that embeds retrieved demonstrations.

Listing 3: Schema-only generation prompt (agentar_primary).

```
SYSTEM:
You are a data science expert. Below,
    you are provided
with a database schema and a natural
    language question.
Your task is to understand the schema
    and generate a
valid SQL query to answer the question.

USER:
Database Engine:
SQLite

Database Schema:
{schema}

Question:
{question_with_knowledge}

Instructions:
- Only output the information asked in
    the question.
- If the question asks for a specific
    column, include
  only that column in the SELECT clause.
- Never use SELECT *.
- Return all and only the requested
    information.
- Think step by step before producing
    the final SQL.
{dynamic_instructions}

Output Format:
```sql
-- Your SQL query
```
```

The secondary_fewshot variant is identical except that it inserts a Contextual Examples block ({few_shot_examples}) between the question and the instructions, with a note that example schemas may differ and the model should focus on query structure rather than specific names.

F.2 Correction Prompts

We use one shared correction template for validation, runtime, and timeout failures, parameterized by a failure type and a small set of type-specific instructions. Listing 4 shows the shared skeleton. Empty-result failures follow a two-stage procedure: the first stage is operator-based and invokes no LLM, and the second stage has two LLM variants, a *baseline rewrite* and a *probe-guided targeted repair*.

Listing 4: Shared template for validation, runtime, and timeout repairs.

```
SYSTEM:
You are an expert SQLite SQL correction
    assistant.
```

```
Given the question, evidence, database
    schema, current
code, and failure feedback, reason step
    by step and
produce one corrected SQL query that
    preserves the
original intent and is executable on
    SQLite.

USER:
Database Engine: SQLite
Question: {question}
Evidence: {evidence}
Value Link Hints:
    {value_link_prompt_hints}
Database Schema: {schema_description}
Current Code: {candidate_sql}
Failure Type: {failure_type}
Failure Feedback: {failure_feedback}
Instructions:
    {type_specific_instructions}

Output Format:
Return only one corrected SQL query
    inside a single
```sql``` code block.
```

The three failure types differ only in `failure_feedback` and `type_specific_instructions`:

- **Validation.** Feedback lists the specific syntax/schema errors (`{errors}`). Instructions emphasize repairing syntax or schema validation issues, ensuring every name matches the schema, and preserving the original intent.
- **Runtime.** Feedback includes the SQLite error string `{exec_error}` together with any additional validation context. Instructions emphasize fixing the exact root cause signaled by the engine while keeping the question intent intact.
- **Timeout.** Feedback notes that execution exceeded `{timeout_seconds}` seconds. Instructions prioritize eliminating missing JOIN conditions, Cartesian products, and expensive nested subqueries without changing semantics.

Listing 5: Stage-2 empty-result repair: baseline rewrite.

```
SYSTEM:
You are an expert SQLite SQL writer.
Write a completely new SQL query from
    scratch.

USER:
Database Engine: SQLite
Question: {question}
Evidence: {evidence}
Database Schema:
{schema}
{db_facts}
{history}

A previous SQL returned 0 rows (for
    reference only):
--- failed SQL ---
{sql}
--- end ---

Task:
- Write a COMPLETELY NEW SQL query from
    scratch.
- Do NOT modify the failed SQL.
    Approach fresh.

Output Format:
Return only one SQL query inside a
    single ```sql```
code block.
```

Listing 6: Stage-2 empty-result repair: probe-guided targeted fix.

```
SYSTEM:
You are an expert SQLite SQL debugger.
Apply a targeted fix based on the
    probing results.

USER:
Database Engine: SQLite
Question: {question}
Evidence: {evidence}
Database Schema:
{schema}
{db_facts}
{history}
{literal_guardrails}

A previous SQL returned 0 rows:
SQL to fix:
{sql}

Database Probing:
{diagnosis}

Task:
{probe_instruction}
{repair_rules}

Output Format:
Return only one corrected SQL query
    inside a single
```sql``` code block.
```

F.3 Selection Prompts

Selection invokes only one LLM prompt: the pairwise reward-model judge that produces S_{rm} for Eq. 4. The structural-support reranker is deterministic at selection time and uses no LLM prompt.

