

PExA: Parallel Exploration Agent for Complex Text-to-SQL

Tanmay Parekh^{1*} Ella Hofmann-Coyle^{2†} Shuyi Wang²
Sachith Sri Ram Kothur² Srivas Prasad² Yunmo Chen^{2†}

¹University of California Los Angeles

²Bloomberg

tparekh@cs.ucla.edu, skothur@bloomberg.net

Abstract

LLM-based agents for text-to-SQL often struggle with a latency-performance trade-off, where performance improvements come at the cost of latency or vice versa. We cast text-to-SQL generation as a problem of passing software test coverage. Similar to its counterpart in software engineering, the original query is prepared with a suite of test cases with simpler, atomic SQL queries that are executed in parallel and together ensure semantic coverage of the original query. After iterating on test case coverage, the final SQL is generated only when enough information is gathered, using the explored test case SQL queries to ground the final generation. We validated our framework on a state-of-the-art benchmark for text-to-SQL, Spider 2.0 (Lei et al., 2025), achieving a new state-of-the-art accuracy of 70.2%.¹

1 Introduction

Natural-language (NL) interfaces to databases unlock enterprise analytics for non-coding experts by translating NL questions into executable SQL queries (Zelle and Mooney, 1996; Yu et al., 2018). While prior text-to-SQL work has shown strong performance on simpler academic benchmarks like SPIDER (Yu et al., 2018) and BIRD (Li et al., 2023), these often do not capture the full complexity of real-world use. To evaluate text-to-SQL in more realistic settings, Lei et al. (2025) introduced Spider 2.0, presenting significant challenges such as large cross-domain databases, nested data types, and long, complex SQL queries.

To address these complex challenges, researchers have increasingly turned to tool-augmented LLMs (Nakano et al., 2021; Schick et al., 2023), deep planning (Wei et al., 2022; DeepSeek-AI et al., 2025), and agentic workflows

(Yao et al., 2023b; Shinn et al., 2023). Recent work in text-to-SQL has also explored such frameworks (Xie et al., 2024; Lei et al., 2025). However, these advanced methods face a well-known latency-performance trade-off (Snell et al., 2024; Chen et al., 2024). Performance improvements are often achieved through complex reasoning, multiple tool calls, and iterative self-correction, all of which substantially increase latency, making them less suitable for interactive analytics.

To achieve better Pareto optimality in this trade-off, we propose a novel framework that *reframes the text-to-SQL task from a software testing perspective*. Rather than treating the NL query as a single complex problem to be solved sequentially, we treat it instead as a set of semantic requirements that must be “covered.” The agent’s goal is to first generate a set of robust “test cases” – simpler, self-contained SQL queries – that collectively cover the full semantics of the user’s question and validate against the database. To boost coverage, test cases are intentionally *over-generated* so that target and non-target database information can both be explored to guide the final SQL generation.

This testing paradigm is effective because it naturally enables parallelism. Rather than a sequential chain of thought, we introduce diverse planning, concurrent execution, and single-step multi-path search, which enables parallel planning and execution of test case SQL. This parallel exploration facilitates the gathering of all necessary information (e.g., table structures, value distributions, intermediate results) simultaneously and flattens the latency curve by bounding latency by the slowest operation rather than the sum of all operations. Furthermore, it also improves performance by broader and more efficient semantic search.

To operationalize this paradigm, we develop PEXA, an agent comprising three specialized sub-agents: (1) *Planner*: Decomposes the original problem into a set of self-contained, simpler questions

* This work was done during an internship at Bloomberg.

† Research performed while working at Bloomberg.

¹ At the time of submission on November 15, 2025.

for test case generation. (2) *Test Case Generator*: Builds, revises, and executes the corresponding SQL test cases against the database in parallel to gather information. (3) *Proposer*: Integrates the information gathered from the test cases to generate the final, long-form SQL for the original user query.

Our contributions include:

- We reformulate the text-to-SQL task within the context of software testing and introduce parallelism to achieve better Pareto optimality between the latency and performance;
- We develop an agent, PEXA, to operationalize our proposed formulation and validate it by achieving the state-of-the-art¹ on the complex Spider 2.0 benchmark.

2 Related Work

Here, we provide a brief overview of related work, with additional details in Appendix A.

Text-to-SQL Spider 1.0 (Yu et al., 2018) and BIRD (Li et al., 2023) serve as foundational datasets for text-to-SQL, whereas Spider 2.0 (Lei et al., 2025) is a recent, more challenging dataset to better evaluate methods on real-world cases. While initial works focused on fine-tuning (Scholak et al., 2021; Li et al., 2024b), advances in LLMs introduced better prompting-based methods (Pourreza and Rafiei, 2023). Recent frameworks also explore agents (Xie et al., 2024; Wang et al., 2025a) and inference-time scaling (Lee et al., 2025; Deng et al., 2025) to tackle complex text-to-SQL. Similarly, we focus on building an agent but use test cases as a way to interact with databases.

Latency-Performance Trade-off Prior work has explored simple task decomposition (Zhou et al., 2023b; Khot et al., 2023) and sequential reasoning (Wei et al., 2022; Yao et al., 2023a; Zhou et al., 2023a) to improve model performance. However, they require sequential execution and can increase latency. Task decomposition is also limited to explore only sub-tasks within the original query, whereas our test cases explore a wider surface of the database robustly and provide the final SQL generation with extensive grounded context. Inference-time scaling (Wang et al., 2023b; Lightman et al., 2024) and breadth-search (Saha et al., 2024; Lu et al., 2025) enable multi-path search, but provide limited control over directions and thus are not efficient. Our parallel exploration offers

simple single-step parallel execution and a better guided multi-path search, providing an efficient, low-latency setting without degrading final SQL generation quality.

Software Testing Robust systems are built not by solving the full complexity of a program at once, but by ensuring that its behavior satisfies a broad suite of targeted tests (Reid, 2005; Pressman, 2005). The philosophy to “cover” specific functional requirements lies at the core of modern testing and test-driven development practices (Fowler, 2002; Beck, 2003).

3 Approach

3.1 A Software Testing Perspective

Inspired by software testing, we formulate text-to-SQL as a test coverage problem: *creating a set of simpler, self-contained unit test cases that collectively “cover” the semantics of the user query and validate against the database*. Different from query decomposition (Deng et al., 2025, *inter alia*) where semantics present in sub-queries always reside in the original query, our testing paradigm covers a larger semantic surface via tests that extend beyond the original query to derive and explore additional information present in the database. We encode this philosophy in PEXA as described in Figure 1.

3.1.1 Sub-Agents

Planner The Planner implements the *Test Planning and Modularization* principle in software testing (Parnas, 1972), determining the different semantic requirements of the user query and surrounding it with a suite of simpler verifiable (unit) test cases. In our framework, we define a test case as a self-contained SQL query that validates semantics present in or related to the user query – such as a particular filter, join, aggregation, or existence check – and provides independent and diverse evidence toward reconstructing the full SQL for the user query.

In addition, the Planner serves as the central control module, orchestrating the interaction between the Test Case Generator and the SQL Proposer. Such interactions are performed through test cases and their execution results. The Planner determines when to continue exploratory test-case generation, and when sufficient evidence has been accumulated to invoke the Proposer for final SQL synthesis. It also performs a final verification step, assessing the

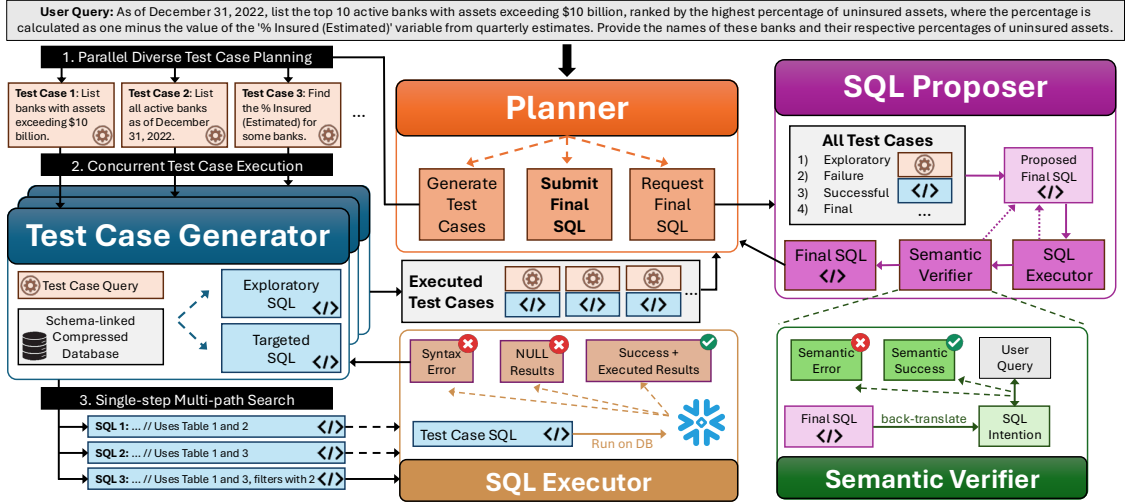


Figure 1: Illustration of our framework PEXA, comprising three sub-agents – Planner, Test Case Generator, and SQL Proposer – and two tools – SQL Executor and Semantic Verifier. In the black boxes (left), we indicate how we induce parallelism in our framework. We indicate deterministic and agentic decision choices in terms of flow using solid and dotted arrows respectively.

