

VeriPIE: Checkable Query Plans and Evidence-Guided Selection for Text-to-SQL

Piyush Jha*
piyush@knowidea.dev
KNOWIDEA Technology
Atlanta, GA, USA

Zhengyu Li
brian@knowidea.dev
KNOWIDEA Technology
Atlanta, GA, USA

Cruise Song*
csong@gatech.edu
Georgia Tech
Atlanta, GA, USA

Yatharth Sejpal
yatharth@knowidea.dev
KNOWIDEA Technology
Toronto, ON, Canada

ABSTRACT

Financial Artificial Intelligence (AI) governance and model risk management (MRM) emphasize independent validation, transparency, and effective challenge of AI-generated computations. Text-to-SQL translates natural-language questions into Structured Query Language (SQL), but queries that agree on one database can still express different computations. We present VeriPIE, a neurosymbolic framework combining language-model generation with symbolic checks. It ensembles direct SQL generation with QuerySketch, a structured plan language checked by a backend implemented in Lean, a theorem prover and programming language. Plan diagnostics guide repair before SQL generation, while execution agreement and bounded searches for distinguishing databases inform candidate selection. These checks expose malformed plans and hidden disagreement without proving agreement with the user’s intent. In single-run experiments on the complete Spider development set, VeriPIE achieves 72.05% test-suite accuracy (745/1,034), compared with DeepEye’s 65.67% (679/1,034). Its two-query shortlist covers 78.43% (811 questions), compared with 74.66% for DIN-SQL (772) and 69.05% for DeepEye (714). On BIRD, VeriPIE reaches 68.90% execution accuracy and 73.73% two-query coverage. The central benefit is reviewable evidence about the generated computation: explicit plan checks, repair diagnostics, and concrete disagreements that support safer financial analysis.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; • **Applied computing** → *Economics*.

KEYWORDS

text-to-SQL, neurosymbolic AI, query plans, bounded equivalence, diagnostic repair, financial AI safety

1 INTRODUCTION

Financial institutions operate under model risk management (MRM) frameworks emphasizing validation, governance, and effective challenge. For example, the Financial Stability Board ¹ identifies AI-related model risk, data quality, and governance as financial stability

vulnerabilities, while the EU AI Act ² introduces risk-based requirements for covered AI systems. These developments highlight the need for checkable components in financial AI applications. This paper examines one such component: text-to-SQL.

Text-to-SQL enables natural-language access to financial databases through executable queries generated by large language models (LLMs). Such queries may support portfolio risk analysis, regulatory reporting, and transaction reconciliation. However, an incorrect query can silently omit exposures, double-count transactions, or miscalculate financial aggregates while returning plausible results. Traditional execution checks cannot reliably detect these errors, and agreement among generated queries does not establish computational correctness. This creates a distinct validation challenge for financial AI: establishing what a generated query computes and producing evidence that independent validators can examine and challenge.

Execution agreement means that queries return the same results on the observed database. Consider “Find accounts with no matching payment,” with non-null account identifiers. A non-membership test (NOT IN) and a test for no matching row (NOT EXISTS) may agree when payment account identifiers contain no nulls, which represent missing or unknown values. If the payment account-identifier column permits nulls, adding one can cause NOT IN to exclude unmatched accounts because its condition becomes unknown, whereas NOT EXISTS still returns them. Agreement on current data therefore does not establish agreement on other valid databases.

VeriPIE addresses this gap by checking query plans, which describe database operations step by step, and making candidate agreement explicit. Building on DeepEye-SQL [8], it supports direct SQL generation and a plan language called QuerySketch. The latter uses Lean, a theorem prover and programming language, to check plans against the database schema: its tables, columns, and types. Failed checks return diagnostics that guide repair. Selection can combine execution agreement with structural checks and bounded searches, which look within a finite model for databases on which candidates disagree. These checks narrow uncertainty about the generated computation, but do not prove that it matches the user’s intent.

The central idea is to connect candidate generation, plan checking, repair, and query selection through explicit evidence about

*Both authors contributed equally to this research.