Listing 7: Pairwise reward-model judge.

```
SYSTEM:
You are a SQL evaluator.
Determine which of two SQL responses
    correctly answers
```

```
the user question based on the schema.

Evaluation Protocol:
1. Comparative Diff: identify the exact
    logic
   difference between SQL A and SQL B.
2. Constraint Necessity: if one SQL
    includes a filter
   that the other omits, verify whether
    the question
   explicitly requests it. The SQL that
    answers with
   the fewest unsupported assumptions
    is usually
   correct.
3. Logic over Results: judge based on
    code logic
   mapping to the question. Do NOT
    conclude
   correctness merely because the
    execution result
   looks right.

Output Format:
1. Difference Analysis: [Briefly
    describe the difference]
2. Logic Check: [For each SQL, state
    whether its logic
   is fully supported by the question
    text]
3. Final Judgment:
   <sql1_judge> Correct/Incorrect
    </sql1_judge>
   <sql2_judge> Correct/Incorrect
    </sql2_judge>

USER:
Database Schema: {schema}
Question: {question}
Evidence: {evidence}

Responses:
<START_OF_RESPONSE_1>
-- SQL: {sql1}
-- Execution Result #rows: {num_rows_1}
-- Execution Result START
{result_1}
-- END Execution Result
<END_OF_RESPONSE_1>

<START_OF_RESPONSE_2>
-- SQL: {sql2}
-- Execution Result #rows: {num_rows_2}
-- Execution Result START
{result_2}
-- END Execution Result
<END_OF_RESPONSE_2>
```

F.4 Pairwise RM Training Prompt

The RM is trained on the following pairwise judgment prompt, which shares the same structure as the inference-time judge in Listing 7: schema, question, optional evidence, two SQL responses with their execution results, and a pair of final correctness tags.

Listing 8: Pairwise RM training prompt.

```
USER:
# Task Objective:
Evaluate the correctness of two
    responses (Response A
and Response B) to a given question.
    You are given a
database schema, question, external
    knowledge, and two
SQLs with their execution results.

# Steps to Follow:
0. Extract answers from both responses.
1. For each response, verify schema
    fidelity, SQLite
   syntax, SELECT-only compliance, and
    question-level
   logical correctness.
2. Output a final correctness judgment
    for both
   responses.

Final Judgment:
<sql1_judge>Correct/Incorrect</sql1_judge>
<sql2_judge>Correct/Incorrect</sql2_judge>

# Input:
Task Description:
-- Database Schema:
{schema}
-- Question:
{question}
-- External Knowledge:
{evidence}

Responses:
<START_OF_RESPONSE_1>
-- SQL: {sql1}
-- Execution Result #rows: {num_rows_1}
-- Execution Result START
{result_1}
-- END Execution Result
<END_OF_RESPONSE_1>

<START_OF_RESPONSE_2>
-- SQL: {sql2}
-- Execution Result #rows: {num_rows_2}
-- Execution Result START
{result_2}
-- END Execution Result
<END_OF_RESPONSE_2>
```

G Extended Discussion

This appendix expands the brief discussion stub at the end of §5.

G.1 The Coverage–Consensus Trade-off

Our experiments surface a recurring pattern between specialist and generalist candidate sources. Pools drawn from a single training distribution either reach a wider set of correct queries but cannot vote them out (high oracle, low majority), or they vote consistently but cannot reach the correct query in the first place (high majority, low oracle).

SOTHIS-SQL is effective on BIRD not because it produces a strictly larger pool, but because pairing distributions sitting on opposite ends of this trade-off lifts both the ceiling and the voteability simultaneously (RQ3). We expect this pattern to generalize to any setting where a domain-specialized RL-trained model can be paired with a broader LLM, beyond Text-to-SQL.

G.2 Coarse Repair Can Pollute the Pool

A second observation from RQ4 is more cautionary: handing failed candidates to a single LLM-based corrector without a structured per-type signal can be *worse* than dropping those candidates entirely (*coarse LLM repair* vs *drop failures*). This suggests that the value of execution feedback in pool-level pipelines lies in the structured diagnosis it carries, not in the LLM call it triggers: uniform LLM repair injects plausibly-clean noise that subsequent voting cannot filter out. Practitioners adding self-correction to multi-candidate pipelines should preserve the failure type rather than collapse it.

G.3 Efficiency–Specificity Trade-off in the Hybrid Selector

The hybrid selector's design embodies a deliberate efficiency–specificity trade-off: most questions are resolved by lightweight result-level signals, and the deterministic structural-support reranker fires only on questions where these signals are tied.