RESIDUAL SKILL OPTIMIZATION FOR TEXT-TO-SQL ENSEMBLES

Anonymous authors
Paper under double-blind review

ABSTRACT

Text-to-SQL ensembles improve over single-candidate generation by drawing multiple SQL candidates and selecting one, but their effectiveness is bounded by Pass@K, the probability that at least one of K candidates is correct. Existing methods source diversity heuristically through stochastic decoding or prompt variants, leaving candidate sets dominated by correlated failures. We present DIVSKILL-SQL, a residual skill optimization framework that builds complementary agentic Text-to-SQL ensembles without model fine-tuning: each new skill is optimized on examples the current skill ensemble fails on, provably targeting its marginal contribution to Pass@K. On Spider2-Lite, DIVSKILL-SQL improves selected accuracy by up to $+11.1$ points on Snowflake and $+8.3$ on BigQuery over the strongest ensemble baseline, with consistent gains across two base models (Opus-4.6 and GPT-5.4). Skills optimized on a single dialect transfer without retraining across dialects (Snowflake, BigQuery, SQLite) and to a different task formulation, such as BIRD-Critic ($+2.6$ pts). Error diagnostics show up to $3\times$ fewer hallucinated schema references and function calls, indicating that gains come from genuinely reliable complementary skills rather than surface-form variation.

1 INTRODUCTION

Text-to-SQL translates natural language questions into executable SQL queries, making relational databases accessible without SQL expertise. While large language models (LLMs) achieve strong results on standard benchmarks (Yu et al., 2018; Pourreza & Rafiei, 2023b), real-world queries, which involve large schemas, dialect-specific syntax, and multi-step query logic, remain difficult. This has motivated agentic Text-to-SQL systems that inspect schemas, execute intermediate queries, observe feedback, and iteratively repair errors (Talaei et al., 2024; Deng et al., 2025).

Yet even with multi-step interaction, no single agentic execution reliably solves every query, which has motivated ensembling as a complementary axis of robustness (Pourreza et al., 2024; Yang et al., 2025b; Liu et al., 2026; Deng et al., 2025): rather than committing to a single SQL generation, ensembles generate multiple SQL candidates and select a final answer. The potential of an ensemble is bounded above by Pass@K, the probability that at least one of K candidates is correct (Chen et al., 2021): no selector, however good, can recover an answer that was never generated. Pass@K therefore captures what the generation stage is responsible for, separately from selection.

Existing Text-to-SQL ensembles produce candidate diversity by combining stochastic decoding, hand-designed prompt or workflow variants, and in some cases multiple fine-tuned generators (Pourreza et al., 2024; Liu et al., 2026). Whether the resulting candidates recover different failure cases, however, is left to chance: nothing in the procedure pushes new candidates to solve examples that earlier ones miss. In agentic systems this is especially fragile: randomness in early planning propagates through long reasoning trajectories, so high-temperature sampling produces noisier variants of the same path rather than complementary solutions, and introduces unstable reasoning, spurious joins, and dialect errors that degrade the candidate pool (Pourreza et al., 2025). Pass@K accordingly plateaus, with little gain from additional candidates.

To address this, we introduce DIVSKILL-SQL, a residual skill optimization framework that makes candidate complementarity an explicit optimization target instead of leaving it to heuristics or chance. A *skill* is a high-level instruction file that controls the agent's decomposition style, schema-exploration policy, drafting strategy, and repair logic; DIVSKILL-SQL operates entirely at this prompt level,

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

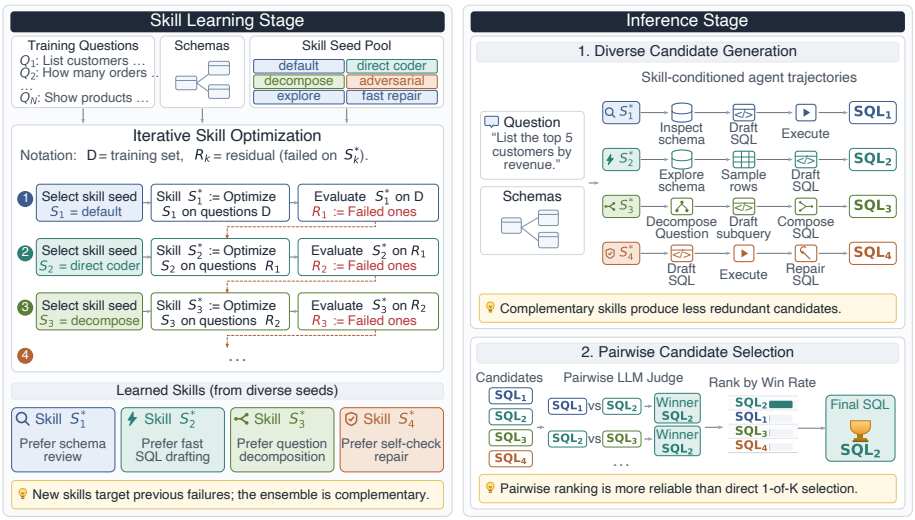

Figure 1: System diagram of DIVSKILL-SQL

with no model fine-tuning. Starting from a base skill, DIVSKILL-SQL evaluates the agent on the training set, identifies the unresolved examples, and uses reflective prompt optimization (Agrawal et al., 2025) to refine a new skill specifically on this residual; it repeats round by round until a pool of K complementary skills has been learned. A new skill need not to be globally better than its predecessors; it is useful precisely when it recovers examples they miss. At inference, DIVSKILL-SQL runs each learned skill and selects a final SQL from the resulting candidate set via pairwise comparison (Pourreza et al., 2024). Each skill is trained to cover what the others miss, so the ensemble is complementary by construction rather than by chance.

We evaluate DIVSKILL-SQL on recent Text-to-SQL benchmarks across four dialects: Spider2-Lite over SQLite, Snowflake, and BigQuery, and BIRD-Critic over PostgreSQL. DIVSKILL-SQL improves selected accuracy on Spider2-Lite by up to **+11.1** points on Snowflake and **+8.3** on BigQuery over the strongest ensemble baseline, with consistent gains across two base models (Opus-4.6 and GPT-5.4). Skills optimized on standard Text-to-SQL in a single dialect (Snowflake) transfer without retraining to BigQuery and SQLite. Moreover, skills optimized for standard Text-to-SQL generalize to the debugging-style generation setting of BIRD-Critic, where DIVSKILL-SQL improves accuracy by **+2.6**. Error diagnostics show **up to 3**$\times$ fewer hallucinated schema references and unsupported-function calls, and tool-use trajectory analysis shows that DIVSKILL-SQL reduces redundancy in agent behavior by 19%–28%, producing more diverse schema-inspection, decomposition, drafting, execution, and repair patterns than repeated runs of the same agent.

Our contributions are as follows:

- We formulate Text-to-SQL ensembling as residual Pass@K optimization over agent skills, where the goal is to learn complementary behaviors that cover different failure modes.
- We propose DIVSKILL-SQL, a residual skill optimization framework that improves candidate-set coverage without model fine-tuning or high-temperature sampling.
- We show that optimizing each new skill on unresolved examples directly targets its marginal contribution to Pass@K.
- We evaluate DIVSKILL-SQL across recent Text-to-SQL benchmarks and four SQL dialects, showing improved accuracy and reduced redundancy in agent trajectories.

2 RELATED WORKS

Pass@K and skill optimization. Pass@K evaluates whether at least one of K sampled solutions is correct (Chen et al., 2021), making it a natural objective for tasks where multiple attempts can be generated and selected. Recent work shows that optimizing only Pass@1 does not necessarily improve Pass@K: RL with verifiable rewards can increase single-sample accuracy while reducing output diversity and leaving hard problems outside the model's support (Yue et al., 2025). Several methods therefore optimize diversity or Pass@K directly through policy updates (Chen et al., 2025b; Walder & Karkhanis, 2025; Yao et al., 2025), but these approaches require modifying model parameters

and are limited to open-weight models. A related line of work optimizes prompts or agent skills from feedback (Shin et al., 2020; Zhou et al., 2022; Yang et al., 2023; Agrawal et al., 2025; Zhang et al., 2026a), including systems that revise, evolve, or jointly train skills (Zhou et al., 2026; Xia et al., 2026; Ma et al., 2026; Zhang et al., 2026b;c). However, most prompt and skill optimization methods seek a single stronger prompt or agent, rather than a set of generators whose successes are complementary. Our method instead learns a set of residual skills that explicitly target failures left by the current ensemble, improving Pass@K coverage without weight updates and remaining applicable to both open and closed models.

Text-to-SQL systems and ensembles. LLM-based Text-to-SQL has progressed from direct SQL generation (Dong et al., 2023; Tai et al., 2023; Li et al., 2024a;b; Chen et al., 2025a) to agentic systems that decompose schema understanding, evidence extraction, query generation, execution, and refinement (Pourreza & Rafiei, 2023a; Gao et al., 2023; Wang et al., 2025; Talaei et al., 2024; Xie et al., 2024; Dai et al., 2025; Zhang et al., 2025). Because no single generation path reliably solves all queries, recent systems use ensembling to improve robustness (Pourreza et al., 2024; Deng et al., 2025; Yang et al., 2025b; Liu et al., 2026; Lee et al., 2025; Guo et al., 2025): CHASE-SQL combines candidates from diverse prompts and selects through a tournament (Pourreza et al., 2024); MARS-SQL trains a multi-agent system with RL (Yang et al., 2025b; MA et al., 2026; Zhai et al., 2025); and XiYan-SQL fine-tunes multiple generators to induce diversity (Liu et al., 2026; Yang et al., 2024). These methods demonstrate the value of multiple candidate generators, but their diversity is either manually designed, selector-driven, or obtained through model training. In contrast, our method directly optimizes the generators themselves for complementary residual coverage, constructing a Pass@K-oriented skill ensemble without hand-designed prompt sets or model weight modification.

A detailed literature review is provided in Section A.

3 METHOD

3.1 SETUP AND NOTATION

Text-to-SQL with agentic execution. A Text-to-SQL instance is a pair $(q, \mathcal{D})$ consisting of a natural-language question q and a relational database $\mathcal{D}$, and the goal is to produce an executable SQL query whose result on $\mathcal{D}$ matches that of a gold reference. We follow the agentic execution paradigm of recent Text-to-SQL systems (Talaei et al., 2024; Deng et al., 2025): rather than generating SQL in a single forward pass, an *agent* interleaves tool calls, such as inspecting schema, sampling rows, drafting candidate SQL, and repairing errors, over multiple steps before returning a final query.