Proposer sub-agent’s output before returning the final SQL and terminating.

Test Case Generator The Test Case Generator sub-agent corresponds to the *Test Execution* phase in software testing (Ammann and Offutt, 2008), where individual test cases are instantiated and run to gather empirical evidence. The Test Case Generator translates each Planner-issued NL test case into an executable SQL query. Given the potentially large and heterogeneous database, we follow ReForCE (Deng et al., 2025) to perform lightweight LLM-based schema linking and database compression, identifying the relevant tables/columns required for the main user query with a focus on recall. Conditioned on this pruned schema context, the sub-agent then synthesizes a precise SQL for each test case. Importantly, we link the sub-agent with the *SQL Executor* tool to build a feedback mechanism, verifying that the generated SQL executes successfully and entails the NL test case.

SQL Proposer The SQL Proposer sub-agent aligns with the *Test Integration* stage in software testing (Ammann and Offutt, 2008), where the results of multiple unit test cases are consolidated to derive the system’s overall intended behavior. In our framework, given the executed test cases and the original user query, the SQL Proposer synthesizes the final long-form SQL by combining the partial semantic signals surfaced through the test-case results. To maintain a focused and efficient

context, the Proposer is intentionally not supplied with full database metadata and instead relies *solely on the distilled evidence provided by the test cases*. The module is further equipped with the same *SQL Executor* and a *Semantic Verifier* to ensure both valid execution and semantic fidelity before returning the final SQL.

3.1.2 Tools

SQL Executor The SQL Executor tool uses a SQL execution engine to execute the query on the database. Based on the execution, it provides three kinds of feedback: (1) **Compilation Error**, if there is a syntax issue, (2) **NULL Error**, if the SQL executes but produces zero results, and (3) **Success**, if the SQL executes with non-zero results, along with the truncated executed results.

Semantic Verifier This tool utilizes an LLM to ensure alignment between the NL instruction and generated SQL. To do this, the proposed SQL is first back-translated into natural language, and then compared to the original user query to identify any semantic disparities. If a huge disparity is found, the tool returns a semantic error.

3.2 Parallel Exploration

Another major advantage of adopting a software testing perspective is that test cases are *inherently parallelizable*. We exploit such parallel exploration in PEXA to boost performance and latency in three different ways, as described below.

Diverse Test Case Planning Our Planner prepares the user query with a suite of independent, self-contained test cases, running the entire planning phase in a single forward pass. This parallel testing plan not only lowers planning overhead but also yields a more diverse set of semantic test cases, improving downstream coverage.

Parallel Test Case Execution Since the test cases are designed to not interact with each other, Test Case Generator can run all of them in parallel. This parallel execution increases semantic breadth without scaling end-to-end latency, analogous to running a fully parallel test suite rather than a sequential battery of checks.

Single-step Multi-path Search For each test case being generated, we prompt PEXA to produce multiple candidate SQL queries with diverse reasoning strategies in one LLM generation call, enabling a structured multi-path search without sequential calls. This single-step branching expands the search frontier early, allowing rapid pruning of weak hypotheses and more efficient exploration compared to sequential multi-solution exploration (Yao et al., 2023a). This parallelization strategy bounds the time complexity to the slowest explored path, rather than the sum of all paths in an iterative framework.

4 Experiments and Results

Dataset We evaluate on the complex text-to-SQL Spider 2.0 benchmark, specifically on the Snow and Lite* versions. Spider 2.0-Snow comprises 547 examples that span 150+ databases, with an average of 800 columns per database in the Snowflake dialect. Spider 2.0-Lite* is a Lite version of the dataset, excluding the examples from BigQuery.²

Evaluation Metrics We follow Yu et al. (2018) to consider Execution Accuracy (EX) and EX@4 (pass @ k) to measure model performance.³ For latency measurement, we report average wall time per user query. GPT-o3 is used in all major experiments in this paper.

² Due to legal and cost constraints, we exclude BigQuery execution from our experiments.

³ To improve accuracy and stability in evaluations, the gold outputs for this dataset were updated on October 29, 2025. In this case, *exact numbers of metrics might differ when compared to prior work*. As most of our experiments were conducted prior to this update, we report our ablation and analytical results under the old evaluation setup. However, to be noted, our state-of-the-art performance at the time of submission was based on the updated evaluation data.

Method	Snow		Lite*		Wall
	EX	EX@4	EX	EX@4	Time
Spider-Agent	25.2	27.4	26.2	28.7	5.90
ReFoRCE	36.6	39.7	36.2	39.5	5.44
Chat2DB	44.1 ¹	–	–	–	–
AgenticData	–	–	44.5 ^{1,2}	–	–
PEXA (ours)	45.7	49.5	46.6	49.9	5.55

¹ As reported in the older leaderboard.

² On complete Lite dataset.

Table 1: Main results comparing the performance (EX, EX@4) and wall time (in mins) for PEXA against open-source baselines.

Setting	EX	Δ
PEXA (Full)	42.9	–
w/o Plan-time parallelization	40.0	-2.9
w/o Test-time parallelization	39.9	-3.0
w/o Semantic Verifier	42.3	-0.6
w/o Proposer	41.1	-1.8

Table 2: Ablation study showing the performance impact when removing specific components from PEXA.

Baselines For baselines, we consider other open-source work on the Spider 2.0 leaderboard, specifically: (1) Spider-Agent (Lei et al., 2025), a bash-based LLM agent, (2) ReFoRCE (Deng et al., 2025), a database compression and inference-time scaling method, (3) Chat2DB-Agent and AgenticData (Sun et al., 2025)⁴.

4.1 Main Results

We present our main results in Table 1. PEXA utilizes the inference-time scaling with majority consensus over four runs and outperforms all other baselines. On the updated evaluation data, PEXA achieves a new state-of-the-art ¹ of 70.2% at the time of submission as shown in § C.10. Meanwhile, parallel exploration ensures efficient and comparable wall time for PEXA. We further provide a detailed theoretical and empirical study between sequential and parallel execution in Appendix C.1.

4.2 Analyses

For the various analyses, we consider a single run of PEXA on the Snow dataset, rather than the inference-time scaling performance (as reported

⁴ As the complete codebase is not available at the time of submission, we could not reproduce the results for comparison. Hence, we took the numbers directly from the leaderboard.

Plan	Generate	Propose	EX
<i>Homogeneous models</i>			
	GPT-o3		42
	GPT-5		45
	Sonnet-4		38
	Opus-4		38
<i>Mixed models (Ablation)</i>			
Sonnet-4	GPT-o3	GPT-o3	44
GPT-o3	Sonnet-4	GPT-o3	40
GPT-o3	GPT-o3	Sonnet-4	39

Table 3: Performance analysis comparing homogeneous against mix-and-match configurations on 100 samples from the Spider 2.0 dataset.

in Table 1). Additional analyses and results in C.

Component Ablation We ablate various components used by our framework in Table 2 (more details about model setup explained in Appendix C.4). Parallelization components are the most significant ones towards performance, while the semantic verifier and proposer also contribute to performance to some extent. This analysis also demonstrates how the parallelization branching factor can be controlled for faster inference and lower model costs while incurring merely 2-3% performance drop.

Studying LLM Synergies The components used by our agent can utilize different LLMs, enabling ablations of different LLMs and their synergies when used together. In Table 3, we study such effects with four LLMs.⁵ With our agent setup, we found that GPT-o3 and GPT-5 perform the best, while Claude models are slightly underperforming. Component-wise ablations with Claude Sonnet-4 over GPT-o3 baseline revealed that Claude performs well on planning, while being poor at long-context/horizon reasoning for SQL generation. The ability to combine different models allows a fine-grained control for our agent to achieve a better balance between performance and latency.

Controlling the Efficiency-Performance Trade-off PEXA offers granular control over the trade-off between computational cost and performance. While our default configuration leaves the search space unconstrained, we can restrict the branching factor during both the planning and execution phases to improve efficiency. Table 4 quantifies

⁵Owing to higher costs of running frontier LLMs, we conduct this experiment on 100 samples from the dataset

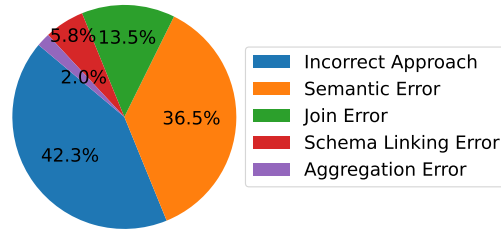


Figure 2: Error analysis and categorization on the major error categories from PEXA.

Settings	Execution Branches			
	1	2	∞	
Plan Branches	1	38.4	39.1	40.0
	2	38.9	39.5	41.1
	∞	39.9	41.6	42.9

Table 4: Performance analysis varying the parallelization degree. ∞ indicates no limit enforced.

this relationship. We observe a 4.5% drop in execution accuracy when limiting the system to a single branch compared to the unconstrained baseline. Consequently, increasing the branching factor consistently improves performance by enabling broader search coverage.

Error Analysis We analyzed a sample of failure cases by comparing generated queries against gold reference SQL queries shown in Figure 2. The breakdown reveals that semantic misinterpretations and flaws in initial planning/approaches are the dominant error sources, rather than simple syntactic mistakes. These high-level reasoning failures represent the primary bottleneck and suggest that future iterations should prioritize improved question comprehension and improve search in plans.