¹<https://www.fsb.org/2024/11/the-financial-stability-implications-of-artificial-intelligence/>

²<https://eur-lex.europa.eu/eli/reg/2024/1689/oj>

each candidate. Figure 1 presents this workflow. Section 3 describes how QuerySketch makes intermediate operations inspectable and how equivalence-aware grouping informs the final selection. Section 5 evaluates the resulting framework on the complete Spider and BIRD development sets. Together, these contributions address the safety of generated financial queries by making their operations and the evidence behind selection open to review.

2 RELATED WORK

Recent text-to-SQL systems divide inference into schema linking (matching question phrases to tables and columns), generation, revision, and selection. DeepEye-SQL follows this design, combining multiple generators with deterministic checks and candidate selection [8]. VeriPIE extends generation with complementary strategies and a structured intermediate representation (IR) between natural language and SQL. It augments candidate selection with execution grouping, bounded verification, and ranking subject to additional checks.

IRs reduce the gap between natural-language questions and executable SQL. SemQL and NatSQL simplify generation by abstracting selected SQL details [4, 5], whereas Query Plan Language (QPL) preserves query structure through explicit operator plans [3]. VeriPIE builds on QPL with QuerySketch, an extended plan language that is elaborated (resolved and type-checked against the schema) into a common representation of database operations in Lean [2]. QuerySketch serves as a shared interface for validation, repair, SQL lowering (translation to executable SQL), and equivalence checking.

At selection time, execution voting chooses the most frequent result among sampled SQL candidates. This applies self-consistency [15] to text-to-SQL [13]. The method requires no reference query, but agreement on one database does not establish semantic equivalence, as distinct queries may return identical results on the observed instance yet diverge on another valid database. SpotIt mitigates this limitation through bounded verification with satisfiability modulo theories (SMT), a method for deciding logical constraints. It searches for counterexample databases—instances on which query results differ—within a fixed bound [1, 7]. A discovered counterexample separates the queries. A completed check finding none supports equivalence only within its encoded model and bound. VeriPIE first ranks candidates by execution consensus, then can use plan checks, question-derived checks on expected structure and outputs (answer contracts), and bounded counterexamples to revise the ranking when evidence conflicts. If verification is unavailable or inconclusive, it falls back to the execution-based ranking.

LogicTrack similarly uses formal logic solvers to audit intermediate reasoning and guide backtracking, which revisits failed reasoning steps [6]. VeriPIE applies a check-and-repair approach to executable query plans. TalkToModel offers a complementary natural-language interface for computing and discussing model explanations [12].

Our focus is query validity and candidate selection. Unlike approaches focused primarily on execution accuracy, VeriPIE emphasizes checkable computational evidence to support independent model validation in financial applications.

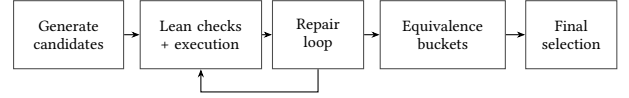


Figure 1: The VeriPIE workflow. Diagnostics guide repair, and revised candidates return to checking. Lean checks apply to QuerySketch plans. Execution agreement and bounded-equivalence evidence guide grouping and final selection.

3 THE VERIPIE APPROACH

Given a question, database schema S , and database instance D , VeriPIE follows the five stages in Figure 1: generate candidates, check and execute them, repair detected problems, group candidates, and select ranked answers. Direct SQL receives structural and execution checks. QuerySketch plans also pass through Lean-backed checking before SQL generation. Repair returns revised candidates to checking. Executable candidates are then organized into *equivalence-aware buckets*, groups informed by execution agreement and bounded-equivalence evidence. Reference SQL is not used during inference. The main results report the combined system, which ensembles the Direct-SQL and Lean-Backed pipelines described below. Their executable candidates enter a common grouping and selection stage.

The workflow separates two questions: whether a candidate describes an executable computation, and whether that computation is a suitable answer. Plan checking and execution address the first question, while grouping and selection use the resulting evidence to address the second. This separation gives each stage a specific role. A type error can trigger repair before selection, while disagreement between executable candidates can inform ranking. The shared plan representation connects these stages without requiring the language model to produce a proof of its answer.