Skills. We modulate the agent's behavior through *skills*. A skill s is a high-level instruction file (a system prompt expressed in natural language) that controls the agent's reasoning and tool-use policy: which decomposition style to favor, when to explore the schema versus draft directly, what repair patterns to apply on execution errors, and so on. We write a_s for the agent equipped with skill s and treat s as identified with its prompt π_s, so that the optimization space $\mathcal{S}$ is the space of natural-language instruction files. Two distinct skills induce two genuinely different agent trajectories on the same input, not merely two stochastic samples of the same trajectory.

Example 3.1 (Skill Examples)**.** We show two simplified skills below that differ not only in wording but in the agent trajectory they encourage: `decompose` delays final SQL generation until the query logic has been broken into verified subcomponents, whereas `direct_coder` pushes the agent to draft early and rely on execution feedback for rapid repair.

decompose skill

```
Break complex questions into simple
subqueries, build bottom-up.
1.  PARSE the question into atomic
requirements.
2.  BUILD each piece as a standalone CTE.
3.  COMPOSE CTEs into the final query
using WITH...SELECT.
```

direct_coder skill

```
You are an EFFICIENT SQL writer.  Write
SQL quickly, test, iterate.
1.  Read the question carefully.  Identify
the core tables, joins, and aggregations.
2.  Write your best SQL attempt
IMMEDIATELY based on the schema.
3.  Execute it.  If errors occur, read the
error message carefully and fix it.
```

Notation. Let $\mathcal{X}$ denote the space of input tasks $(q, \mathcal{D})$ and let P be the underlying task distribution. For a skill $s \in \mathcal{S}$, we write $p_s(x) \in [0, 1]$ for the probability that one execution of a_s on input $x \in \mathcal{X}$ produces a correct SQL query (i.e., a query whose execution result matches the gold reference). For a finite training set $D_{\text{train}} \subseteq \mathcal{X}$ and a subset $R \subseteq D_{\text{train}}$, we write

Algorithm 1 DIVSKILL-SQL: batch-sequential residual skill optimization.

Require: training question pool D_{train}; seed skill pool $\mathcal{S}_0 = \{s_1, \ldots, s_K\}$ with prompts $\{\pi_s^{(0)}\}_{s \in \mathcal{S}_0}$; batch size b; number of batches T; skill optimizer $\text{SkillOptimizer}(\pi, R, V)$ that returns a refined prompt with highest accuracy on V.

$\pi_s \leftarrow \pi_s^{(0)}$ for each $s \in \mathcal{S}_0$ ▷ initial seed pool
for $t = 1, \ldots, T$ **do**
  sample $B_t \subseteq D_{\text{train}}$ with $|B_t| = b$, and split into disjoint B_t^{tr} and B_t^{val}
  $R_{t,0} \leftarrow B_t^{\text{tr}}$ ▷ initial training residual
  choose an ordering $\sigma_t : [K] \to \mathcal{S}_0$ ▷ without replacement
  for $j = 1, \ldots, K$ **do**
    $s \leftarrow \sigma_t(j)$ ▷ draw next seed
    $\widetilde{\pi}_s \leftarrow \text{SkillOptimizer}(\pi_s; R_{t,j-1}, B_t^{\text{val}})$ ▷ optimize on residual
    if $\hat{p}_{\widetilde{\pi}_s}(B_t^{\text{val}}) > \hat{p}_{\pi_s}(B_t^{\text{val}})$ **then**
      $\pi_s \leftarrow \widetilde{\pi}_s$ ▷ accept on validation
    end if
    $R_{t,j} \leftarrow \{\, x \in R_{t,j-1} : a_s \text{ fails on } x \,\}$ ▷ use retained π_s
  end for
  $\pi_s^{(t)} \leftarrow \pi_s$ for each $s \in \mathcal{S}_0$ ▷ update seed pool
end for
return $\{\pi_s^{(T)}\}_{s \in \mathcal{S}_0}$

$$\hat{p}_s(R) \;=\; \tfrac{1}{|R|} \sum\nolimits_{x \in R} p_s(x)$$

for the empirical success rate of skill s on R. For a collection of K skills $A = \{s_1, \ldots, s_K\}$, the population Pass@K, defined as the probability that at least one of the K corresponding agent executions succeeds, is

$$\text{Pass@K}(A) \;=\; \mathbb{E}_{x \sim P}\left[1 - \prod\nolimits_{j=1}^{K} (1 - p_{s_j}(x))\right].$$

3.2 RESIDUAL SKILL OPTIMIZATION

The residual principle. As demonstrated in Figure 1, we construct an ensemble of K skills sequentially. After selecting skills $s_1, \ldots, s_{j-1}$, we define the residual training set

$$R_{j-1} \;=\; \{\, x_i \in D_{\text{train}} : a_{s_\ell} \text{ fails on } x_i \;\; \forall \ell < j \,\},$$

and pick the next skill by maximizing success on this residual:

$$s_j \;\in\; \arg\max\nolimits_{s \in \mathcal{S}} \; \hat{p}_s(R_{j-1}).$$

Later skills are therefore not pushed to be globally better than earlier ones. They are pushed to cover examples the current ensemble misses—which is exactly the marginal contribution of a new skill to Pass@K. This is the mechanism by which the procedure encourages complementary skills and directly targets ensemble coverage rather than average accuracy.

The DIVSKILL-SQL algorithm. The residual arg-max above is an idealized objective: it assumes access to the full training distribution and optimization over the infinite space of natural-language skills. DIVSKILL-SQL turns this principle into a practical batch-sequential process presented in Algorithm 1. At each of T rounds, DIVSKILL-SQL draws a batch $B_t \subseteq D_{\text{train}}$ and splits it into disjoint training and validation portions, B_t^{tr} and B_t^{val}. Skills are visited in order σ_t. Each skill is refined using the training residual $R_{t,j-1}$, containing examples not solved by preceding skills in the current round. A proposed update is accepted only if it improves execution accuracy on B_t^{val}. The retained prompt is then used to remove solved examples from the training residual. Accepted updates are carried forward to subsequent rounds. The algorithm has two main ingredients that make the ideal residual arg-max practical: a finite set of *diverse seed skills* that defines the initial search space, and an *inner-loop optimizer* that refines each seed on the current residual failures.

Skill seed pool. The arg-max in $\arg\max_{s \in \mathcal{S}} \hat{p}_s(R_{j-1})$ is over an infinite, unstructured space of natural-language prompts and is intractable directly. To make the inner optimization tractable, we initialize from a small *seed pool* $\mathcal{S}_0 = \{s_1^{(0)}, \ldots, s_K^{(0)}\}$ of K LLM-assisted, manually curated skills, each encoding a distinct high-level reasoning strategy. Concretely, we first run a generic LLM agent on a subset of the training data and inspect representative and recurring failure modes, and propose diverse strategy prompts from several perspectives: whether to decompose the question before coding, how much schema and value exploration to perform before drafting SQL, etc. We inspect and keep

only strategies with distinct intended trajectories, removing near-duplicates that differ only in wording. Example 3.1 presents two simple, high-level skills we adopted as seeds, and Appendix E provides the full list of seed skills. Subsequent rounds optimize prompts *starting from* the seed pool, restricting the effective search space to natural refinements of these high-level strategies.

Inner-loop optimization. The inner loop uses an LLM to propose refined skill prompts based on observed agent failures, a standard reflective prompt-optimization step (Fernando et al., 2023; Zhou et al., 2022; Yang et al., 2023; Agrawal et al., 2025). The reflector receives failure traces sampled from executions of the current prompt π_s on the training residual $R_{t,j-1}$ and proposes revised prompts. Each proposal is first checked on the sampled training failures; proposals that fail this repair check are discarded. The remaining proposals are evaluated on the disjoint validation portion B_t^{val}, and the candidate with the highest validation accuracy is returned. If no proposal passes the repair check, the current prompt is returned. Algorithm 1 retains the returned candidate only if its validation accuracy exceeds that of the current prompt.

The following example of `decompose` skill evolution illustrates how residual optimization turns diversity in the seed pool into more targeted coverage in the inner-loop optimization.

Example 3.2 (Decompose Skill Evolution)**.** The `decompose` seed skill in Example 3.1 already differs from skills such as `direct_coder`: it encourages the agent to solve a query by breaking it into subproblems rather than drafting SQL immediately and repairing it based on execution feedback. However, the seed remains too high-level to specify what the agent should verify during decomposition. When optimized on the residual set, the prompt is refined precisely around the frequent kinds of errors that other skills leave unresolved. The example below shows the optimized `decompose` skill.

optimized decompose skill

```
Break complex questions into simple subqueries, build bottom-up.

Step 1: PARSE the question into atomic requirements
- What is the output? columns, derived metrics, ratios, counts, sums.
- What is the grain? one row per what? month? patient? group?
- What grouping dimensions are needed? only those that match the desired grain.
- Is there ranking, ordering, limiting, or a ratio/composition calculation?

Step 2: ANCHOR the grain before grouping
- Match GROUP BY columns precisely to the output grain: no more, no less.
- If the question asks for monthly totals, group by month only; do not add
  route, city, or other columns unless explicitly requested.
- If the question asks for a split, identify the correct column and values that
  represent that split; do not substitute a loosely related column.

Step 3: BUILD each piece as a standalone CTE
- Add joins one at a time, verifying row counts do not explode.
- For ratio/composition queries, compute totals in one CTE, subgroup counts in
  another, then JOIN and divide.

Step 4: VERIFY join logic and filter semantics
- Confirm join keys actually link the intended entities; avoid fan-out.
- Confirm filter values match the domain as they appear in the data.
```

After optimization, it evolves from generic advice to "build CTEs" into a concrete rubric: identify the intended output grain before grouping, match `GROUP BY` columns to that grain, choose the correct column for requested splits, and compute ratio/composition queries using compatible aggregation levels. Thus, the learned skill is not merely a more detailed version of the seed prompt; its added details are shaped by the residual cases it is meant to cover. The full before/after skill comparison pool is provided in Appendix E.

In practice, we further improve the finite-batch learning procedure through careful reflection prompt and reward design, rotating the skill order, etc.; Appendix D.2 discusses these implementation practices in detail.