5 Conclusion and Future Work

In this work, we present PEXA, a text-to-SQL agent composed of a Planner, Test Case Generator, and SQL Proposer sub-agents. By utilizing parallelized exploration, PEXA efficiently traverses the search space even under strict latency constraints. Empirically, we demonstrate that our approach achieved state-of-the-art results on Spider 2.0 while maintaining wall-time comparable to existing baselines. Future works can explore adapting our parallelized agentic architecture for wider benchmarking on other Text-to-SQL datasets, as well as broader software engineering and code generation tasks.

Acknowledgements

Tanmay Parekh would like to express his gratitude to Bloomberg for their support through the Bloomberg Data Science Ph.D. Fellowship.

Limitations

Our baselines represent the state-of-the-art as of November 2025. While newer approaches may have emerged since, our baseline analysis is constrained as these methods are not open, preventing fair benchmarking. Apart, we do not conduct any post-training while developing our framework. While post-training can improve the agentic capabilities of the base LLM, we believe it can hamper the robustness. Future works can explore better post-training with PEXA.

Ethical Considerations

Our framework utilizes state-of-the-art LLMs like GPT-o3 off-the-shelf, and we do not filter/check for bias mitigation while using these LLMs. For practical utility, it is advised to consider separate bias mitigation strategies. Also, since our framework is agentic in nature, it generates many LLM calls for the various agentic actions. Thus, running PEXA can be computationally expensive and should be considered accordingly. We also acknowledge the utilization of LLMs/AI for paper writing, and coding.

References

- Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press.
- Samuel David Arcadinho, David Aparicio, Hugo Veiga, and Antonio Alegria. 2022. [T5QL: Taming language models for SQL generation](#). In *Proceedings of the Second Workshop on Natural Language Generation, Evaluation, and Metrics (GEM)*, pages 276–286, Abu Dhabi, United Arab Emirates (Hybrid). Association for Computational Linguistics.
- Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. [\$\tau^2\$ -bench: Evaluating conversational agents in a dual-control environment](#). *CoRR*, abs/2506.07982.
- Kent Beck. 2003. *Test-driven development: by example*. Addison-Wesley Professional.
- Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoeffler. 2024. [Graph](#)
- [of thoughts: Solving elaborate problems with large language models](#). In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 17682–17690. AAAI Press.
- Jipeng Cen, Jiaxin Liu, Zhixu Li, and Jingjing Wang. 2025. [Sqlfixagent: Towards semantic-accurate text-to-sql parsing via consistency-enhanced multi-agent collaboration](#). In *AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA*, pages 49–57. AAAI Press.
- Yanxi Chen, Xuchen Pan, Yaliang Li, Bolin Ding, and Jingren Zhou. 2024. [A simple and provable scaling law for the test-time compute of large language models](#). *CoRR*, abs/2411.19477.
- DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 81 others. 2025. [Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning](#). *CoRR*, abs/2501.12948.
- Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. 2025. [Reforce: A text-to-sql agent with self-refinement, format restriction, and column exploration](#). *CoRR*, abs/2502.00675.
- Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. [Structure-grounded pretraining for text-to-SQL](#). In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1337–1350, Online. Association for Computational Linguistics.
- Shaoming Duan, Youxuan Wu, Chuanyi Liu, Yuhao Zhang, Zirui Wang, Peiyi Han, Shengyuan Yu, Liang Yan, and Yingwei Liang. 2025. [DSQG-syn: Synthesizing high-quality data for text-to-SQL parsing by domain specific question generation](#). In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 2971–2989, Albuquerque, New Mexico. Association for Computational Linguistics.
- Martin Fowler. 2002. [Refactoring: Improving the design of existing code](#). In *Extreme Programming and Agile Methods - XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4-7, 2002, Proceedings*, volume 2418 of *Lecture Notes in Computer Science*, page 256. Springer.
- Yujian Gan, Xinyun Chen, Qiuping Huang, and Matthew Purver. 2022. [Measuring and improving](#)

- compositional generalization in text-to-SQL via component alignment. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 831–843, Seattle, United States. Association for Computational Linguistics.
- Yujian Gan, Xinyun Chen, and Matthew Purver. 2021. Exploring underexplored limitations of cross-domain text-to-SQL generalization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8926–8931, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.
- Chunxi Guo, Zhiliang Tian, Jintao Tang, Shasha Li, Zhihua Wen, Kaixuan Wang, and Ting Wang. 2023. Retrieval-augmented gpt-3.5-based text-to-sql framework with sample-aware prompting and dynamic revision chain. *CoRR*, abs/2307.05074.
- Mingqian He, Yongliang Shen, Wenqi Zhang, Qiuying Peng, Jun Wang, and Weiming Lu. 2025. *STaR-SQL: Self-taught reasoner for text-to-SQL*. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 24365–24375, Vienna, Austria. Association for Computational Linguistics.
- Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2023. *Decomposed prompting: A modular approach for solving complex tasks*. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.
- Mathew J. Koretsky, Maya Willey, Adi Asija, Owen Bianchi, Chelsea X. Alvarado, Tanay Nayak, Nicole Kuznetsov, Sungwon Kim, Mike A. Nalls, Daniel Khashabi, and Faraz Faghri. 2025. *Biomedsql: Text-to-sql for scientific reasoning on biomedical knowledge bases*. *CoRR*, abs/2505.20321.
- Rahul Kumar, Amar Raja Dibbu, Shrutendra Harsola, Vignesh Subrahmaniam, and Ashutosh Modi. 2024. *BookSQL: A large scale text-to-SQL dataset for accounting domain*. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 497–516, Mexico City, Mexico. Association for Computational Linguistics.
- Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025. *MCS-SQL: Leveraging multiple prompts and multiple-choice selection for text-to-SQL generation*. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 337–353, Abu Dhabi, UAE. Association for Computational Linguistics.
- Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. 2025. *Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows*. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.
- Chunhui Li, Yifan Wang, Zhen Wu, Zhen Yu, Fei Zhao, Shujian Huang, and Xinyu Dai. 2024a. *MultiSQL: A schema-integrated context-dependent Text2SQL dataset with diverse SQL operations*. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13857–13867, Bangkok, Thailand. Association for Computational Linguistics.
- Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024b. *Codes: Towards building open-source language models for text-to-sql*. *CoRR*, abs/2402.16347.
- Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiayi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. *Can LLM already serve as A database interface? A big bench for large-scale database grounded text-to-sqls*. *CoRR*, abs/2305.03111.
- Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. *Let’s verify step by step*. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.
- Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. *Bridging textual and tabular data for cross-domain text-to-SQL semantic parsing*. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4870–4888, Online. Association for Computational Linguistics.
- Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S. Yu. 2023. *A comprehensive evaluation of chatgpt’s zero-shot text-to-sql capability*. *CoRR*, abs/2303.13547.
- Tao Liu, Hongying Zan, Yifan Li, Dixuan Zhang, Lulu Kong, Haixin Liu, Jiaming Hou, Aoze Zheng, Rui Li, Yiming Qiao, Zewei Luo, Qi Wang, Zhiqiang Zhang, Jiayi Li, Supeng Liu, Kunli Zhang, and Min Peng. 2025. *Logiccat: A chain-of-thought text-to-sql benchmark for multi-domain reasoning challenges*. *CoRR*, abs/2505.18744.
- Zhengdong Lu, Weikai Lu, Yiling Tao, Yun Dai, Zixuan Chen, Huiping Zhuang, Cen Chen, Hao Peng, and Ziqian Zeng. 2025. *Decompose, plan in parallel, and merge: A novel paradigm for large language models based planning with multiple constraints*. *CoRR*, abs/2506.02683.
- Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders,

- Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. 2021. [Webgpt: Browser-assisted question-answering with human feedback](#). *CoRR*, abs/2112.09332.
- Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. 2023. [Enhancing text-to-SQL capabilities of large language models: A study on prompt design strategies](#). In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 14935–14956, Singapore. Association for Computational Linguistics.
- Zhihong Pan, Kai Zhang, Yuze Zhao, and Yupeng Han. 2025. [Route to reason: Adaptive routing for LLM and reasoning strategy selection](#). *CoRR*, abs/2505.19435.
- Tanmay Parekh, Yuxuan Dong, Lucas Bandarkar, Artin Kim, I-Hung Hsu, Kai-Wei Chang, and Nanyun Peng. 2025a. [SNaRe: Domain-aware data generation for low-resource event detection](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 20583–20604, Suzhou, China. Association for Computational Linguistics.
- Tanmay Parekh, Jeffrey Kwan, Jiarui Yu, Sparsh Johri, Hyosang Ahn, Sreya Muppalla, Kai-Wei Chang, Wei Wang, and Nanyun Peng. 2024. [SPEED++: A multilingual event extraction framework for epidemic prediction and preparedness](#). In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 12936–12965, Miami, Florida, USA. Association for Computational Linguistics.
- Tanmay Parekh, Kartik Mehta, Ninareh Mehrabi, Kai-Wei Chang, and Nanyun Peng. 2025b. [DiCoRe: Enhancing zero-shot event detection via divergent-convergent LLM reasoning](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 20560–20582, Suzhou, China. Association for Computational Linguistics.
- Tanmay Parekh, Pradyot Prakash, Alexander Radovic, Akshay Shekher, and Denis Savenkov. 2025c. [Dynamic strategy planning for efficient question answering with large language models](#). In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 6038–6059, Albuquerque, New Mexico. Association for Computational Linguistics.
- David Lorge Parnas. 1972. [On the criteria to be used in decomposing systems into modules](#). *Commun. ACM*, 15(12):1053–1058.
- Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Ö. Arik. 2025. [CHASE-SQL: multi-path reasoning and preference optimized candidate selection in text-to-sql](#). In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.
- Mohammadreza Pourreza and Davood Rafiei. 2023. [DIN-SQL: decomposed in-context learning of text-to-sql with self-correction](#). In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
- Mohammadreza Pourreza and Davood Rafiei. 2024. [DTS-SQL: Decomposed text-to-SQL with small large language models](#). In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 8212–8220, Miami, Florida, USA. Association for Computational Linguistics.
- Roger S Pressman. 2005. *Software engineering: a practitioner’s approach*. Palgrave macmillan.
- Sandra Rapps and Elaine J. Weyuker. 1985. [Selecting software test data using data flow information](#). *IEEE Trans. Software Eng.*, 11(4):367–375.
- Stuart Reid. 2005. [The art of software testing, second edition](#). glenford j. myers. revised and updated by tom badgett and todd m. thomas, with corey sandler. john wiley and sons, new jersey, usa, 2004, ISBN 0-471-46912-2. *Softw. Test. Verification Reliab.*, 15(2):136–137.
- Swarnadeep Saha, Omer Levy, Asli Celikyilmaz, Mohit Bansal, Jason Weston, and Xian Li. 2024. [Branch-solve-merge improves large language model evaluation and generation](#). In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 8352–8370, Mexico City, Mexico. Association for Computational Linguistics.
- Riya Sawhney, Samrat Yadav, Indrajit Bhattacharya, and Mausam. 2025. [Iterative repair with weak verifiers for few-shot transfer in KBQA with unanswerability](#). In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 24578–24596, Vienna, Austria. Association for Computational Linguistics.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. [Toolformer: Language models can teach themselves to use tools](#). *CoRR*, abs/2302.04761.
- Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. [PICARD: Parsing incrementally for constrained auto-regressive decoding from language models](#). In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.
- Zhili Shen, Pavlos Vougiouklis, Chenxin Diao, Kausubh Vyas, Yuanyi Ji, and Jeff Z. Pan. 2024. [Improving retrieval-augmented text-to-SQL with AST-based ranking and schema pruning](#). In *Proceedings of the 2024 Conference on Empirical Methods in*

- Natural Language Processing*, pages 7865–7879, Miami, Florida, USA. Association for Computational Linguistics.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. [Reflexion: language agents with verbal reinforcement learning](#). In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
- Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. [Scaling LLM test-time compute optimally can be more effective than scaling model parameters](#). *CoRR*, abs/2408.03314.
- Ji Sun, Guoliang Li, Peiyao Zhou, Yihui Ma, Jingzhe Xu, and Yuan Li. 2025. [Agentdata: An agentic data analytics system for heterogeneous data](#). *CoRR*, abs/2508.05002.
- Shayan Talaie, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. [CHESS: contextual harnessing for efficient SQL synthesis](#). *CoRR*, abs/2405.16755.
- Dayton G. Thorpe, Andrew J. Duberstein, and Ian A. Kinsey. 2024. [Dubo-sql: Diverse retrieval-augmented generation and fine tuning for text-to-sql](#). *CoRR*, abs/2404.12560.
- Immanuel Trummer. 2022. [Codexdb: Generating code for processing SQL queries using GPT-3 codex](#). *CoRR*, abs/2204.08941.
- Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020a. [RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers](#). In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, Online. Association for Computational Linguistics.
- Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025a. [MAC-SQL: A multi-agent collaborative framework for text-to-SQL](#). In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557, Abu Dhabi, UAE. Association for Computational Linguistics.
- Dingzirui Wang, Longxu Dou, Xuanliang Zhang, Qingfu Zhu, and Wanxiang Che. 2024a. [Improving demonstration diversity by human-free fusing for text-to-SQL](#). In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 1193–1207, Miami, Florida, USA. Association for Computational Linguistics.
- Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023a. [Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models](#). In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2609–2634, Toronto, Canada. Association for Computational Linguistics.
- Lilin Wang, Lucas Ramalho, Alan Celestino, Phuc Anthony Pham, Yu Liu, Umang Kumar Sinha, Andres Portillo, Onassis Osunwa, and Gabriel Maduekwe. 2025b. [Swe-bench++: A framework for the scalable generation of software engineering benchmarks from open-source repositories](#). *arXiv preprint arXiv:2512.17419*.
- Ping Wang, Tian Shi, and Chandan K. Reddy. 2020b. [Text-to-sql generation for question answering on electronic medical records](#). In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 350–361. ACM / IW3C2.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023b. [Self-consistency improves chain of thought reasoning in language models](#). In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.
- Zhongyuan Wang, Richong Zhang, Zhijie Nie, and Jaemin Kim. 2024b. [Tool-assisted agent on SQL inspection and refinement in real-world scenarios](#). *CoRR*, abs/2408.16991.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. 2022. [Chain of thought prompting elicits reasoning in large language models](#). *CoRR*, abs/2201.11903.
- Wenxuan Xie, Gaochen Wu, and Bowen Zhou. 2024. [MAG-SQL: multi-agent generative approach with soft schema linking and iterative sub-sql refinement for text-to-sql](#). *CoRR*, abs/2408.07930.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. [Tree of thoughts: Deliberate problem solving with large language models](#). In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023b. [React: Synergizing reasoning and acting in language models](#). In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.
- Zhewei Yao, Guoheng Sun, Lukasz Borchmann, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. 2025. [Arctic-text2sql-r1: Simple rewards, strong reasoning in text-to-sql](#). *CoRR*, abs/2505.20315.
- Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. [TaBERT: Pretraining for joint](#)

- understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426, Online. Association for Computational Linguistics.
- Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 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*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.
- John M. Zelle and Raymond J. Mooney. 1996. [Learning to parse database queries using inductive logic programming](#). In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2*, pages 1050–1055. AAAI Press / The MIT Press.
- Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023a. [Language agent tree search unifies reasoning acting and planning in language models](#). *CoRR*, abs/2310.04406.
- Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. 2023b. [Least-to-most prompting enables complex reasoning in large language models](#). In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

A Additional Related Work

We provide a broader related work and the relative position of our work here.

Datasets for Text-to-SQL One of the prominent datasets utilized for Text-to-SQL is Spider 1.0 (Yu et al., 2018). Subsequent works also created various variants like Spider-Realistic (Deng et al., 2021), Spider-DK (Gan et al., 2021), and Spider-SS (Gan et al., 2022). To make the datasets more realistic and domain-specific, various new datasets like BIRD (Li et al., 2023), BookSQL (Kumar et al., 2024), and MultiSQL (Li et al., 2024a). Some works have also focused on specific domains with datasets like MIMICSQL (Wang et al., 2020b), BioMedSQL (Koretsky et al., 2025), and LogicCat (Liu et al., 2025). To combine various challenges and create a complex industry-scale dataset, Spider 2.0 (Lei et al., 2025) was proposed, and we evaluate our framework on this dataset in our work.

Automating Text-to-SQL Pre-LLM works (Wang et al., 2020a; Lin et al., 2020; Yin et al., 2020; Arcadinho et al., 2022) majorly relied on fine-tuning pre-trained language models on a set of training data. The advent of LLMs introduced basic constraint-decoding and prompting methods (Scholak et al., 2021; Trummer, 2022; Liu et al., 2023). Some works focused on improving in-context examples (Pourreza and Rafiei, 2023; Nan et al., 2023; Wang et al., 2024a), utilizing retrieval-augmented generation (Guo et al., 2023; Talaei et al., 2024; Shen et al., 2024), and inference-time scaling (Lee et al., 2025; Deng et al., 2025; Pourreza et al., 2025). With more compute, some works also show the effectiveness of fine-tuning small and large LLMs for text-to-SQL (Pourreza and Rafiei, 2024; Thorpe et al., 2024; He et al., 2025; Yao et al., 2025). Recent works have also proposed the use of multi-agent frameworks (Xie et al., 2024; Cen et al., 2025; Wang et al., 2024b, 2025a) to tackle complex SQL queries. Utilization of semantic verifiers Sawhney et al. (2025) have also been explored lately. In our work, we work on similar directions by proposing a simple multi-agent framework PEXA for the Spider 2.0 benchmark.

Analogy to Software Testing Software testing provides the conceptual backbone for the formulation of our agentic framework PEXA. Classical works formalize software-testing as specification, coverage, and evidence aggregation rather than

exhaustive verification, and emphasize designing suites of targeted tests to characterize system behavior (Reid, 2005; Pressman, 2005). Subsequent works have unified these ideas into rigorous coverage criteria, automated test execution, and oracle-based evaluation for determining correctness from multiple test outcomes (Ammann and Offutt, 2008). Our work also draws inspiration from the concept of executable specifications, central to test-driven development and agile engineering (Beck, 2003). Structural and data-flow testing works also show how carefully chosen, small test units provide maximal information about program semantics (Rapps and Weyuker, 1985). Overall, we root our conceptual design choices deeply in software testing.