3.1 Direct-SQL VeriPIE

VeriPIE generates SQL through standard generation, question decomposition with self-correction, and schema-focused refinement, then applies verification-guided selection without an IR. Its bounded SQL verification uses SpotIt+, which is added by VeriPIE and is not part of native DeepEye. Each candidate undergoes deterministic structural and execution checks covering parsing, executability, schema references, output shape (the returned columns and their types), and question-relevant features such as aggregation, grouping, ordering, and limits. Candidates that pass these checks are partitioned into buckets according to their observed results on the target database. Buckets are ranked primarily by candidate support, the number of queries in each bucket, providing an initial execution-consensus ranking.

Answer contracts combine model proposals with rules grounded in the question. Each Lean generation proposes a partial contract. After deterministic parsing, a field is retained only when more than half of all generation samples agree on its value. The resulting contract is checked. Agreement does not prove correctness. Direct-SQL also uses handwritten safeguards. One flags replacing an explicitly supplied $\text{SUM}(x)/\text{COUNT}(*)$ formula with $\text{AVG}(x)$: the former

counts all rows in its denominator, whereas the latter excludes null values.

3.2 Lean-Backed VeriPIE

Lean-Backed VeriPIE modifies both the candidate-generation and selection stages of the DeepEye-SQL pipeline. Instead of generating database-engine-specific SQL, the language model produces multiple QuerySketch plans. QuerySketch is a QPL-derived IR whose parser, abstract syntax tree (AST), elaborator, and typed representation of database operations (the relational core) are implemented in Lean. The parser builds the AST, a tree describing the plan syntax. The elaborator resolves references and checks types. Its numbered steps scan tables, filter rows, project selected columns, join tables, aggregate values, sort or limit results, or apply set operations to earlier results. QuerySketch also extends QPL with typed arithmetic, conditional, string, and date expressions needed for BIRD and Spider.

For each question, the model receives the database schema, a compact QuerySketch syntax specification, and several examples. It returns one or more textual QuerySketch programs rather than Lean terms or SQL. Appendix A.1 illustrates how a two-step QuerySketch plan makes intermediate column dependencies explicit.

After generation, each textual QuerySketch candidate is passed with schema S to the Lean frontend. Let t_i denote the generated text, p_i its QuerySketch AST, and r_i a relational plan whose types reflect S . The output s_i is SQL in dialect δ , the syntax used by the target database engine. The lowering pipeline is

$$t_i \xrightarrow{\text{parse}} p_i \xrightarrow{\text{elaborates}_S} r_i \xrightarrow{\text{lower}_\delta} s_i.$$

Elaboration resolves schema references and checks types, column origins, aggregation, grouping, and output shape. Valid plans are lowered to SQL, while invalid plans produce structured diagnostics for model repair.

Every accepted QuerySketch plan has passed the frontend’s checks for supported operations, schema references, and types. This gives a concrete validation guarantee at the plan level: acceptance depends on explicit checks, not the model’s confidence in its own output. Rejected plans receive diagnostics that identify what must be corrected. This evidence remains useful even when two methods have similar benchmark accuracy. It does not establish that SQL lowering preserves every aspect of plan meaning or that the plan answers the question.

The typed relational core also supports bounded equivalence checking. Direct-SQL VeriPIE encodes pairs of SQL queries, whereas this variant encodes the relational plans elaborated from QuerySketch candidates. Sharing one representation between SQL lowering and equivalence encoding avoids reparsing dialect-specific SQL and reduces opportunities for semantic drift between execution and verification.

3.3 Diagnostic Repair

Appendix A.1 gives a recorded repair example. Repair uses feedback from a failed check to revise the candidate that produced it. Feedback can concern plan structure and types, SQL execution, or the requested output. For example, an incompatible comparison calls for revisiting its operands, whereas a missing output column calls

for revisiting the projection. These are different failures, even if both prevent a candidate from answering the question. The diagnostic makes the reason for revision explicit.

The return arrow in Figure 1 is essential: a revised candidate must pass through checking again. For QuerySketch, the revised text is parsed and elaborated before SQL generation. The generated SQL is then checked through execution. Repair thus connects candidate generation to validation rather than treating a model’s revision as evidence of success. The repaired candidate contributes to selection through its checked plan and execution result, using the same grouping process as other executable candidates.