Population-level guarantee. Under the population-level objective, residual skill optimization is greedy maximization of the Pass@K coverage objective. Formally, Proposition C.1 (stated and proved in Appendix C.1) shows that, in the population limit,

$$\text{Pass@K}(\{s_1, \ldots, s_K\}) \;\geq\; (1 - 1/e) \max_{|A| \leq K} \text{Pass@K}(A).$$

Thus, under exact greedy selection in the population setting, the resulting skill bank is guaranteed to achieve at least a constant-factor fraction of the best possible K-skill ensemble. The intuition is that the marginal value of a new skill lies in its ability to solve problems that the current skill set still fails to address. Residual optimization therefore greedily adds the skill with the largest additional coverage

of the remaining failure, yielding the standard approximation guarantee for monotone submodular maximization (Nemhauser et al., 1978).

3.3 INFERENCE

At inference time, the K learned skill-conditioned agents run in parallel on each test instance, producing K candidate SQL queries. We select the final query using pairwise candidate comparisons following Pourreza et al. (2024). Since pairwise comparison scales quadratically in K, we first deduplicate candidates by execution output: queries returning identical results on the target database are collapsed into one equivalence class, with one representative retained. This leaves $G \leq K$ candidates and reduces comparisons when multiple skills agree. We then run an exhaustive round-robin over all $\binom{G}{2}$ unordered pairs, rather than sampling pairs, because G is small in practice.[1] To mitigate LLM judge position bias, each pair (i, j) is judged twice with swapped presentation order. Each judgment gives one win to the selected candidate, and we return the candidate with the highest win count, breaking ties arbitrarily.

4 EXPERIMENTS

In the experiments, we evaluate DIVSKILL-SQL through three research questions:

> **RQ1: End-to-end effectiveness.** How does DIVSKILL-SQL compare with state-of-the-art Text-to-SQL systems on recent complex benchmarks?
> **RQ2: Transferability.** Do skills optimized on a single dataset or SQL dialect transfer to unseen datasets, alternative dialects, and new task formats?
> **RQ3: Behavioral diversity.** How does residual skill optimization change the behavior of an agentic Text-to-SQL system, beyond simply changing final SQL strings?

4.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate DIVSKILL-SQL on two recent Text-to-SQL benchmarks that provide clean ground-truth annotations and involve complex reasoning. First, we use Spider2-Lite (Lei et al., 2024), which tests complex SQL generation over realistic schemas and multiple dialects. We report results on its SQLite, Snowflake, and BigQuery subsets, with 135, 207, and 205 examples respectively. These subsets differ in schema organization, query style, and dialect-specific syntax, allowing us to study both in-domain performance and cross-dialect transfer. Second, we evaluate on BIRD-Critic (Li et al., 2025), a SQL debugging benchmark derived from BIRD (Li et al., 2024c). Each instance provides a natural-language issue description, a buggy SQL query, and database context; the system must diagnose and repair the query rather than generate SQL from scratch. This setting tests whether DIVSKILL-SQL transfers beyond direct SQL generation to agentic SQL correction. We evaluate on its pure PostgreSQL version.

Skill optimization. We adopt the state-of-the-art prompt and skill optimization technique GEPA (Agrawal et al., 2025) for skill optimization. For the Spider2-Lite benchmark, we learn skills from only approximately 200 examples sampled from proprietary data in Snowflake SQL dialect, and then evaluate the learned skills on Spider2-Lite. For the BIRD-Critic benchmark, we optimize skills on BIRD-mini-dev Li et al. (2024c), which contains 500 Text-to-SQL examples.

Baselines. We compare DIVSKILL-SQL against representative open-source or most relevant Text-to-SQL systems, including DIN-SQL (Pourreza & Rafiei, 2023a), REFORCE (Deng et al., 2025), and CHASE-SQL (Pourreza et al., 2024). We directly use the open-source implementations of DIN-SQL and REFORCE. We examined two versions of CHASE-SQL: CHASE-SQL (naive), which uses the original three-path workflow, and CHASE-SQL (agentic), our adaptation that replaces the fixed workflow with the same agent architecture and tools used by DIVSKILL-SQL, while retaining schema-link shuffling, high-temperature sampling, and pairwise candidate selection. We compare both with DIVSKILL-SQL, but use the agentic version by default as it appears to be much stronger than the naive version. By default, we use the Opus-4.6Anthropic (2026) model. We also show results with GPT-5.4 OpenAI (2026) and the open-source Qwen3-Coder-30B-A3B Yang et al. (2025a). We share the implementation details in Section D.

Metrics. We report three metrics. *Pass@1* measures the execution accuracy of a single generated candidate. *Pass@8* measures the oracle candidate-set accuracy: a problem is counted as solved if

[1] With $K = 8$, $G \approx 1.7$ for baselines and 2.6 for DIVSKILL-SQL on BIRD-Critic.

Method	SQLite			Snowflake			BigQuery		
	pass@1	pass@8	Sel. acc.	pass@1	pass@8	Sel. acc.	pass@1	pass@8	Sel. acc.
DIN-SQL	40.74	/	/	0.97	/	/	18.54	/	/
REFORCE	55.28	/	57.78	43.47	/	50.72	51.10	/	55.12
CHASE-SQL	62.59	76.30	63.70	51.21	68.60	53.14	59.02	71.71	56.59
DIVSKILL-SQL	62.13	73.33	**64.44**	60.08	72.46	**64.25**	62.07	73.17	**64.88**

Table 1: Spider2-Lite results with Opus 4.6.

Method	SQLite			Snowflake			BigQuery		
	pass@1	pass@8	Sel. acc.	pass@1	pass@8	Sel. acc.	pass@1	pass@8	Sel. acc.
DIN-SQL	37.04	/	/	0	/	/	2.93	/	/
REFORCE	64.44	/	58.52	47.83	/	41.06	48.29	/	51.22
CHASE-SQL	61.57	83.70	66.67	51.69	73.91	57.97	53.90	74.15	59.51
DIVSKILL-SQL	62.87	84.44	**71.85**	52.23	76.33	**61.84**	57.80	77.56	**63.41**

Table 2: Spider2-Lite results with GPT-5.4.

at least one of the eight generated candidates is correct[2]. Finally, *selected accuracy* measures the execution accuracy of the single SQL query returned by the selector, either through LLM judge or majority voting, and is the end-to-end performance of the deployed system. This applies to ensemble-based methods including DIVSKILL-SQL, REFORCE, and CHASE-SQL.

Skill seed pool and choice of K. We observe that the Pass@k curve for various methods nearly saturates beyond K=8 empirically. Therefore, unless explicitly specified, we use K=8 by default in the experiments. We initialize the optimization with K=8 seed skills (see seed construction in Section 3), each representing a distinct agent behavior family. Appendix E presents the full set of seeds and their optimized versions.

4.2 END-TO-END PERFORMANCE AND TRANSFERABILITY

Tables 1, 2 and 6 report end-to-end results on BIRD-Critic and Spider2-Lite, grouped by dialect. Overall, DIVSKILL-SQL achieves the strongest selected accuracy in almost all settings and consistently outperforms the strongest ensemble baseline, CHASE-SQL.

The advantage is most visible on the harder dialects in Table 1. With Opus 4.6, DIVSKILL-SQL improves selected accuracy over CHASE-SQL by +11.11 on Snowflake and +8.29 on BigQuery, where large schemas and dialect-specific syntax make unstable decoding more costly. On SQLite, DIVSKILL-SQL has slightly lower Pass@8 than CHASE-SQL (73.33 vs. 76.30), but still achieves higher selected accuracy (64.44 vs. 63.70). This suggests that raw coverage alone is insufficient: SQLite is easier, so stochastic sampling can already cover many cases, but the additional candidates may include plausible wrong queries that are harder for the selector to distinguish Pourreza et al. (2025). The same pattern holds with GPT-5.4 in Table 2, where DIVSKILL-SQL improves both Pass@8 and selected accuracy across all three dialects. Note that REFORCE selects its final SQL by majority voting over internal candidates. With GPT-5.4, its selected result underperforms mean Pass@1 on SQLite and Snowflake, showing that majority voting can fail when many candidates converge to the same incorrect result. See Appendix F for detailed analysis.

Comparison with the workflow version of CHASE-SQL. We additionally evaluate CHASE-SQL (naive) on Spider2-Lite (SQLite) with GPT-5.4. Its original three-path workflow achieves 47.41% selected accuracy. Expanding this workflow to eight candidates by sampling the divide-and-conquer, query-plan, and synthetic paths 3, 3, and 2 times, respectively, yields 48.15%. With eight candidates, CHASE-SQL (agentic) achieves 66.67% and DIVSKILL-SQL achieves 71.85%.

Method	K	Pass@1	Pass@K	Sel. acc.
CHASE-SQL (naive)	3	40.74	50.37	47.41
CHASE-SQL (naive)	8	48.15	58.52	48.15
CHASE-SQL (agentic)	8	61.57	83.70	66.67
DIVSKILL-SQL	8	**62.87**	**84.44**	**71.85**

Table 5: Comparison with the original CHASE-SQL workflow on Spider2-Lite SQLite using GPT-5.4.

[2]We compute Pass@8 based on internal candidates of ensemble-based methods. As REFORCE is generating a dynamic number of candidates in its workflow, we are not able to compute Pass@8.

Hallucination Type	CHASE-SQL	DIVSKILL-SQL	Ratio
Pools with invalid-reference candidate	10	**7**	1.43x
Solvable pools with invalid-reference candidate	6	**2**	3.00x
Invalid-reference candidate slots	19	**13**	1.46x
Missing-function hallucination cases	6	**2**	3.00x

Table 3: Hallucination diagnostics on Snowflake instances based on invalid-reference failures.

Structural Mismatch vs. Gold SQL	CHASE-SQL	DIVSKILL-SQL	Ratio
Wrong DISTINCT usage	15	**10**	1.50x
Wrong window-function usage	6	**2**	3.00x
Wrong UNION structure	6	**3**	2.00x

Table 4: Structural mismatch analysis on Snowflake instances with reference SQL available.

These results show that the agentic adaptation provides a stronger baseline in this setting (even with the same # paths), while DIVSKILL-SQL further improves selected accuracy by 5.18%.