Parallelization in LLMs Decomposition in planning (Zhou et al., 2023b; Khot et al., 2023; Wang et al., 2023a) improves model performance; however, the co-dependent plans demand sequential execution and can increase the latency. Newer approaches (Saha et al., 2024; Lu et al., 2025) have explored parallel execution-based planning, however only in limited settings like evaluation and constrained generation where subtasks are more explicit. Other works (Parekh et al., 2025c; Pan et al., 2025) explore dynamic planning of model reasoning methods to improve search. Sequential reasoning (Wei et al., 2022) and various multi-path breadth search approaches (Yao et al., 2023a; Besta et al., 2024; Zhou et al., 2023a) iteratively explore each node, which increases latency. On the other side, inference-time scaling approaches (Wang et al., 2023b; Lightman et al., 2024) that utilize temperature for multi-path search offer little control over the diversity in approaches, rendering them inefficient. In our work, we propose parallel exploration, which offers the benefits of simple single-step parallel executable decomposition and a better guided multi-path search in an efficient, low-latency setting.

B Implementation Details

Here, we describe the details of our implementation, including the prompts used for the different agents and the hyperparameters.

B.1 PEXA Prompt Details

Our PEXA pipeline comprises three major agents with distinctive roles. Similar to other unconstraining works (Parekh et al., 2025b), one of the advantages of our division is focused tasks for each

agent and the reduction of constraints in the model prompts. We discuss some distinctions and complete prompts for each agent/component below.

Planner We provide the prompt utilized for the Planner in Figure 3. Specifically, we describe the tools available, provide extensive guidelines as well as both positive and negative in-context examples, and final test question information. Note that the prompt is short and the LLM is encouraged to think and plan for the main question.

Test Case Generator We provide the prompt utilized for the Test Case Generator in Figure 4. We provide the main instructions first with detailed steps. Then, we provide an extensive guidelines for writing SQL queries in the dialect, and domain-specific instructions. We provide some examples for diverse SQL queries and batched exploration queries. We then provide the table information with table names. Finally, we provide any external knowledge if present and the main question/probe from the planner. We ask the LLM to provide two kinds of SQL queries - one which answers the question, and the other which is used to explore the database for more information. This is the only place where we provide the long table information in entirety and encourage the Generator LLM to utilize its long context reasoning.

SQL Proposer We provide the prompt utilized for the Test Case Generator in Figure 5. For the SQL proposer, we start with similar SQL writing and domain-specific guidelines as for the Test Case Generator. Next, we provide the main question. After this, we provide all the probes from the test case generator, categorized into four clusters: (1) Exploratory - These are used to explore the database and provide additional information. (2) Failure - These SQL queries failed and provide useful information for how not to write the final SQL. (3) Successful - These SQL queries succeeded and provide information for what are other ways to answer the question. (4) Final SQL queries - These are the most important and chosen by the test case generator as the most promising SQL queries for answering the question. Based on all this information, we ask the Proposer LLM to write the final SQL based on its long-horizon reasoning.

Semantic Verifier Finally, we provide the prompt utilized for the Semantic Verifier in Figure 6. We provide the three major steps involved in semantic verification: (1) Back-translation of the

Hyperparameter	Value
Number of workers	6
Max Planner Iterations	20
Max Testcase Iterations	15
Max Proposer Iterations	20
LLM temperature	0.3
Verifier temperature	1.0
Requests per second	0.5
Max retries	15
Timeout	120s
Execution Format	csv
Execution Max Length	500
Execution Max Rows	3

Table 5: Hyperparameter values for PEXA.

SQL query into natural language, (2) Comparison of translated query and original question to find semantic intent, and (3) Response generation with explanation. We then provide the original question, the LLM-written SQL query, and the execution results for its reference. Finally, we also provide some specific guidelines to steer its behavior.

B.2 Code Implementation

We provide the hyperparameters set for the PEXA runs in Table 5. We use LangGraph⁶ for implementing our LLM agent. To avoid endless LLM loops, we limit the max number of iterations per LLM agent (as highlighted in the table). We utilize API from OpenAI and Anthropic for calling respective LLMs.

C Additional Results and Analyses

Here we provide additional results and analyses to support our main results.

C.1 Latency analysis for sequential vs. parallelization

To objectively evaluate the speed gains achieved by our parallelization, we conduct a theoretical latency analysis for sequential vs. parallelization with PEXA. First, we build a theoretical model for studying the latency of PEXA architecture. Specifically, we split the total latency into the latency of each component as

$$t_{total} = t_{plan} + t_{generate} + t_{propose}$$

SQL Proposer operates the same way for both sequential and parallel execution and we will focus only on t_{plan} and $t_{generate}$ for our analysis.

⁶<https://www.langchain.com/langgraph>

Component	Average Latency (s)
Planner	18.9
Test Case Generator	20.4
SQL Executor	2.4
SQL Proposer	51.8
Semantic Verifier	10.6

Table 6: Component-wise latency analysis using o3 as the LLM in PEXA.

Mode	Total Latency (s)
Sequential	680
Parallel	351

Table 7: Theoretical model latencies for PEXA in sequential vs. parallel modes.

In the sequential case, the latency term of $t_{plan}^s + t_{generate}^s$ can be written as

$$t_{plan}^s + t_{generate}^s = \sum_{i=1}^{k_1} \left(t_{plan}^i + \sum_{j=1}^{k_{i2}} t_{generate}^{ij} \right)$$

where t_{plan}^i and $t_{generate}^{ij}$ indicates the time for making the i -th plan and executing it using the j -th strategy. Here k_1 is the total number of plans and k_{i2} indicates the total execution depth tried for this i -th plan.

In a similar way, when run in parallel as in PEXA, in sequential this term can be written as

$$t_{plan}^p + t_{generate}^p = \max_{i=1}^{k_1} t_{plan}^i + \max_{i=1, j=1}^{i=k_1, j=k_{i2}} t_{generate}^{ij}$$

Now, to objectify empirically, we measure the average latency of each component, as shown in Table 6. Using average conservative values for $k_1 = 3$ and k_2 sampled from normal distribution with $\mu = 8$ and $\sigma = 2$, we can estimate the time for the sequential and parallel cases of PEXA as shown in Table 7. The above values are chosen based on empirical estimation of the actually observed ones. For parallel case, we do account for communication delay with a liberal 5-20% per max operation.

Overall, we note how parallel exploration, theoretically, brings about a latency improvement of nearly 2x relative to sequential exploration of the same search space. In practical scenarios, this gain is slightly higher owing to lower communication delays and early stopping owing to the parallel search.

Iteration Number	SCR \uparrow	OR \downarrow	# TC \downarrow
1.0	78-86%	25%	5.0
2.0	86-91%	24%	4.2
2.1	85-87%	23%	4.0
2.2	84-96%	20-21%	3.9
2.3	90-91%	15-17%	3.6
3.0	90-94%	13-16%	3.7
3.1	88-92%	13-22%	3.7
3.2	87-95%	13-21%	3.3
4.0	91-97%	15-21%	3.5
4.1	92-95%	13-14%	3.1

Table 8: Tracking self-containment rate (SCR), overlap rate (OR), and number of proposed test cases (# TC) by the planner across iterations of prompt-based steering. The LLM used for Planner for these iterations was o3-mini and number of runs was 3 per iteration.

C.2 Deep-dive into Planner

In this analysis, we provide additional studies to highlight how our Planner agent operates. While the larger goal of Planner is to reduce the user query into smaller test cases, it is more than a simple decomposition. The generated test cases are aimed to "over-generate" and can have overlap across each other - but the main goal remains to semantically holistically cover the main user query. To steer the model towards this desired behavior, we rely on two in-context examples and guidance-based prompting. To verify the quality of the Planner across iterations, we track metrics like self-containment ratio, overlap ratio, and the number of proposed test cases. Ideally, we want a high self-containment rate with the lowest overlap ratio and number of proposed test cases. We measured the performance on these metrics utilizing Claude Sonnet-4. We provide the iterative improvement of the Planner (each iteration is an improved prompt setup) in Table 8. Clearly, our guidance-based steering improved the quality of the generated plans by the Planner. Some of the updated definitions of our test cases / probes are provided below for reference:

- Probes are simple natural language queries to provide basic information about the data to the proposer for writing the final SQL
- Keep your probes simple and focused on exploring the data. They should be answerable by simple SQL queries
- Make sure each probe is independent and self-contained and does not depend on previous probes. Do not include any probes for combin-

ing results or just for an extra computation/filtering operation.

C.3 Confidence Error Bounds

Owing to the high cost of executing multiple runs end-to-end, we could not provide strong confidence bounds for all models and runs. However, across four runs for PEXA and ReFoRCE, the confidence error bounds were 0.9% and 1.1% EX, respectively. This indicates how the variation across runs for PEXA is comparable to ReFoRCE and other previous works.

C.4 Component-wise Ablations - Implementation Details

We discussed and provided component-wise ablations in § 4.2. Here, we provide additional implementation details for each ablation for additional information:

- w/o Plan-time parallelization: removes the parallelization of generating self-sufficient independent plans from the Planner, which leads to sequential planning. The test case generator is also run one at a time for this version. The other components remain the same.
- w/o Test-time parallelization, removes the single-step multi-path search in the Test Case Generator. This leads to the exploration of one SQL at a time. The other components remain the same.
- w/o the Semantic Verifier, removes this tool from PEXA. Thus, the agent only relies on self knowledge and syntactic verification for feedback on its SQL. The other components remain the same.
- w/o Proposer, removes the SQL Proposer and asks the Planner to directly propose the final SQL. The Planner still has access to the other tools for verification and feedback. The other components remain the same.