3.4 Equivalence-Aware Grouping and Selection

Candidates sampled from the same model can share errors. Moreover, distinct computations can agree on a particular database, so a popular execution result need not represent one consistent interpretation of the question. Bounded checking tests this apparent agreement rather than treating popularity as sufficient evidence.

For SQL candidates s_i and s_j , execution buckets group those whose results agree on D . These groups provide an initial execution-consensus ranking. Structural grouping matches normalized SQL or Lean plans while requiring compatible observed results. Bounded evidence can connect structurally different candidates or separate them with a counterexample. Write $\llbracket s \rrbracket_D$ for the result of query s on database D . Bounded verification searches for a database D' within a supported model and a finite search bound b such that

$$\llbracket s_i \rrbracket_{D'} \neq \llbracket s_j \rrbracket_{D'}.$$

A valid counterexample separates two candidates that may agree on D , but it does not identify which candidate matches the user’s intent. Grouping therefore informs selection alongside the question and structural checks, rather than replacing interpretation of the question.

A completed bounded check finding no counterexample provides evidence only within its encoded model and bound. The selector accepts the leading bucket when its execution support, structural checks, and verification evidence are consistent. Otherwise, it compares the strongest bucket representatives using question-specific checks, with execution voting as a fallback. Buckets are ranked first by sample count. For tied buckets, the earliest-generated valid candidate among the tied buckets is returned.

Lean checks plan structure and types. Bounded verification compares query results within its encoded semantics and bound. Unsupported constructs and timeouts remain unknown. The encoding must account for SQL behavior, including nulls, duplicate rows, ordering, and schema constraints.

For financial workflows, the advantage is a computation that can be examined before its output informs a decision. A reconciliation or transaction-review query can silently omit relevant records while returning plausible totals. QuerySketch exposes the operations, diagnostics identify rejected plans, and counterexamples reveal dependence on the observed data. Together, these provide concrete material for challenging a generated answer. Sampling more LLM candidates alone does not provide such checks. VeriPIE contributes a query-validation component to financial AI safety, with human interpretation still needed to assess the intended business meaning.

Table 1: Complete development-set results. Top-1 is first-query accuracy. Top-2 is coverage by either returned query (both percentages). Time is estimated mean seconds per question. Bold marks the highest accuracy or coverage.

Method	Main mechanism	Spider: test-suite evaluation			BIRD: execution evaluation		
		Top-1	Top-2	Time (s)	Top-1	Top-2	Time (s)
LLM listwise selection	Joint ranking of the candidate list	59.48	66.34	38.00	60.95	67.86	63.82
MAC-SQL	Schema selection, decomposition, refinement	64.99	68.09	5.26	63.89	67.01	19.06
DeepEye	Execution agreement, confidence-aware judging	65.67	69.05	35.44	68.45	73.66	60.58
DIN-SQL	Decomposition and self-correction	71.86	74.66	12.04	62.65	65.71	13.23
VeriPIE	Plan checks, repair, equivalence-aware selection	72.05	78.43	39.89	68.90	73.73	60.59

4 EXPERIMENTAL SETUP

4.1 Benchmarks and Baselines

We use Spider 1.0 [16], covering all 1,034 development questions across 20 databases, scored with the official test-suite evaluator [17]. For BIRD [9], we evaluate all 1,534 development questions across 11 databases. We use the November 2025 annotations with the benchmark databases and official execution accuracy. Spider requires agreement with the reference across all test databases. BIRD checks the benchmark instance alone, so coincidental agreement can count as correct. The scores are not directly comparable across benchmarks.

All methods use openai/gpt-5.6-luna through OpenRouter. Table 1 reports one evaluation run per system. Reasoning effort was medium and max_tokens was 16,384. Temperature and top_p used their default values. These are our local results, not published leaderboard scores. Under a 16-slot budget, combined VeriPIE uses eight Lean-backed candidates plus four standard, two decomposition/self-correction, and two schema-focused/refinement SQL candidates. Checking and repair work can differ. The Lean arm allowed one plan-repair round. Direct-SQL verification in the full-development replay used SpotIt+, with bound 2 and an 8-second timeout per checked pair.