Hallucination and Error Analysis. Tables 3 and 4 provide a more structured error breakdown, evaluated on the Snowflake part of Spider2-Lite. Compared with CHASE-SQL, DIVSKILL-SQL produces fewer invalid-reference failures: the number of pools containing such a candidate drops from 10 to 7, and among solvable pools, the count drops from 6 to 2. Missing-function hallucinations also decrease from 6 to 2. The structural comparison against gold SQL shows a similar pattern: DIVSKILL-SQL makes fewer errors involving DISTINCT, window functions, and UNION structure. These results indicate that residual skills *improve diversity in a more controlled way*. Rather than relying on high-temperature perturbations that can introduce hallucinated references or unstable SQL structures, DIVSKILL-SQL induces different agent behaviors while preserving candidate quality.

Transfer across SQL dialects and task settings. We next analyze whether skills encode reusable problem-solving strategies or merely overfit to one dialect or task.

Recall that on Spider2-Lite, the skills are optimized only on Snowflake SQL format data, but are applied to all dialects without further optimization. The end-to-end results in Tables 1 and 2 show that DIVSKILL-SQL outperforms all baselines on SQLite and BigQuery by a large margin. This suggests that residual skill optimization does not merely memorize Snowflake-specific syntax. Instead, as we will show later in Section 4.4, the learned skills capture higher-level strategies for schema exploration, decomposition, query construction, and error checking that transfer across dialects.

We also evaluate whether skills optimized on BIRD-mini-dev, a standard Text-to-SQL dataset, transfer to BIRD-Critic's SQL-debugging tasks, which lack the same-task training data. Results in Table 6 confirm that the skills transfer: DIVSKILL-SQL improves selected accuracy by 2.64% over CHASE-SQL, with consistent gains in pass@1 (+1.47) and pass@8 (+2.27). The pass@8 improvement indicates that the learned skills broaden the candidate pool, even on a task that they were not optimized for.

Method	**pass@1**	**pass@8**	**Sel. acc.**
DIN-SQL	30.38	–	–
REFORCE	41.50	–	43.21
CHASE-SQL	44.69	52.26	46.23
DIVSKILL-SQL	46.16	54.53	**48.87**

Table 6: Bird-Critic PostgreSQL Results.

4.3 ABLATION STUDIES

Pass@k comparison. We compare DIVSKILL-SQL with its three variants: *Base*, which repeatedly samples from the original base skill; *Opt. Base*, which runs GEPA to optimize residuals from only base skills; and *Seeds*, which uses the unoptimized initial skill seeds. Figure 2 studies how candidate-set coverage changes as we increase the number of generated candidates. Repeated sampling from the base skill (Base) improves Pass@k only gradually, indicating that independent runs of the same skill tend to make correlated mistakes. Optimizing a single base skill improves individual quality, but still leaves many residual failures uncovered. An initial diverse set of seed skills helps compared to Base in some cases (Figure 2b), but might also degrade performance (Figure 2a). In contrast, DIVSKILL-SQL achieves the strongest Pass@k curve across nearly all values of k. The gain is especially meaningful at small and moderate k, where each additional candidate must cover new failure modes to be useful. As a result, to match DIVSKILL-SQL's pass@8, baselines need 3 to 8 additional passes, making DIVSKILL-SQL a more cost-efficient choice. This behavior is exactly

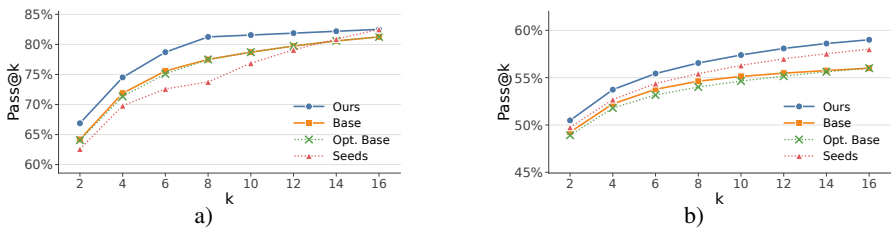

Figure 2: Pass@k comparison between DIVSKILL-SQL and its variants on 100-instance subsets of a) Spider2-lite and b) Bird-Critic.

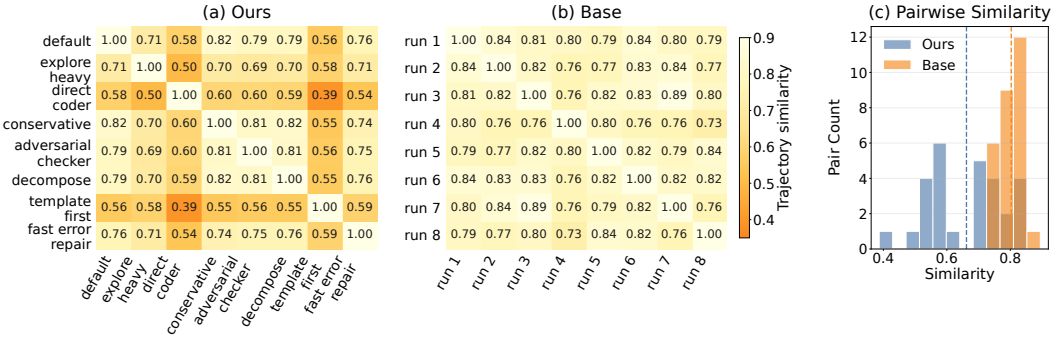

Figure 3: Trajectory comparison between DIVSKILL-SQL and repeated runs of the default skill on Snowflake part of Spider2-Lite.

what residual skill optimization is designed to produce: each later skill is not optimized to be globally better on all questions, but to solve examples that previous skills miss.

Open-source models and data. We additionally evaluate DIVSKILL-SQL and CHASE-SQL with Qwen3-Coder-30B-A3B on BIRD dev, optimizing skills only on the public BIRD training split. As shown in Table 7, DIVSKILL-SQL improves accuracy by 12.44%, 14.68%, and 6.92% on simple, moderate, and challenging examples, respectively. These results show that DIVSKILL-SQL also benefits open-source models trained on public data, which is a cost-efficient option.

Method	Simple	Moderate	Challenging
CHASE-SQL	64.88	50.33	28.57
DIVSKILL-SQL	**77.32**	**65.01**	**35.49**

Table 7: BIRD dev accuracy breakdown with Qwen3-Coder-30B-A3B as backbone.

4.4 HOW DIVERSE SKILLS CHANGE AGENT BEHAVIOR

We analyze agent trajectories, which include sequences of high-level actions such as schema inspection, exploratory SQL generation, and error repair, to examine how skills affect agent behavior. Trajectory similarity is defined as 1 minus edit distance normalized to $[0, 1]$, with lower values indicating more distinct action sequences.

Figure 3 shows that repeated sampling produces highly similar trajectories, with most pairwise similarities ranging from 0.75 to 0.85. In contrast, DIVSKILL-SQL produces a wider similarity range: skills such as direct-coder, template-first, and exploration-heavy form low-similarity pairs, indicating distinct approaches to the task. The same pattern appears on the BigQuery and SQLite subsets of Spider2-Lite (Figures 4 and 5), where skills optimized on Snowflake are applied without retraining. These results suggest that learned skills promote behavioral diversity across SQL dialects, consistent with the stronger candidate coverage shown by the Pass@K curve in Figure 2.

5 CONCLUSIONS

We introduce DIVSKILL-SQL, a residual skill optimization framework for constructing complementary Text-to-SQL ensembles. Rather than relying on hand-designed prompts or workflow variants, or high-temperature sampling, DIVSKILL-SQL learns a bank of skill-conditioned agents, where each skill is optimized to recover examples missed by the others. This directly targets candidate-set coverage and improves Pass@K while preserving the quality of SQL candidates. Across Spider2-Lite and BIRD-Critic, spanning multiple SQL dialects and task formats, DIVSKILL-SQL improves end-to-end accuracy over strong ensemble baselines while producing fewer hallucinated outputs and less redundant agent behavior.

AI USE STATEMENT

In this work, we used generative AI tools to help with paper polishing and method implementation. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, clean and reformat datasets, support qualitative and thematic data analysis, and interpret results. Generating synthetic data sets is not applicable to this work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Anthropic. Introducing Claude Opus 4.6. https://www.anthropic.com/news/claude-opus-4-6, February 2026.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Kaiwen Chen, Yueting Chen, Nick Koudas, and Xiaohui Yu. Reliable text-to-sql with adaptive abstention. *Proc. ACM Manag. Data*, 3(1), February 2025a. doi: 10.1145/3709719. URL https://doi.org/10.1145/3709719.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Zhipeng Chen, Xiaobo Qin, Youbin Wu, Yue Ling, Qinghao Ye, Wayne Xin Zhao, and Guang Shi. Pass@ k training for adaptively balancing exploration and exploitation of large reasoning models. *arXiv preprint arXiv:2508.10751*, 2025b.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Yaxun Dai, Haiqin Yang, Mou Hao, and Pingfu Chao. PARSQL: Enhancing text-to-SQL through SQL parsing and reasoning. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 661–681, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.37. URL https://aclanthology.org/2025.findings-acl.37/.

Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. Reforce: A text-to-sql agent with self-refinement, format restriction, and column exploration. In *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild*, 2025.

Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Jinshu Lin, Dongfang Lou, et al. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306*, 2023.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363*, 2023.

Yu Guo, Dong Jin, Shenghao Ye, Shuangwu Chen, Jian Yang, and Xiaobin Tan. SQLForge: Synthesizing reliable and diverse data to enhance text-to-SQL reasoning in LLMs. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 8441–8452, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.443. URL https://aclanthology.org/2025.findings-acl.443/.

Arian Hosseini, Xingdi Yuan, Nikolay Malkin, Aaron Courville, Alessandro Sordoni, and Rishabh Agarwal. V-star: Training verifiers for self-taught reasoners. *arXiv preprint arXiv:2402.06457*, 2024.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. MCS-SQL: Leveraging multiple prompts and multiple-choice selection for text-to-SQL generation. In Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert (eds.), *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 337–353, Abu Dhabi, UAE, January 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.coling-main.24/.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*, 2024.

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. The dawn of natural language to sql: Are we fully ready? *Proc. VLDB Endow.*, 17(11):3318–3331, July 2024a. ISSN 2150-8097. doi: 10.14778/3681954.3682003. URL https://doi.org/10.14778/3681954.3682003.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. Codes: Towards building open-source language models for text-to-sql. *Proc. ACM Manag. Data*, 2(3), May 2024b. doi: 10.1145/3654930. URL https://doi.org/10.1145/3654930.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024c.