C.5 Controlling efficiency with smaller LLMs

Apart from controlling the branching factor, we provide another way to improve model efficiency - by replacing the LLMs for different sub-components with smaller ones. To this end, we provide a comprehensive table of using a combination of o3-mini and o3 as the LLMs for the three different components of our pipeline on 100 datapoints from

Plan	Generate	Propose	EX	Wall Time
o3	o3	o3	42	5.55
o3-mini	o3	o3	42	5.01
o3	o3-mini	o3	41	3.95
o3	o3	o3-mini	39	4.77
o3-mini	o3-mini	o3	32	3.47
o3-mini	o3	o3-mini	35	4.36
o3	o3-mini	o3-mini	31	3.50
o3-mini	o3-mini	o3-mini	28	2.99

Table 9: Performance and latency (in mins) analysis for mix-and-match in PEXA with four different LLMs.

Statistic	Correct	Incorrect
# Output Tokens	19,631	29,312
# Generation Tokens	6,653	9,823
# Reasoning Tokens	12,977	19,488
# LLM Calls	16.74	22.63
# Planner Generates Plans	1.26	1.41
# Parallel Plans	7.06	7.2
# Probes Generated	29.2	41.9
# SQL Executions	8.99	13.48
# Semantic Verifications	1.09	1.32
Planner Recursion Reached	0%	5%
Generator Recursion Reached	2.8%	5.4%
Proposer Recursion Reached	0.8%	6%

Table 10: Averaged statistics per datapoint for PEXA run on Spider 2.0-Snow dataset using the o3 model as the LLM.

Spider 2.0-Snow in Table 9. As noted, replacing a single component in a pure o3 pipeline with o3-mini can improve the efficiency by 20% while suffering a slight drop relatively. On the other hand, if 2-3 components used are with smaller LLMs, then a more drastic performance drop can be noticed. Overall, we believe, the Proposer and Test Case Generator are the major components that need a stronger LLM to do long-context and long-horizon reasoning.

C.6 Characterization of PEXA runs

Here, we provide some basic statistics of PEXA using o3 on the Spider 2.0-Snow dataset to characterize some insights about the model runs. We split these statistics on successful/correct runs vs. incorrect ones and detail them in Table 10. First, we note how incorrect runs need 50% more tokens and cost than correct ones. They also need 35% more LLM calls. This is mainly owing to the LLM Agent trying to find a solution or needing more reasoning steps to solve the problem at hand. To improve efficiency for such runs, one can reduce the recursion limits for the different components.

At the same time, we note that the number of

Method	Snow	
	EX	EX@4
PEXA	45.7	49.5
PEXA w/ Gold Schema	47.2	50.8

Table 11: Performance comparison for PEXA with and w/o Gold Schema.

LLM calls by PEXA is on the higher end, indicating higher model costs. In our best opinion, it is really difficult to balance and improve the Pareto optimality across all three of cost, latency, and performance. In our work, we focus on improving the Pareto optimality for latency and performance, while our model costs do increase as an after-effect.

Next, we note that the number of planner calls and plans generated is in the range of 1.25-1.4 and 7-7.2, respectively, providing insights for the level of decomposition done for problem solving. While planner statistics are similar for correct/incorrect runs, in terms of execution, incorrect runs are 45-50% more expensive as measured by the number of probes and SQL executions.

Finally, we note the rate of recursion limit hits. For correct runs, it's low, touching about 3%. However, for incorrect runs, this number touches 5-6% consistently for all agents in PEXA. This indicates that a part of the failure of the incorrect runs lies in the limited reasoning steps and possibly can be improved by expending more resources.

C.7 Experiments with Gold Schema

To study the impact of gold schema, and schema linking in general, we run PEXA with gold schema and compare it without the gold schema in Table 11. We note how the improvement by using the gold schema for both metrics is minimal, capped at about 1.5%. This aligns with our error analysis as well in § 4.2, and we conclude that schema linking by itself is not a major issue to improve performance. Instead, we posit that logical reasoning and long-context reasoning are better improvement areas.

C.8 Qualitative Examples

To provide more insight where PEXA improves the existing models, we provide some qualitative samples in Table 12 and Table 13.

Example 1 Here, we note that Spider-Agent suffers from using the wrong table and completely misses the final results. On the other hand, Re-

FoRCE and PEXA use the right table/schema; but ReFoRCE has additional entries and looser filters. Additionally, Spider-Agent also uses station id to identify unique NOAA stations; while ReFoRCE and PEXA correctly use a combination USAF and WBAN for this identification. Finally, the main logic issue is in the data filtering step. Spider-Agent only utilizes "precipitation" > 0 as the main filter. ReFoRCE improves this by adding filters like "prcp" IS NOT NULL and "prcp" <> 99.99. PEXA further improves over both by adding an additional filter of "stn" <> '999999' - which it procures after deep database exploration to be a placeholder station number.

Example 2 First, in terms of query structure and readability, Spider-Agent uses nested subqueries and multiple CTEs, which makes it hard to follow. ReFoRCE improves readability by modularizing unions, while PEXA is the most logically separated, with two explicit CTEs (engaged_7d, engaged_2d). Creating these separate CTEs also helps to ensure better date filtering. Spider-Agent uses implicit joins without clear separation of logic, while ReFoRCE and PEXA use LATERAL FLATTEN joins more explicitly, making semantics clearer. Finally, the major distinction which leads to undercounting for Spider-Agent and ReFoRCE, relative to PEXA is the additional redundant filter/casting of date introduced in these SQL queries. Instead, PEXA picks the date from the table name and avoids adding additional filters.

C.9 Case Study of Parallelization

In Table 14, we demonstrate an end-to-end case study for how PEXA utilizes the multiple agents to explore one datapoint. Overall, the trajectory has two planning/decomposition components from the planner. In the first plan, two subplans are executed in parallel. The first subplan is explored in 5-7 different manners and the final SQL is written for it. For the second one, two execution strategies are explored and used. In the second plan, one subplan is executed, for which four different explorations are conducted. Finally, the proposer uses all this information to draft final SQL queries. Multiple feedbacks from the SQL Executor help it to fix the query which finally is syntactically and semantically correct. Finally, the planner reformats it a bit and returns as the final SQL.

C.10 Updated SPIDER 2.0 Leaderboard

In Figure 7, we provide a snapshot highlighting PEXA’s state-of-the-art performance when it was updated on the main leaderboard for Spider 2.0. We also note how significantly PEXA outperforms all other models.

D Directions for Future Work

In our work, we provide a strong proof-of-concept for parallelized exploration to improve the Pareto optimality of performance-latency in text-to-SQL on Spider 2.0. As part of future work, it would be imperative to provide the generalizability of this paradigm across other text-to-SQL datasets like SPIDER (Yu et al., 2018) and BIRD (Li et al., 2023). To this end, exploring post-training of individual components of our agentic workflow would be required. Synthetic data generation methods (Duan et al., 2025; Parekh et al., 2025a) should also be explored to this end to support this post-training. At a broader scale, works can also explore utilizing this framework for broader set of tasks in code generation, tool-calling, and beyond (Wang et al., 2025b; Barres et al., 2025; Parekh et al., 2024).

You are an excellent SQL agent whose task is to manage complex SQL query writing for a given natural language question.

Tools Available to Agent:

- Database Exploration: Call the worker (generate_testcase tool) to generate SQL queries for probes. Probes are simple natural language queries to provide basic information about the data to the proposer for writing the final SQL.
- Propose Final SQL: Call the proposer (propose_final_sql tool) to integrate probes and synthesize the final SQL query. You should call the proposer only when you feel you have enough probes to generate the final SQL.
- Finish: If you are confident that the final SQL query is ready and the executed results look good, you can submit it (finish tool). Else you can generate more probes to guide the final SQL generation.

Guidelines:

- Keep your probes simple and focused on exploring the data. They should be answerable by simple SQL queries.
- Make sure each probe is independent and self-contained and does not depend on previous probes. Do not include any probes for combining results or just for an extra computation/filtering operation.
- Generate only as many probes as needed. Do not be redundant.

...

Failure Examples:

Question: "Suppose it is 2020 currently. For each customer who purchased on the website in December last year, find the days between their first visit in December and their first purchase in December. Also find what type of device did they make that first purchase?"

Probes: ["Retrieve a table of all customers who made purchases in December last year", "Retrieve a table of all customers who visited the website who also made purchases in December last year", "Retrieve a table of devices used by customers for their purchases. Assume there is a table which provides the customer id and device id and the purchase date", "Calculate the days between the first visit and the first purchase for customers in December last year"]

Explanation: The first probe does not specify the year. The second probe is not simplest as it combines two different aspects (visits and purchases) into one unit. The third probe assumes the existence of a table that may not be present in the database. The fourth probe combines probes 1 and 2 to calculate the days - thus, not self-contained. You should not write probes like the fourth.

...

Correct Examples:

Question: "Suppose it is 2020 currently. For each customer who purchased on the website in December last year, find the days between their first visit in December and their first purchase in December. Also find what type of device did they make that first purchase?"