We choose baselines that represent different ways to generate and select queries. DeepEye [8] generates 16 direct-SQL candidates and selects by execution agreement and its native ranking. The Direct-SQL-only ablation reuses these candidates, changing selection alone. LLM-only listwise selection jointly ranks the saved DeepEye direct-SQL candidate pool, providing a comparison with model-based ranking over those candidates. It does not use the Lean candidate pool.

MAC-SQL (2025) [14] uses agents for schema selection, question decomposition, and execution-guided refinement, making it a comparison for modular generation and repair. DIN-SQL (2023) [11] combines schema linking, query decomposition, SQL generation, and self-correction. Its use of NatSQL for complex queries also makes it relevant to intermediate-representation approaches. These methods provide established design comparisons. CHASE-SQL [10], which combines diverse generation with learned pairwise selection, provides a newer direction for comparison.

4.2 Evaluation Metrics

Top-1 is the primary metric and scores the first returned query. Top-2 measures whether either returned query passes, rather than automatic identification of the correct answer. For DIN-SQL and MAC-SQL, the second query comes from an independent complete generation attempt (Appendix A.3). Reference SQL is used only for scoring. Mean latency is estimated from parallel execution logs per question when returning up to two candidates.

5 RESULTS AND ANALYSIS

5.1 Main Results

Spider provides the clearest improvement. VeriPIE answers 745 questions correctly at top-1, compared with DeepEye’s 679, a net increase of 66 questions or 6.38 percentage points. Its top-2 shortlist covers 811 questions, compared with 714 for DeepEye and 772 for DIN-SQL, net increases of 97 and 39 questions. Relative to DIN-SQL, the top-1 difference is only two questions (0.19 points), while the top-2 difference is 3.77 points. The stronger shortlist result is therefore distinct from the small top-1 difference. On Spider, combining the paths improves top-1 accuracy by 3.68 percentage points over Direct-SQL alone and 0.87 over Lean-Backed alone (Table 2).

BIRD shows a smaller difference: VeriPIE exceeds DeepEye by 0.45 points in top-1 accuracy and 0.07 points in top-2 coverage. Within VeriPIE, the gap between top-1 and top-2 corresponds to 66 Spider questions and 74 BIRD questions for which the second candidate succeeds after the first fails. This leaves room to improve selection. In an interactive workflow, a knowledgeable user could inspect the alternative if the first answer appears wrong. This is a potential use of the shortlist rather than an evaluated user interface.

VeriPIE spends computation on plan checking, diagnostic repair, and bounded verification in addition to generation and execution. On Spider, estimated mean latency is 39.89 seconds, compared with 35.44 for DeepEye and 12.04 for DIN-SQL. DIN-SQL offers nearly the same top-1 accuracy with a shorter wait, while VeriPIE adds explicit checking evidence and broader shortlist coverage. On BIRD, VeriPIE and DeepEye have nearly identical latency (60.59 and 60.58 seconds). The design targets financial analysis and review, where an inspectable basis for accepting a query can matter alongside response time. These aggregate timings do not isolate the cost of individual checking stages.

5.2 Interpreting the Benchmark Difference

The scoring rules suggest one hypothesis for the different gains. If two candidates produce the same evaluated result on BIRD’s database, choosing between them cannot change that execution score. Splitting their bucket can still change its support and allow a different bucket to win. Spider’s test suite can distinguish candidates that agree on the original instance but differ on other databases. It can therefore reward a choice that BIRD’s single-instance metric cannot distinguish. On the fixed combined pool, adding bounded evidence to structural selection improves Spider top-1 by 1.16 points (Appendix A.5), but does not establish this explanation for the benchmark difference. Reference-based bounded evaluation, as in SpotIt [7], could test whether predictions differ in ways that execution accuracy misses.

In general, from an MRM perspective, identifying computational disagreements and producing reviewable counterexamples may be valuable even when benchmark accuracy improvements are modest.

6 CONCLUSION

We presented VeriPIE, a text-to-SQL pipeline that connects generated answers to checkable evidence. Direct-SQL VeriPIE combines execution agreement with structural checks and bounded counterexamples. Lean-Backed VeriPIE adds QuerySketch plans that are checked, repaired, and lowered through a typed relational core. Their ensemble joins flexible SQL generation with explicit plan validation. For financial AI safety and model risk management (MRM), the contribution is an inspectable computation that supports independent validation and effective challenge: independent reviewers can examine which operations passed checks, why a revision was needed, and where candidate answers disagree. Such evidence can strengthen AI governance without establishing semantic correctness or regulatory compliance.