Jinyang Li, Xiaolong Li, Ge Qu, Per Jacobsson, Bowen Qin, Binyuan Hui, Shuzheng Si, Nan Huo, Xiaohan Xu, Yue Zhang, et al. Swe-sql: Illuminating llm pathways to solve user sql issues in real-world applications. *arXiv preprint arXiv:2506.18951*, 2025.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The twelfth international conference on learning representations*, 2023.

Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, et al. Xiyan-sql: A novel multi-generator framework for text-to-sql. *IEEE Transactions on Knowledge and Data Engineering*, 2026.

Peixian MA, Xialie Zhuang, Chengjin Xu, Xuhui Jiang, Ran Chen, and Jian Guo. SQL-r1: Training natural language to SQL reasoning model by reinforcement learning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. URL https://openreview.net/forum?id=hgJQcuDwm1.

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. Skillclaw: Let skills evolve collectively with agentic evolver. *arXiv preprint arXiv:2604.08377*, 2026.

George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical programming*, 14(1):265–294, 1978.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pp. 26106–26128. PMLR, 2023.

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

Silviu Pitis, Michael R Zhang, Andrew Wang, and Jimmy Ba. Boosted prompt ensembles for large language models. *arXiv preprint arXiv:2304.05970*, 2023.

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

Mohammadreza Pourreza and Davood Rafiei. Evaluating cross-domain text-to-sql models and benchmarks. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 1601–1611, 2023b.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943*, 2024.

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

Taylor Shin, Yasaman Razeghi, Robert L Logan IV, Eric Wallace, and Sameer Singh. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*, pp. 4222–4235, 2020.

Chang-Yu Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. Exploring chain of thought style prompting for text-to-sql. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 5376–5393, 2023.

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

Christian Walder and Deep Karkhanis. Pass@ k policy optimization: Solving harder reinforcement learning problems. *arXiv preprint arXiv:2505.15201*, 2025.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, et al. Mac-sql: A multi-agent collaborative framework for text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 540–557, 2025.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026.

Yuanzhen Xie, Xinzhou Jin, Tao Xie, Mingxiong Lin, Liang Chen, Chenyun Yu, Lei Cheng, Chengxiang Zhuo, Bo Hu, and Zang Li. Decomposition for enhancing attention: Improving LLM-based text-to-SQL through workflow paradigm. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 10796–10816, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.641. URL https://aclanthology.org/2024.findings-acl.641/.

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

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2023.

Haolin Yang, Jipeng Zhang, Zhitao He, and Yi R Fung. Mars-sql: A multi-agent reinforcement learning framework for text-to-sql. *arXiv preprint arXiv:2511.01008*, 2025b.

Jiaxi Yang, Binyuan Hui, Min Yang, Jian Yang, Junyang Lin, and Chang Zhou. Synthesizing text-to-SQL data from weak and strong LLMs. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7864–7875, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.425. URL https://aclanthology.org/2024.acl-long.425/.

Jian Yao, Ran Cheng, Xingyu Wu, Jibin Wu, and Kay Chen Tan. Diversity-aware policy optimization for large language model reasoning. *arXiv preprint arXiv:2505.23433*, 2025.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 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*, pp. 3911–3921, 2018.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? *arXiv preprint arXiv:2504.13837*, 2025.

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

Barry Zhang, Keith Lazuka, and Mahesh Murag. Equipping agents for the real world with agent skills, october 2025. *URL https://www. anthropic. com/engineering/equipping-agents-for-the-real-world-with-agent-skills. Accessed*, pp. 01–28, 2026a.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, et al. Evoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026b.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. Memskill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*, 2026c.

Qinggang Zhang, Hao Chen, Junnan Dong, Shengyuan Chen, Feiran Huang, and Xiao Huang. Structure-guided large language models for text-to-SQL generation. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=gT8JSEFqaS.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, et al. Memento-skills: Let agents design agents. *arXiv preprint arXiv:2603.18743*, 2026.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The eleventh international conference on learning representations*, 2022.

A Detailed Literature Review

Pass@K Optimization Pass@K measures whether a model produces at least one correct solution among K samples (Chen et al., 2021), and has recently been adopted as an optimization target. Yue et al. (2025) show that RL with verifiable rewards improves Pass@1 but not Pass@K: training reduces output diversity and leaves hard problems outside the base model's support unsolved. This motivates directly optimizing for diversity, which several works pursue through policy optimization (Chen et al., 2025b; Walder & Karkhanis, 2025; Yao et al., 2025). However, these methods all require parameter updates, limiting them to open-weight models and to improving a single model's output distribution. Our method instead learns K complementary skills without parameter updates, improving Pass@K through ensemble construction for both open and closed models.

Skill and Prompt Optimization Prompts strongly influence LLM behavior (Brown et al., 2020), motivating automatic optimization methods that improve prompts from task feedback rather than manual design. Early work optimizes discrete trigger tokens or textual prompts (Shin et al., 2020), while recent methods use LLMs to propose, evaluate, and revise prompts iteratively (Zhou et al., 2022; Yang et al., 2023; Agrawal et al., 2025). Boosted prompt ensembles (Pitis et al., 2023) also optimize prompt ensembles by adding few-shot prompts that target examples on which the current ensemble is uncertain or incorrect. Beyond short prompts and few-shot demonstrations, agent systems increasingly use *skills*: modular instruction files encoding task strategies, constraints, and tool-use policies (Zhang et al., 2026a). Recent work explores agents that rewrite their skills (Zhou et al., 2026), jointly optimize skills with model parameters (Xia et al., 2026), or evolve skills over time (Ma et al., 2026; Zhang et al., 2026b;c). Our work follows this direction but targets ensemble construction: rather than learning one strong skill, we learn complementary skills that address different failure modes and improve the ensemble's Pass@K.

Text-to-SQL Early LLM-based Text-to-SQL methods prompt a single model to generate SQL directly (Dong et al., 2023; Tai et al., 2023). More recent systems use agent-based pipelines that decompose the task into schema pruning, evidence extraction, SQL generation, and refinement (Pourreza & Rafiei, 2023a; Gao et al., 2023; Wang et al., 2025; Talaei et al., 2024). Several works further improve performance through ensembling (Pourreza et al., 2024; Deng et al., 2025): CHASE-SQL generates candidates from diverse prompts and selects among them via a tournament (Pourreza et al., 2024); MARS-SQL trains a multi-agent system with RL but is restricted to open models and does not explicitly optimize for diversity (Yang et al., 2025b); XiYan-SQL fine-tunes multiple generators for diverse outputs (Liu et al., 2026). In contrast, our method requires no weight modifications or hand-designed ensembles. We optimize skills to explicitly construct complementary generators, increasing Pass@K coverage while remaining applicable to both open and closed LLMs.

B Detailed Discussions and Limitations

DivSkill-SQL primarily improves candidate generation and does not focus on enhanced candidate selection method. Across several settings, there remains a substantial gap between Pass@8 and selected accuracy, indicating that the correct SQL is often present in the candidate pool but not selected. Our pairwise comparison procedure mitigates direct 1-of-K selection difficulty and reduces position bias by swapping candidate order, but it still relies on an LLM judge and scales quadratically in the number of surviving candidates. Future work could combine residual skill optimization with stronger selectors. Recent advances in language model reasoning demonstrate that verifiers trained on self-generated correct and incorrect trajectories—often formulated as outcome or process reward models—can significantly enhance test-time selection among multiple candidates (Cobbe et al., 2021; Lightman et al., 2023; Hosseini et al., 2024; Ni et al., 2023). This paradigm is naturally aligned with our framework: DivSkill-SQL already generates diverse candidate reasoning paths and SQL programs, while execution feedback provides an abundant, automated source of positive and negative supervision to effectively train such selectors without human annotation.

In addition, our evaluation focuses on recent Text-to-SQL and SQL-debugging benchmarks with executable ground truth. Although the learned skills transfer across SQL dialects and even from standard Text-to-SQL training data to BIRD-Critic, broader deployment settings may introduce additional challenges, including ambiguous user intent, missing schema documentation, and multi-

turn interactive feedback-based debugging. Future work might explore how to adapt DIVSKILL-SQL to these more complex settings.

C PROOFS

C.1 PROOF OF PROPOSITION C.1

We analyze the population-level version of residual skill optimization. In this setting, the next skill is chosen to maximize its expected contribution on the residual failure mass of the current skill bank. The finite-batch procedure in Algorithm 1 can be viewed as an empirical approximation to this objective; deriving finite-sample guarantees would require additional assumptions on sample size and generalization of the learned skills.

Proposition C.1 (Residual skill optimization approximates optimal Pass@K)**.** *Let $\mathcal{S}$ be a fixed skill family, and let $p_s(x) \in [0,1]$ denote the probability that one execution of skill s solves input x. For any skill bank $A \subseteq \mathcal{S}$, define its population Pass@K objective as*

$$F(A) = \mathbb{E}_{x\sim P}\left[1 - \prod_{s\in A}(1 - p_s(x))\right].$$

Starting from $A_0 = \emptyset$, suppose that at each round $j = 1, \ldots, K$, residual skill optimization selects

$$s_j \in \arg\max_{s\in\mathcal{S}} \mathbb{E}_{x\sim P}\left[p_s(x) \prod_{s'\in A_{j-1}} (1 - p_{s'}(x))\right],$$

where $A_{j-1} = \{s_1, \ldots, s_{j-1}\}$, and sets $A_j = A_{j-1} \cup \{s_j\}$. Let

$$A^\star \in \arg\max_{A\subseteq\mathcal{S},\ |A|\le K} F(A)$$

be the optimal size-K skill bank. Then

$$F(A_K) \ge (1 - 1/e)F(A^\star).$$

Equivalently,

$$\text{Pass@K}(\{s_1, \ldots, s_K\}) \ge (1 - 1/e) \max_{|A|\le K} \text{Pass@K}(A).$$

Proof. The proof follows the standard greedy analysis for monotone submodular maximization under a cardinality constraint (Nemhauser et al., 1978); we first verify that the Pass@K objective in our setting is indeed monotone submodular.