Probes: ["Retrieve a table of all customers who made purchases in December last year (the current year is 2020)", "Retrieve a table of all customers who visited the website in December last year (the current year is 2020)", "Retrieve a table of devices used by customers for their purchases"]

...

Test Data

Question: As of December 31, 2022, list the top 10 active banks with assets exceeding \$10 billion, ranked by the highest percentage of uninsured assets, where the percentage is calculated as one minus the value of the '% Insured (Estimated)' variable from quarterly estimates. Provide the names of these banks and their respective percentages of uninsured assets. Another way to say it: As of December 31, 2022, list the top 10 active banks with assets exceeding \$10 billion, ranked by the highest percentage of uninsured assets, where the percentage is calculated as one minus the value of the '% Insured (Estimated)' variable from quarterly estimates. Provide the names of these banks and their respective percentages of uninsured assets.

Now based on the instructions and examples provided, generate the probes (using the generate_testcase tool) for the above question.

Figure 3: Prompt utilized for PEXA's Planner

You are an SQL expert. Your task is to generate as many diverse SQL queries as possible for the question below. Each SQL should use a different combination of tables, so that at least one will execute correctly.

Instructions:

1. Check Question Completeness: If the question is not self-contained or unanswerable, use the "not enough information" tool with an explanation.
2. Generate many Diverse SQL queries: If answerable, generate as many diverse SQL queries as possible (even if you are confident about your first choice). Each SQL should use a different set of tables. Do NOT simply use the same table with different joins, filters, or aliases and call them diverse. True diversity means using fundamentally different tables or table combinations, not just different ways of querying the same table. For each SQL, provide a short description of what it does and how it is different from the others. Submit this list of SQL queries to the "execute_sql" tool for execution validation.
3. Explore Table Schema: Make sure that you are entirely sure about the attribute values before writing the SQL. If you are unsure, you can explore table columns or values using the "execute_sql" tool (using regex or ILIKE functions). Set the exploration attribute to True for such queries. Batch and execute all such exploration queries in a single tool call. Do not explore tables one by one in separate tool calls.
4. Tool Usage: In every response, you must use one of the tools.

SQL Writing Guidelines:

- MUST stay faithful to the exact wording of the question. Prefer exact table/column matches over partial matches. If an exact-match table exists, do not union with partial matches; only explore alternates if no exact match is found. Examples: If the question mentions 'total functional expenses', only use tables which mention them exactly.
- MUST keep queries simple (Occam's razor). Avoid extra joins/filters/rankings unless explicitly requested. Examples: If the question asks for "average measurement in a month", do not average across hours — average across days.
- MUST pay careful attention to any External Knowledge hints provided below; treat them as guidance for table/column.
- SHOULD prefer inner joins (which naturally drop NULLs) unless the question requires outer joins. In general, remove rows with NULLs unless told otherwise. ...

Dialect, Naming, and Qualification

- MUST use Snowflake SQL. Enclose all identifiers (databases, schemas, tables, columns, aliases) in double quotes.
- MUST fully qualify table names using the format: "<DATABASE>."<SCHEMA>."<TABLE>". To reference INFORMATION_SCHEMA, use "<DATABASE>."<INFORMATION_SCHEMA>."<TABLES>" instead of INFORMATION_SCHEMA.TABLES or anything else.
- ...

Examples

Example 1: (Diverse SQL queries)

Question: Retrieve the weather for all active days at an XYZ school for 2022.

Tool Call: execute_sql([('description': "Uses Table 1 and Table 2: Joins school active days with weather data.", 'sql_query': "SELECT ...", 'exploration': False), ('description': "Uses Table 4 and Table 3: Alternative join for weather info.", 'sql_query': "SELECT ...", 'exploration': False)])

...

Table Info:

Table full name:

FINANCE_ECONOMICS.CYBERSYN.BANK_FOR_INTERNATIONAL_SETTLEMENTS_ATTRIBUTES Column name: COUNTERPARTY_GEO_NAME Type: TEXT Description: Country or country group, representing the counterparty geography for the variable

...

Table Names:

The table structure information is ({{database name: {schema name: [table name]}}):
{'FINANCE_ECONOMICS': {'CYBERSYN': ...

External Knowledge: ...

Question: Retrieve a table of all banks with their identifiers, names, total assets, the '% Insured (Estimated)' value, and the reporting date for the quarter ending '2022-12-31'.

Figure 4: Prompt utilized for PEXA's Test Case Generator

You are an SQL expert. Your task is to generate a final SQL query for the given question. You are provided with a set of probes and corresponding SQL queries to provide you information about the database structure.

SQL Writing Guidelines:

- MUST stay faithful to the exact wording of the question. Do not add extra tables/filters/grouping beyond what is asked. Examples: If the question asks about top rising terms, do not add top rising international terms ...
- MUST keep queries simple (Occam's razor). Avoid extra ranking (RANK = 1), FLOOR() on dates, redundant GROUP BY, or unnecessary unions unless explicitly requested. Examples: If the question asks for "overall score", do not average "overall direct" and "overall indirect"; use overall direct as overall score ...
- MUST try to be faithful to the Final Probes. Example: If the probe uses "DICOM_ALL" and results are non-null, prefer "DICOM_ALL" in the final SQL (not "DICOM_PIVOT" or others)

...

Dialect, Naming, and Qualification

- MUST use Snowflake SQL and enclose all identifiers (DBs, schemas, tables, columns, aliases) in double quotes.
- MUST fully qualify tables: "<DATABASE>."<SCHEMA>."<TABLE>". To reference INFORMATION_SCHEMA, use "<DATABASE>."<SCHEMA>."<TABLE>" instead of INFORMATION_SCHEMA.TABLES or anything else.
- Preserve output case (uppercase/lowercase/mixed) as stored; do not LOWER()/UPPER() results.

...

Here are the details:

Main Question: As of December 31, 2022, list the top 10 active banks with assets exceeding \$10 billion, ranked by the highest percentage of uninsured assets, where the percentage is calculated as one minus the value of the '% Insured (Estimated)' variable from quarterly estimates. Provide the names of these banks and their respective percentages of uninsured assets.

Another way to say it: As of December 31, 2022, list the top 10 active banks with assets exceeding \$10 billion, ranked by the highest percentage of uninsured assets, where the percentage is calculated as one minus the value of the '% Insured (Estimated)' variable from quarterly estimates. Provide the names of these banks and their respective percentages of uninsured assets.

Probes and SQL Queries:

Exploratory Probes:

I. Probe: Retrieve a count of banks that reported total assets exceeding 10,000,000,000 (ten billion) dollars for the quarter ending '2022-12-31'.

SQL: SELECT "VARIABLE", "VARIABLE_NAME" FROM "FINANCE_ECONOMICS"."CYBERSYN"."FINANCIAL_INSTITUTION_ATTRIBUTES" WHERE LOWER("VARIABLE_NAME") LIKE '%total asset%';

Executed Result (truncated):

VARIABLE,VARIABLE_NAME

ASSET5,Average Total Assets

ASSET,Total Assets

...

Failure Probes:

...

Successful Probes:

...

Final SQL queries (pay more attention here):

...

Now use these information to generate a final SQL query that answers the question. The final SQL query should be a complete and valid SQL query that can be executed on the database.

Figure 5: Prompt utilized for PEXA's Final SQL Proposer

You are a semantic verifier. Your task is to verify the semantic correctness of a SQL query against a natural language question.

Here are the steps you should follow:

1. Back-translate the SQL query: Convert the SQL query (conditioned on the executed result) into natural language query that captures what the SQL query is trying to achieve. This should be a faithful representation of the SQL query and the executed result in natural language.
2. Comparison for Semantic Verification: Compare the back-translated natural language query (from part 1) with the original question. The back-translated query should capture the same intent and meaning as the original question.
 - If the back-translated query matches the original question, it indicates that the SQL query is semantically correct.
 - If the back-translated query does not match the original question, it indicates that the SQL query is semantically incorrect.
3. Return a response: Your final response should be a structured JSON of three fields - "correct", "explanation", and "back_translated_query". The "back_translated_query" should always be the query back-translated from step 1. Based on the comparison in step 2:
 - If the SQL query is semantically correct: "correct" = true and "explanation" = <Provide a one line explanation of why the back-translated query matches the original question.>
 - If the SQL query is semantically incorrect: "correct" = false and "explanation" = "<Give one line explanation of what is the difference between the back-translated query. Add one line for what can be potential improvement to fix this issue.>"

Original Question: As of December 31, 2022, list the top 10 active banks with assets exceeding \$10 billion, ranked by the highest percentage of uninsured assets, where the percentage is calculated as one minus the value of the '% Insured (Estimated)' variable from quarterly estimates. Provide the names of these banks and their respective percentages of uninsured assets.

SQL Query: WITH combined as SELECT ...

Executed Result:

Bank,Asset,% Insured (Estimated), ...

Some final guidelines for your semantic matching:

- Be extra careful with complex regex and try to ensure they are written correctly. If not provide an explanation to improve the regex.
 - Check the COALESCE statements carefully. Verify that they are not redundant the default values are not incorrect. We want to keep the queries simple and avoid any redundant/overthought statements.
 - Check the filters carefully. Ensure that there are no redundant filters which are unrelated to the original question. For example, if the question asks for trees not marked as dead, the SQL should only have filters for status not 'dead'. We want to avoid additional status checks like 'stump', 'cut down', 'removed' etc.
- ...