Future work will evaluate financial reporting and regulatory workloads and extend counterexample-guided judging, repair, and efficient verification. Appendix A.4 details these directions.

ACKNOWLEDGMENTS

We are grateful to Ava Pettit, Preetha Saha, Mingshu Li, and Dhagash Mehta of BlackRock, Inc. for their close collaboration on this work, including discussions that shaped its motivation in financial model risk management, guidance on the design of the evaluation, and detailed review of the manuscript.

REFERENCES

- [1] Clark Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer International Publishing, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11
- [2] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28 (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37
- [3] Ben Eyal, Moran Mahabi, Ophir Haroche, Amir Bachar, and Michael Elhadad. 2023. Semantic Decomposition of Question and SQL for Text-to-SQL Parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Association for Computational Linguistics, 13629–13645. <https://doi.org/10.18653/v1/2023.findings-emnlp.910>
- [4] Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John 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*, Association for Computational Linguistics, 2030–2042. <https://doi.org/10.18653/v1/2021.findings-emnlp.174>
- [5] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, 4524–4535. <https://doi.org/10.18653/v1/P19-1444>
- [6] Jingyu Hu, Shu Yang, Weiru Liu, and Di Wang. 2026. LogicTrack: Auditing Reasoning Trajectories of Large Language Models with Formal Logic Solvers. *arXiv preprint arXiv:2609.21492* (2026).
- [7] Rocky Klopfenstein, Yang He, Andrew Tremante, Yuepeng Wang, Nina Nardyska, and Haoze Wu. 2026. SpotIt: Evaluating Text-to-SQL Evaluation with Formal Verification. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=IMkvR2ICSE>
- [8] Boyan Li, Chong Chen, ZhuJun Xue, Yinan Mei, and Yuyu Luo. 2026. DeepEye-SQL: A Software-Engineering-Inspired Text-to-SQL Framework. *Proceedings of the ACM on Management of Data* 4, 3, Article 158 (2026). [arXiv:2510.17586 \[cs.DB\]](https://doi.org/10.1145/3698866)
- [9] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2023. Can LLM Already Serve as a Database Interface? A BIG Bench for Large-Scale Database Grounded Text-to-SQLs. *Advances in Neural Information Processing Systems* 36 (2023), 42330–42357.
- [10] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Taleai, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2024. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. *arXiv preprint arXiv:2410.01943* (2024).
- [11] 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 (2023), 36339–36348.
- [12] Dylan Slack, Satyapriya Krishna, Himabindu Lakkaraju, and Sameer Singh. 2023. Explaining machine learning models with interactive natural language conversations using TalkToModel. *Nature Machine Intelligence* 5, 8 (2023), 873–883.
- [13] Ruoxi Sun, Sercan O. Arik, Alex Muzio, Lesly Miculicich, Satya Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, and Tomas Pfister. 2023. SQL-PaLM: Improved Large Language Model Adaptation for Text-to-SQL. *arXiv preprint arXiv:2306.00739* (2023).
- [14] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, et al. 2025. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. In *Proceedings of the 31st International Conference on Computational Linguistics*. 540–557.
- [15] 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*. <https://openreview.net/forum?id=1PLINIMMrw>
- [16] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 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*. 3911–3921.
- [17] 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)*. 396–411.

A ADDITIONAL EXAMPLES AND FUTURE WORK

A.1 QuerySketch and Repair Example

The TopSort operator sorts rows and keeps a requested number. For example, for non-null ages, “return the name of one youngest singer” becomes:

```
#1 = Scan Table [ singer ]
      Output [ name, age ];
#2 = TopSort [ #1 ] Rows [ 1 ]
      OrderBy [ age ASC ]
      Output [ name ]
```

The two steps make the requested computation explicit. The scan exposes both the name to return and the age needed for ordering. The second step selects a row using age and returns only its name. Column availability therefore matters between steps, even when a column is absent from the final answer. QuerySketch lets

the checker examine these intermediate dependencies before generating SQL. It also gives repair a structured plan to revise when a reference or type is incompatible with the schema.