We first show that the Pass@K objective is monotone submodular. For a skill bank $A \subseteq \mathcal{S}$, recall that

$$F(A) = \mathbb{E}_{x\sim P}\left[1 - \prod_{s\in A}(1 - p_s(x))\right].$$

For any skill $s \notin A$, the marginal gain of adding s is

$$\Delta(s \mid A) = F(A \cup \{s\}) - F(A).$$

Expanding the definition of F, we obtain

$$\begin{aligned}\Delta(s \mid A) &= \mathbb{E}_{x\sim P}\left[1 - (1 - p_s(x)) \prod_{s'\in A}(1 - p_{s'}(x))\right] - \mathbb{E}_{x\sim P}\left[1 - \prod_{s'\in A}(1 - p_{s'}(x))\right] \\ &= \mathbb{E}_{x\sim P}\left[p_s(x) \prod_{s'\in A}(1 - p_{s'}(x))\right].\end{aligned}$$

Since $p_s(x) \in [0,1]$, this marginal gain is nonnegative. Hence F is monotone.

Next, let $A \subseteq B \subseteq \mathcal{S}$. Since each factor $1 - p_{s'}(x) \in [0, 1]$, we have

$$\prod_{s' \in B} (1 - p_{s'}(x)) \leq \prod_{s' \in A} (1 - p_{s'}(x)).$$

Therefore,

$$\Delta(s \mid B) = \mathbb{E}_{x \sim P} \left[p_s(x) \prod_{s' \in B} (1 - p_{s'}(x)) \right] \leq \mathbb{E}_{x \sim P} \left[p_s(x) \prod_{s' \in A} (1 - p_{s'}(x)) \right] = \Delta(s \mid A).$$

Thus F satisfies diminishing marginal returns and is submodular.

The residual arg-max rule is exactly greedy maximization of this objective, because the marginal gain of adding skill s to the current bank A_{j-1} is

$$\Delta(s \mid A_{j-1}) = \mathbb{E}_{x \sim P} \left[p_s(x) \prod_{s' \in A_{j-1}} (1 - p_{s'}(x)) \right].$$

Thus choosing s_j by residual maximization is the same as choosing the skill with largest marginal increase in F.

Let $A^\star \in \arg\max_{|A| \leq K} F(A)$ be an optimal size-K skill bank. Since F is monotone,

$$F(A^\star) - F(A_j) \leq F(A_j \cup A^\star) - F(A_j).$$

By submodularity,

$$F(A_j \cup A^\star) - F(A_j) \leq \sum_{s \in A^\star} \Delta(s \mid A_j).$$

Since $|A^\star| \leq K$, at least one skill $s \in A^\star$ has marginal gain at least

$$\frac{F(A^\star) - F(A_j)}{K}.$$

Greedy chooses the skill with largest marginal gain, so

$$F(A_{j+1}) - F(A_j) \geq \frac{F(A^\star) - F(A_j)}{K}.$$

Equivalently,

$$F(A^\star) - F(A_{j+1}) \leq \left(1 - \frac{1}{K}\right) \left(F(A^\star) - F(A_j)\right).$$

Applying this recurrence for K rounds gives

$$F(A^\star) - F(A_K) \leq \left(1 - \frac{1}{K}\right)^K F(A^\star).$$

Hence

$$F(A_K) \geq \left(1 - \left(1 - \frac{1}{K}\right)^K\right) F(A^\star) \geq (1 - 1/e) F(A^\star).$$

This proves the claim. □

D Implementation Details and Best Empirical Practices

D.1 Implementation Details

Machine setup. All experiments were run on a single Apple Silicon MacBook Pro (Apple M-series CPU, 10+ cores; 32 GB unified memory; macOS 14). Since our pipeline issues all language-model calls to hosted inference APIs, no local GPU is required: the workstation only orchestrates prompting, parsing, and result aggregation.

Hyperparameter	Value
Agent (evaluation)	
Max agent turns	12
Max SQL executions	20
Max completion tokens	64000
GEPA optimizer	
Reflection max tokens	64000
Metric calls per skill per batch	20
Reflection minibatch size	3
Training loop	
Number of batches	3
Batch size (train / val)	70 / 30
Held-out validation size	80
Max prompt length	12000 chars

Table 8: GEPA hyperparameters for BIRD-mini-dev skill optimization. Spider2-Lite Snowflake training uses the same settings except the training data source.

LLM backbone and inference settings. We evaluate DIVSKILL-SQL with two LLM backbones: Opus-4.6 and GPT-5.4. Both are used in non-reasoning mode with a decoding temperature of 0.2, a canonical choice for coding tasks, and a maximum completion budget of 64000 tokens. The low temperature is a deliberate design choice: since DIVSKILL-SQL achieves diversity through learned skills rather than stochastic decoding, a low temperature preserves the reasoning stability and precision needed for complex SQL generation. Each agent run is allowed up to 12 reasoning turns and 20 SQL executions.

CHASE-SQL adaptation. CHASE-SQL was originally designed for a workflow-based Text-to-SQL pipeline. To produce a fair comparison in our agentic setting, we adapt its three transferable design choices to our agent architecture:

- *Schema-link shuffling.* We permute the column ordering presented to the agent across different candidates, matching CHASE-SQL's schema perturbation strategy.
- *High-temperature decoding.* We set the agent's decoding temperature to 1.0 for CHASE-SQL candidates, reproducing the stochastic variation that CHASE-SQL uses to induce candidate diversity.
- *Pairwise candidate selection.* We use the same LLM-based pairwise comparison selector for both CHASE-SQL and DIVSKILL-SQL, ensuring the selection mechanism is held constant across methods.

Candidate selection. For both DIVSKILL-SQL and CHASE-SQL, we use an LLM-based pairwise selector that compares each candidate pair and selects the winner via win-rate-based aggregation. The selector uses the same backbone model as the candidate generator (either Opus-4.6 or GPT-5.4) at a temperature of 0.2. Each pairwise comparison receives the question, database schema, and the two candidate SQL queries along with execution previews, and returns a preference judgment.

GEPA optimization settings. Table 8 summarizes the GEPA hyperparameters used for skill optimization.

Agent tools. The agent has access to six tools during both training and evaluation:

- `execute_sql`: run SQL against the live database and return results or error messages.
- `lookup_docs`: retrieve dialect-specific documentation (e.g., dialect function/grammar reference), database meta-data, external knowledge, etc.
- `review_sql`: invoke an LLM-based critic to review the current SQL draft before submission.
- `get_sql_pattern`: retrieve anonymized SQL patterns for similar query types (e.g., top-N, running totals).
- `get_sql_templates`: retrieve SQL templates that are masked training instances categorized by query types.
- `submit_final_sql`: submit the final answer.

The tool set and their implementations remain fixed across all skills and all experiments. Skills influence only the strategy text in the system prompt; they cannot add, remove, or modify tools.

D.2 Practices for Better Skill Learning

We share practical small tricks we applied for skill optimization (in particular GEPA).

Brevity as a tiebreaker. During GEPA optimization, multiple candidate skill mutations often achieve the same accuracy gain on the hard batch. Rather than selecting arbitrarily, we break ties by preferring the shorter prompt. This acts as a lightweight regularizer: longer prompts tend to accumulate instance-specific details from the training batch, increasing the risk of overfitting to particular schemas or question patterns.

Dialect- and instance-agnostic reflection. We explicitly instruct the reflector model to avoid including any instance-specific or dialect-specific details in the optimized strategy text, including concrete table names, column names, dialect-specific function syntax, or schema patterns observed in the training batch. The reflector is told that the skill must generalize across unseen databases, SQL dialects, and task formats. This constraint is important because the skills optimized on training data with Snowflake grammar are later evaluated on SQLite and BigQuery, making dialect-specific advice actively harmful.

Skill-order rotation. In Algorithm 1, each skill removes its solved examples before the next skill is optimized, so earlier skills in a batch see an easier and broader residual, while later skills are trained on a narrower and harder subset. To guarantee that each skill occasionally sees the full batch before other skills remove solved examples, we rotate the skill order across batches. This reduces positional bias and prevents later skills from being systematically specialized only to the hardest tail of failures. Note that if the number of batches is fewer than the number of skills, we apply a larger stride to the rotation to maintain uniform positional coverage.

E SKILL POOL

We list the $K = 8$ seed skills used in our experiments, showing both the initial hand-designed seed prompt and the prompt after residual optimization on Snowflake training data. Each prompt is injected into the agent's system message; the agent's tool set and control flow remain unchanged. Optimized prompts are labeled with a round suffix (e.g., `_r1`) indicating the GEPA batch that produced the accepted mutation.

default. *Strategy:* the baseline agentic behavior—balanced exploration followed by incremental SQL construction.

Seed prompt: default

```
## Strategy
1. EXPLORE first: run queries to understand the data--check table
structures, column values, data types, join keys, actual string values
in the data
2. If unsure about any SQL function's syntax or behavior, call
lookup_docs BEFORE writing the query
3. For common patterns (top-N, running totals, pivots), call
get_sql_pattern for a template
4. PLAN your approach based on what you discovered
5. WRITE and TEST your SQL incrementally--run it via execute_sql to
check results
6. VERIFY results look reasonable (right number of rows, right
columns, sensible values)
7. Call review_sql to get a second opinion before submitting
8. SUBMIT only when confident
```

Optimized prompt: default

```
1. EXPLORE the data first (as in seed).
2. CLARIFY ambiguities--identify potential traps: NULLs in key
columns, case sensitivity, duplicate rows, date formats, and whether
counts should be DISTINCT.
3. MAP the question to SQL primitives--explicitly decide join type
(INNER vs LEFT), filter placement (WHERE vs HAVING), aggregation scope,
and NULL handling before coding.
4. Check templates--call get_sql_pattern and lookup_docs (as in
seed).
5. BUILD incrementally--write and execute_sql each CTE or subquery
alone (as in seed).
6. VALIDATE against the question--re-read the question, then check:
correct columns returned? correct filter conditions? DISTINCT where
needed? NULL-safe denominators? ordering and limits match?
7. CROSS-CHECK edge cases--run a quick sanity query (e.g., total
counts, min/max values, a spot-check join) to confirm the final result
is not inflated by fanout or deflated by over-filtering.
8. REVIEW--call review_sql and address any flagged issues.
9. SUBMIT only after incremental checks and review pass.
```

direct_coder. *Strategy:* drafts SQL immediately, refines through execution feedback.

Seed prompt: direct_coder

```
## Strategy: DIRECT CODING
You are an EFFICIENT SQL writer. Write SQL quickly, test, iterate.

1. Read the question carefully. Identify the core tables, joins, and
aggregations needed.
2. Write your best SQL attempt IMMEDIATELY based on the schema.
3. Execute it. If errors occur, read the error message carefully and
fix.
4. If the query runs but results look wrong, investigate specific
columns/values.
5. Iterate rapidly--each revision should fix one specific issue.
6. Do NOT over-explore. Only investigate columns/values that are
directly relevant to errors.
7. SUBMIT as soon as the query produces reasonable results.
```

GEPA added lookup-table awareness and structured error-repair guidance. Key additions over the seed (new or substantially expanded material in **bold**):

Optimized prompt: direct_coder

```
## Strategy: DIRECT CODING

1. Read the schema first. Before writing any SQL, identify ALL
tables mentioned or implied by the question. Pay special attention
to lookup/reference/static tables (e.g., category tables, node
tables, type tables) that provide human-readable names or filter
criteria--these almost always require a JOIN.
2. Map question terms to schema columns. If the question references
a name, label, or category, find which table owns that column.
Never filter or select on a column that doesn't exist in the target
table--use the correct table via JOIN instead.
3. Write your best SQL IMMEDIATELY based on the schema. Use explicit
JOIN conditions. When lookup tables exist, join them rather than
filtering on raw IDs or payload strings.
4. Execute. Fix errors specifically:
- Column-not-found → find the correct table and JOIN it
- Wrong results → verify JOIN keys and WHERE filters match actual
values
- Missing rows → check JOIN type (INNER vs LEFT)
5. Aggregation and output checks: Verify GROUP BY includes all
non-aggregated SELECT columns; verify ORDER BY, LIMIT, and NULLS LAST
where appropriate; confirm output column names match the question.
6. Iterate rapidly--each revision fixes one specific issue.
7. SUBMIT as soon as the query produces reasonable results.

Key reminder: Questions involving categories, types, nodes, or
classifications almost always require joining to a static/lookup
table. Never assume the needed label lives in the fact table--check
the schema.
```

decompose. *Strategy:* identifies sub-questions before composing the final SQL.

Seed prompt: decompose

```
## Strategy: DECOMPOSE & CONQUER
Break complex questions into simple subqueries, build bottom-up.

1. PARSE the question into atomic requirements: What is being
counted/summed/averaged? What are the filter conditions? What are
the grouping columns? Is there ranking, ordering, or limiting?
2. BUILD each piece as a standalone CTE: Start with the base data,
add joins one at a time verifying row counts, add aggregations
verifying results at each step.
3. COMPOSE CTEs into the final query using WITH...SELECT.
4. Run each CTE individually via execute_sql to verify intermediate
results.
5. Call review_sql on the assembled final query before submitting.
6. SUBMIT only when confident.
```

GEPA expanded this into a structured six-step process with explicit grain-anchoring and filter-validation phases. Key additions (in **bold**):

Optimized prompt: decompose

```
## Strategy: DECOMPOSE & CONQUER

Step 1: PARSE the question into atomic requirements--explicitly
identify: output columns/metrics, grain (one row per what?), filter
conditions, grouping dimensions, and whether there is ranking,
ordering, limiting, or a ratio/composition calculation.
Step 2: ANCHOR the grain before grouping. Match GROUP BY columns
precisely to the output grain--no more, no less. If the question asks
for monthly totals, group by month only; do not add route, city, or
other columns unless explicitly requested.
Step 3: BUILD each piece as a standalone CTE. For ratio/composition
queries, compute totals in one CTE, subgroup counts in another, then
JOIN and divide.
Step 4: VERIFY join logic and filter semantics. Confirm join keys
actually link the intended entities; avoid fan-out. Confirm filter
values match the domain as they appear in the data, not as paraphrased
in the question.
Step 5: ASSEMBLE and REVIEW. Call review_sql; confirm output columns
and grain match the question.
Step 6: SUBMIT only when confident.
```

explore_heavy. *Strategy:* spends extra steps on schema and sample-row inspection before drafting.

Seed prompt: explore_heavy

```
## Strategy: DEEP EXPLORATION
You are a THOROUGH explorer. Before writing ANY SQL, deeply
understand the data.

1. CATALOG SCAN: List all schemas and tables. Check which tables
have data (SELECT COUNT(*)).
2. COLUMN AUDIT: For each relevant table, run DESCRIBE or SHOW
COLUMNS. Check actual column types.
3. VALUE PROFILING: For key columns, run SELECT DISTINCT to see
actual values, formats, ranges.
4. JOIN DISCOVERY: Test joins between tables with small queries
before using them in final SQL.
5. Only after thorough exploration, write your query.
6. Test edge cases: What if there are NULLs? What if the join
produces duplicates?
7. Call review_sql before submitting.
8. SUBMIT only when confident.
```

GEPA reorganized the strategy into six explicit phases and added an output-requirements clarification phase (Phase 4) that was absent from the seed. Key additions (in **bold**):

Optimized prompt: explore_heavy

```
## Strategy: DEEP EXPLORATION
Phase 1-3: Schema Discovery, Value Profiling, Join Validation (as in
seed).
Phase 4: Clarify Output Requirements Before Writing SQL.
Before coding, explicitly answer:
- Granularity: one summary row, or one row per dimension?
- Aggregation: which columns require AVG, SUM, COUNT?
- Filters: date ranges, status filters, or categorical constraints
implied?
- Extra columns: would adding dimension columns change the
granularity?
If the question asks for summary metrics without specifying a
grouping dimension, produce a single aggregated row--do NOT return
raw row-level data or add unrequested dimension columns.
Phase 5: Write and Validate SQL--apply NULLIF in denominators to
avoid divide-by-zero; use CTEs to separate filtering, joining, and
aggregation stages.
Phase 6: Submit only when the output granularity, aggregation, and
columns exactly match what was requested.
```

conservative. *Strategy:* prefers the simplest faithful query; avoids speculative constructs.

Seed prompt: conservative

```
## Strategy: CONSERVATIVE & SAFE
Prefer simple, safe SQL. Avoid unnecessary complexity.
1. Start with the SIMPLEST possible query that could answer the
question.
2. Avoid: Complex nested subqueries, window functions unless
required, multiple joins when one will do, HAVING when WHERE suffices,
correlated subqueries.
3. Do NOT add any clause the question didn't ask for: No ROUND
unless asked, no COALESCE unless NULLs are a proven problem, no extra
WHERE filters or output columns.
4. When ambiguous, pick the LITERAL interpretation.
5. Test your query. If it works and looks reasonable, submit.
6. SUBMIT once results make sense. Don't over-iterate.
```

GEPA added explicit guidance on AND-vs-OR filter logic and output granularity matching—two recurring failure modes in the training data. Key additions (in **bold**):

Optimized prompt: conservative

```
(Steps 1-3 retained from seed.)
4. Filter logic--AND vs OR:
- AND: both conditions must hold simultaneously on the same row.
- OR: either condition suffices.
- When filtering across related entities (e.g., origin OR destination),
default to OR unless the question explicitly requires all conditions
on the same record.
5. Output structure--match the question's requested granularity
exactly:
- Identify the grouping dimensions the question asks for before writing
GROUP BY.
- Do not substitute finer-grained groupings (e.g., day) when coarser
ones (e.g., month) are requested.
- Column names and aliases should reflect the question's terminology.
6. When ambiguous, pick the LITERAL interpretation. Consult
reference templates for grouping, join, and conditional aggregation
patterns.
7. Test and submit once results make sense.
```

adversarial_checker. *Strategy:* actively stress-tests the query for edge cases before finalizing.

Seed prompt: adversarial_checker

```
## Strategy: ADVERSARIAL SELF-CHECK
After writing SQL, actively try to BREAK it before submitting.

1. Explore and write your initial SQL query.
2. Execute it and get results.
3. NOW, CHALLENGE your own query: Does the question ask for X
but you computed Y? Could a join be producing duplicates? Are you
filtering correctly? For ratios: verify numerator and denominator
independently. For 'top N': verify ordering.
4. Fix any issues you discover.
5. Call review_sql for an independent check.
6. SUBMIT only after surviving both your own and the reviewer's
scrutiny.
```

GEPA restructured the strategy into four explicit phases, adding an upfront decomposition step and specific guidance for period-over-period comparisons and join-type selection:

Optimized prompt: adversarial_checker

Phase 1--Decompose the Question Before Writing:
Identify ALL computations required. If the question involves change/comparison, plan for two separate aggregations and a join or pivot--never a single flat filter. If it involves rates or ratios, plan numerator and denominator explicitly. Sketch the output shape.
Phase 2--Write SQL Matching Full Complexity:
Use CTEs for multi-step logic. Use window functions for rankings.
Use FULL OUTER JOIN when comparing two periods where either side may have no data. Avoid the trap of "just filter and return raw rows" when computation is required.
Phase 3--Adversarial Challenge (expanded from seed): complexity check, output check, aggregation check, filter check, **join type check (OUTER vs INNER)**, ratio/rate check.
Phase 4--Fix and Validate. **Consult templates for period-over-period CTE patterns.**

template_first. *Strategy:* anchors query shape from retrieved SQL patterns before coding.

Seed prompt: template_first

```
## Strategy: TEMPLATE-FIRST PLANNER
Before writing SQL, call get_sql_templates and get_sql_pattern to
anchor the query shape. Pick the closest template, adapt only the
table names, columns, filters, grain, and ordering confirmed from
the schema, then execute one focused validation query. Prefer
template-guided CTE/window/ratio patterns over ad hoc exploration.
Submit after the result shape matches the question.
```

GEPA appended a mandatory pre-submit checklist targeting schema hallucination—a failure mode where the agent references tables or columns it has not verified. Key addition (in **bold**):

Optimized prompt: template_first

(Seed text retained in full.)

MANDATORY pre-submit checklist (do NOT skip):
1. Tables exist as named--every table in your final SQL was confirmed to exist via execute_sql.
2. Columns exist as named--every column appeared in actual schema output.
3. Identifier quoting is correct for the dialect.
4. Final query has executed at least once and returned non-empty rows whose shape matches the question.
Submitting SQL that references non-existent columns, or that has never been successfully executed, is an automatic failure.

fast_error_repair. *Strategy:* prioritizes execution-error recovery over upfront planning.

Seed prompt: fast_error_repair

```
## Strategy: FAST ERROR REPAIR
Move quickly: inspect only the most relevant tables, write the
first plausible SQL early, execute it, and repair from concrete
errors or wrong-shaped results. Do not exhaustively profile. Each
iteration changes one thing: missing column, join key, filter value,
aggregation grain, or dialect syntax. Submit as soon as the SQL runs
and the output shape answers the question.
```

GEPA expanded the single-paragraph seed into a five-phase process, adding a structured triage table and a value-sanity-check phase. Key additions (in **bold**):

Optimized prompt: fast_error_repair

Core loop: write → run → verify values → repair → submit.