One recorded example asks, “Show the stadium names without any concert.” Initial QuerySketch plans failed type checking. Diagnostic repair produced a compatible NOT EXISTS plan that passed the official Spider test suite. The example links a detected plan error to a successful revision.

A.2 Agreement Beyond the Observed Database

The account example in Section 1 illustrates why this extra evidence matters. Suppose the current payment table contains no null account identifiers. The two candidate queries can enter the same execution bucket because they return the same unmatched accounts. A valid database containing a null payment account identifier can distinguish the candidates even though their observed answers originally agreed. The counterexample supplies a concrete reason to reconsider the group: the apparent agreement depends on the current data. The question and schema then provide the context for choosing the intended treatment of unmatched accounts.

A.3 Two-Answer Baseline Protocol

DIN-SQL and MAC-SQL each obtain two answers through independent, complete runs of their own pipeline. Rank 1 retains the original final output. Rank 2 comes from another complete generation attempt, including the method’s normal refinement steps. Top-2 scores success by either final output. A single reported evaluation thus includes two generation attempts per question for these baselines.

A.4 Further Evaluation and Verification

Future work will evaluate financial schemas and recurring reporting queries. We will also compare each system’s BIRD predictions with reference SQL using bounded equivalence checking, reporting counterexamples, completed bounded checks, and unknown outcomes separately to study coincidental correctness.

For judging, an LLM or user could inspect competing outputs on a separating database and assess which matches the question. For repair, the generator could receive that database and the differing outputs to revise a candidate. These are proposed extensions rather than evaluated features.

Selective verification when leading buckets are close may reduce latency. Additional directions include comparisons with CHASE-SQL [10], broader SQL dialect support, and proofs that plan transformations preserve query meaning.

A.5 Generation and Selection Ablations

An ablation removes or replaces a component to measure its contribution. We distinguish candidate generation from selection among candidates. Table 2 reports Spider top-1 accuracy.

Candidate generation. DeepEye generates 16 direct-SQL candidates and uses its native selector. The Direct-SQL-only ablation uses the same 16 candidates with structural and bounded-equivalence evidence, isolating a selection change without adding Lean-generated candidates. Lean-Backed VeriPIE instead generates 16 QuerySketch

Table 2: Spider top-1 accuracy. Higher values are better. All generation variants start with 16 slots. Selection variants reuse the combined 8+8 pool.

Configuration	Top-1 (%)
VeriPIE	72.05
<i>Candidate generation</i>	
Direct-SQL only (16 direct)	68.37
Lean-Backed only (16 Lean)	71.18
<i>Selection on the same combined pool</i>	
Execution-only selection	71.28
Structural selection	70.89
<i>Plan repair in the combined pipeline</i>	
Without Lean plan repair	71.76

plans, checks them in Lean, permits repair, and lowers valid plans to SQL. Combined VeriPIE uses four standard direct-SQL candidates, two generated through decomposition and self-correction, two through schema-focused refinement, and eight Lean-backed candidates, followed by structural and bounded-equivalence selection.

Selection and repair. The three selection configurations reuse the same combined 8+8 candidate pool and retain the shared question-based safeguards. Execution-only selection groups candidates by their observed database results and ranks by support. Structural selection groups matching normalized SQL or Lean plans, requiring compatible observed results. Structural plus bounded selection, used by combined VeriPIE, additionally uses verifier evidence to connect structurally different candidates and separate candidates when a counterexample is found. Execution-only selection is therefore not native DeepEye: its input includes Lean-generated candidates and it retains VeriPIE’s question-based safeguards.

The repair ablation uses the original Lean plans before repair, retains invalid slots, and recomputes the relevant SQL checks. It changes candidate validity within the combined slot allocation rather than only replacing the selector on an unchanged executable pool.

Interpretation. Combining generation paths achieves the best Spider top-1 accuracy under the same 16-slot budget, exceeding both paths alone. On the fixed combined pool, structural plus bounded selection gains 1.16 percentage points over structural selection and 0.77 over execution-only selection. Structural selection alone does not improve on execution-only selection. Removing Lean plan repair costs 0.29 points. These comparisons support complementary generation and a benefit from bounded selection evidence in this experiment.