Phase 1--Minimal Table Scan (as in seed).
Phase 2--First SQL Draft. **Prefer CTEs for multi-step logic. Commit one join key and one filter assumption up front.**
Phase 3--Execute & Triage. **Change exactly one thing per iteration:**
Column not found → alias or rename
Join returns 0 rows → relax join key or flip direction
Count inflated → add DISTINCT or check fan-out join
Wrong grain → re-examine GROUP BY columns
Phase 4--Value Sanity Check (the key addition):
Before submitting: do not only check shape--check values. Are counts suspiciously 0 or implausibly large? Does a percentage exceed 1.0 or go negative?
Phase 5--Submit when both shape and values are plausible.

F ADDITIONAL EXPERIMENTAL RESULTS

F.1 VARIANCE AND CANDIDATE-LEVEL BREAKDOWNS

Table 9 reports the full Spider2-Lite Opus-4.6 table with standard deviations. The qualitative conclusion from the main text is unchanged after adding variance: DIVSKILL-SQL remains the strongest method on selected accuracy in all three dialects, with the largest gains on the more complex Snowflake and BigQuery settings.

	SQLite			Snowflake			BigQuery		
Method	pass@1	pass@8	Sel. acc.	pass@1	pass@8	Sel. acc.	pass@1	pass@8	Sel. acc.
DIN-SQL	40.74±2.72	/	/	0.97±0.40			18.54±1.71		
REFORCE	55.28±2.62	/	57.78±2.72	43.47±2.09	/	50.72±2.25	51.10±2.23	/	55.12±2.28
CHASE-SQL	62.59±2.47	76.30±2.59	63.70±2.59	51.21±1.94	68.60±2.01	53.14±2.33	59.02±2.00	71.71±2.03	56.59±2.28
DIVSKILL-SQL	62.13±2.04	73.33±1.82	64.44±1.62	60.08±1.59	72.46±1.55	64.25±1.52	62.07±1.84	73.17±1.67	64.88±1.12

Table 9: Spider2-Lite results with Opus 4.6.

Why REFORCE selection can underperform mean candidate quality. In the main experiment, we noted that ReFoRCE's final selected SQL can be worse than the average quality of its individual candidate runs. Table 10 makes this concrete. On SQLite, the mean per-candidate Pass@1 is 64.44 while the final selected accuracy is only 58.52; on Snowflake, the same gap is 47.83 vs. 41.06. This indicates that REFORCE often produces correlated candidates that agree on the same wrong answer, so majority voting can amplify a dominant failure mode rather than recover the strongest candidate. BigQuery is less pathological: selected accuracy (51.22) slightly exceeds the mean candidate accuracy (48.19), but still remains well below the oracle upper bound of 56.59. Here, "Max" is exactly Pass@4 because REFORCE exposes four candidate runs in this evaluation. ReFoRCE's `--num_votes` parameter only sets the number of self-refinement threads per instance, not the number of surviving candidates: each thread is a 5-step refine loop that may terminate without writing any SQL (max-iter exhaustion, empty-result early-stop, invalid response, or oversized schema). In our GPT-5.4 run the realized candidate count per instance ranges over all threads, so a Pass@k oracle is ill-defined and not directly comparable to the Pass@8 results we report for the other methods. Pass@1 is defined as a complete set of first generated candidates for each question to show the interior intermediate results of ReFoRCE before majority voting. In this table, we simply give four sets of candidates to explain why ReFoRCE gets lower accuracy on Sel. acc. compared to so-called Pass@1.

Dialect	cand 0	cand 1	cand 2	cand 3	Min	Max	Mean	Sel. acc.
SQLite (n=135)	63.70	61.48	65.93	65.93	53.33	74.81	64.44	58.52
Snowflake (n=207)	41.55	45.89	43.96	44.44	38.65	56.52	47.83	41.06
BigQuery (n=205)	48.29	46.83	45.37	46.34	41.95	56.59	48.19	51.22

Table 10: REFORCE per-candidate Pass@1 on Spider2-Lite for GPT-5.4

F.2 TRANSFERRED TRAJECTORY DIVERSITY

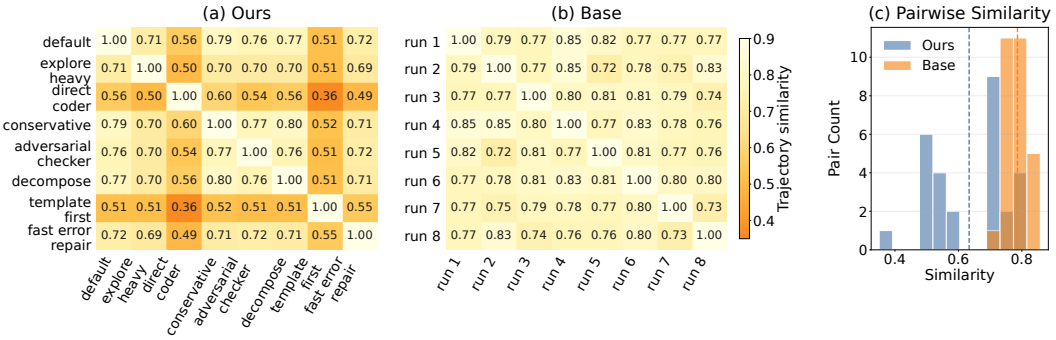

Figure 4: Trajectory comparison between DIVSKILL-SQL and base system's repeated runs. Trained on Snowflake SQL data, and **transferred to BigQuery part of Spider2-Lite**.

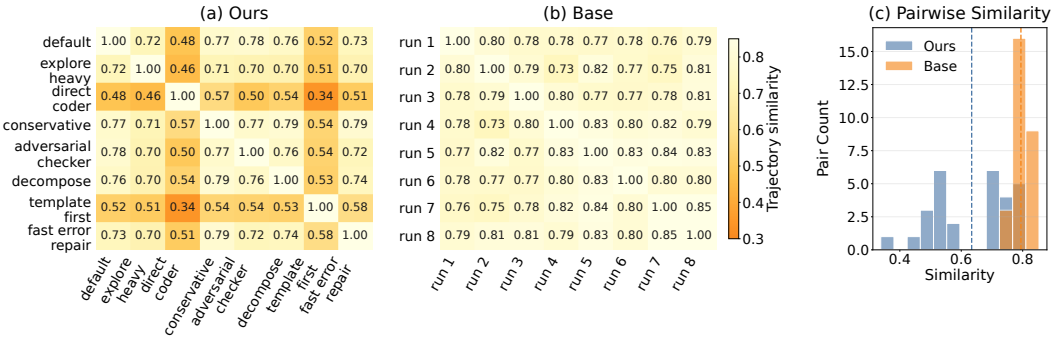

Figure 5: Trajectory comparison between DIVSKILL-SQL and base system's repeated runs. Trained on Snowflake SQL data, and **transferred to SQLite part of Spider2-Lite**.

Figures 4 and 5 extend the trajectory analysis from Snowflake in the main experiment to BigQuery and SQLite. The same qualitative pattern persists after transfer: repeated runs of the base agent remain concentrated in a high-similarity cluster, whereas the skill-conditioned runs spread across a broader range of pairwise similarities. This indicates that the learned skills transfer as behavioral strategies rather than as dialect-specific prompt hacks.

F.3 ADDITIONAL CHASE-SQL COMPARISON WITH GPT-5.4

Candidate-pool quality and rankability. The main Spider2-Lite results suggest that DIVSKILL-SQL's advantage over CHASE-SQL is not only that it contains slightly more correct candidates, but also that its candidate pool is easier to rank. Tables 11 to 13 provide supporting evidence on the GPT-5.4 Spider2-Lite evaluation. Compared with CHASE-SQL, DIVSKILL-SQL improves pass@1 (56.95 vs. 54.96), pass@8 (78.79 vs. 76.42), and selected accuracy (64.90 vs. 60.69), while reducing the oracle–selector gap (13.89 vs. 15.72).

The same-instance head-to-head analysis shows 47 DIVSKILL-SQL-only wins versus 24 CHASE-SQL-only wins on final selection, and McNemar's test confirms that the selected-accuracy gain is statistically significant ($p = 0.0086$). The candidate-density view further shows that DIVSKILL-SQL yields fewer dead pools with zero correct candidates, slightly more rich pools with 6–8 correct candidates, and fewer exact duplicate slots. Together, these results support the claim that residual skill optimization improves both candidate quality and candidate rankability, rather than merely increasing surface-form variation.

Method	0 corr.	1 corr.	2–3 corr.	4–5 corr.	6–8 corr.
CHASE-SQL	23.58	6.76	10.97	11.33	47.35
DIVSKILL-SQL	**21.21**	6.95	10.60	12.43	**48.81**

Table 11: Distribution of the number of correct candidates per instance in the 8-candidate pool. DIVSKILL-SQL pool yields fewer dead pools (0 correct), slightly more rich pools (6–8 correct).

Dialect	Ours-only sel.	CHASE-only sel.	Ours-only oracle	CHASE-only oracle
Local	12	5	6	5
Snowflake	16	8	11	6
BigQuery	19	11	12	5
Total	47	24	29	16

Table 12: Same-instance head-to-head on Spider2-Lite GPT-5.4. "Ours-only sel." counts instances where the final selected SQL is correct for DIVSKILL-SQL but not for CHASE-SQL. "Ours-only oracle" counts instances where the 8-candidate pool contains at least one correct SQL for DIVSKILL-SQL but not for CHASE-SQL.

Comparison	N	DIVSKILL-SQL	CHASE-SQL	Ours-only wins	CHASE-only wins	Exact p
Final selected accuracy (all instances)	547	**64.90**	60.69	47	24	**0.0086**
Oracle pass@8 coverage (all instances)	547	**78.79**	76.42	29	16	0.0725
Final accuracy on jointly solvable instances	402	**86.32**	81.34	39	19	**0.0119**

Table 13: Paired same-instance comparisons on Spider2-Lite GPT-5.4 using the exact McNemar test. The strongest effect is on final selected accuracy, including the subset where both methods already contain at least one correct candidate somewhere in the pool. This supports the claim that the low-temperature DIVSKILL-SQL pool is easier to rank, not merely that it has slightly better oracle coverage.