Now please follow the steps above and return your response in the specified JSON format.

Figure 6: Prompt utilized for PEXA's Semantic Verifier

Example 1	
Question	Which weather stations in Washington State recorded more than 150 rainy days in 2023 but fewer rainy days compared to 2022? Defining a “rainy day” as one having precipitation greater than zero millimeters and not equal to 99.99. Only include stations with valid precipitation data.
Spider-Agent	<pre>WITH rainy_counts AS (SELECT "station_id", DATE_PART('year', "date") AS "year", COUNT_IF("precipitation" > 0) AS "rainy_days" FROM NOAA_DATA.NOAA_GSOD.GSOD_DAILY WHERE "state" = 'WA' AND DATE_PART('year', "date") IN (2022, 2023) GROUP BY "station_id", DATE_PART('year', "date")), counts_2022 AS (SELECT "station_id", "rainy_days" FROM rainy_counts WHERE "year" = 2022), counts_2023 AS (SELECT "station_id", "rainy_days" FROM rainy_counts WHERE "year" = 2023) SELECT s."station_id", s."name" FROM counts_2023 c23 JOIN counts_2022 c22 ON c23."station_id" = c22."station_id" JOIN NOAA_DATA.NOAA_GSOD.STATIONS s ON s."station_id" = c23."station_id" WHERE c23."rainy_days" > 150 AND c23."rainy_days" < c22."rainy_days";</pre>
ReFoRCE	<pre>WITH rainy_2023 AS (SELECT "stn", "wban", COUNT(*) AS "rainy_days_2023" FROM NOAA_DATA.NOAA_GSOD."GSOD2023" WHERE "prcp" IS NOT NULL AND "prcp" <> 99.99 AND "prcp" > 0 GROUP BY "stn", "wban"), rainy_2022 AS (SELECT "stn", "wban", COUNT(*) AS "rainy_days_2022" FROM NOAA_DATA.NOAA_GSOD."GSOD2022" WHERE "prcp" IS NOT NULL AND "prcp" <> 99.99 AND "prcp" > 0 GROUP BY "stn", "wban"), counts AS (SELECT r23."stn", r23."wban", r23."rainy_days_2023", r22."rainy_days_2022" FROM rainy_2023 r23 INNER JOIN rainy_2022 r22 ON r23."stn" = r22."stn" AND r23."wban" = r22."wban") SELECT s."usaf" AS "station_usaf", s."wban" AS "station_wban", TRIM(s."name") AS "station_name", counts."rainy_days_2023", counts."rainy_days_2022" FROM counts JOIN NOAA_DATA.NOAA_GSOD."STATIONS" s ON counts."stn" = s."usaf" AND counts."wban" = s."wban" WHERE s."state" = 'WA' AND counts."rainy_days_2023" > 150 AND counts."rainy_days_2023" < counts."rainy_days_2022" ORDER BY counts."rainy_days_2023" DESC NULLS LAST;</pre>
PEXA	<pre>WITH "rainy_2022" AS (SELECT g."stn" AS "usaf", g."wban" AS "wban", TRIM(s."name") AS "station_name", COUNT(*) AS "rainy_days_2022" FROM "NOAA_DATA"."NOAA_GSOD"."GSOD2022" g JOIN "NOAA_DATA"."NOAA_GSOD"."STATIONS" s ON g."stn" = s."usaf" AND g."wban" = s."wban" WHERE s."state" = 'WA' AND g."stn" <> '999999' AND g."prcp" IS NOT NULL AND g."prcp" <> 99.99 AND g."prcp" > 0 GROUP BY g."stn", g."wban", TRIM(s."name")), "rainy_2023" AS (SELECT g."stn" AS "usaf", g."wban" AS "wban", TRIM(s."name") AS "station_name", COUNT(*) AS "rainy_days_2023" FROM "NOAA_DATA"."NOAA_GSOD"."GSOD2023" g JOIN "NOAA_DATA"."NOAA_GSOD"."STATIONS" s ON g."stn" = s."usaf" AND g."wban" = s."wban" WHERE s."state" = 'WA' AND g."stn" <> '999999' AND g."prcp" IS NOT NULL AND g."prcp" <> 99.99 AND g."prcp" > 0 GROUP BY g."stn", g."wban", TRIM(s."name")) SELECT r23."usaf" AS "station_id", r23."station_name", r23."rainy_days_2023", r22."rainy_days_2022" FROM "rainy_2023" r23 JOIN "rainy_2022" r22 ON r23."usaf" = r22."usaf" AND r23."wban" = r22."wban" WHERE r23."rainy_days_2023" > 150 AND r23."rainy_days_2023" < r22."rainy_days_2022" ORDER BY r23."rainy_days_2023" DESC, r23."station_name";</pre>

Table 12: Qualitative Example 1 showcasing how PEXA improves over some of the previous approaches.

Example 2

Question	How many distinct pseudo users had positive engagement time in the 5-day period ending on January 7, 2021 at 23:59:59, but had no positive engagement time in the 2-day period ending on the same date (January 7, 2021 at 23:59:59) ?
Spider-Agent	<pre>WITH engagement AS (SELECT e."USER_PSEUDO_ID", \textbf{TO_DATE(e."EVENT_DATE", 'YYYYMMDD')}} AS "evt_date" FROM (SELECT * FROM GA4.GA4_OBFUSCATED_SAMPLE_ECOMMERCE.EVENTS_20210103 UNION ALL SELECT * FROM GA4.GA4_OBFUSCATED_SAMPLE_ECOMMERCE.EVENTS_20210104 UNION ALL SELECT * FROM GA4.GA4_OBFUSCATED_SAMPLE_ECOMMERCE.EVENTS_20210105 UNION ALL SELECT * FROM GA4.GA4_OBFUSCATED_SAMPLE_ECOMMERCE.EVENTS_20210106 UNION ALL SELECT * FROM GA4.GA4_OBFUSCATED_SAMPLE_ECOMMERCE.EVENTS_20210107) e, LATERAL FLATTEN (INPUT => e."EVENT_PARAMS") ep WHERE e."EVENT_NAME" = 'user_engagement' AND ep.value:"key" = 'engagement_time_msec' AND COALESCE(ep.value:"value": 'int_value'::INTEGER, 0) > 0), user_flags AS (SELECT "USER_PSEUDO_ID", MAX(CASE WHEN "evt_date" BETWEEN '2021-01-03' AND '2021-01-07' THEN 1 ELSE 0 END) AS seven_day_positive, MAX(CASE WHEN "evt_date" BETWEEN '2021-01-06' AND '2021-01-07' THEN 1 ELSE 0 END) AS two_day_positive FROM engagement GROUP BY "USER_PSEUDO_ID") SELECT COUNT(*) AS "distinct_pseudo_users" FROM user_flags WHERE seven_day_positive = 1 AND two_day_positive = 0;</pre>
ReFoRCE	<pre>WITH union_events AS (SELECT * FROM GA4.GA4_OBFUSCATED_SAMPLE_ECOMMERCE. "EVENTS_20210103" UNION ALL SELECT * FROM GA4.GA4_OBFUSCATED_SAMPLE_ECOMMERCE. "EVENTS_20210104" UNION ALL SELECT * FROM GA4.GA4_OBFUSCATED_SAMPLE_ECOMMERCE. "EVENTS_20210105" UNION ALL SELECT * FROM GA4.GA4_OBFUSCATED_SAMPLE_ECOMMERCE. "EVENTS_20210106" UNION ALL SELECT * FROM GA4.GA4_OBFUSCATED_SAMPLE_ECOMMERCE. "EVENTS_20210107"), engagement AS (SELECT t."USER_PSEUDO_ID" AS user_id, TO_DATE(t."EVENT_DATE", 'YYYYMMDD') AS event_date, f.value:"value"::int_value::INTEGER AS engagement_time_msec FROM union_events t, LATERAL FLATTEN(INPUT => t."EVENT_PARAMS") f WHERE t."EVENT_NAME" = 'user_engagement' AND f.value:"key"::STRING = 'engagement_time_msec' AND f.value:"value"::int_value::INTEGER > 0), users_5d AS (SELECT DISTINCT user_id FROM engagement WHERE event_date BETWEEN '2021-01-03' AND '2021-01-07'), users_2d AS (SELECT DISTINCT user_id FROM engagement WHERE event_date BETWEEN '2021-01-06' AND '2021-01-07') SELECT COUNT(*) AS distinct_users FROM users_5d WHERE user_id NOT IN (SELECT user_id FROM users_2d);</pre>
PEXA	<pre>WITH "engaged_5d" AS (SELECT DISTINCT "USER_PSEUDO_ID" FROM (SELECT "USER_PSEUDO_ID", "EVENT_PARAMS" FROM "GA4"."GA4_OBFUSCATED_SAMPLE_ECOMMERCE". "EVENTS_20210103" UNION ALL SELECT "USER_PSEUDO_ID", "EVENT_PARAMS" FROM "GA4"."GA4_OBFUSCATED_SAMPLE_ECOMMERCE". "EVENTS_20210104" UNION ALL SELECT "USER_PSEUDO_ID", "EVENT_PARAMS" FROM "GA4"."GA4_OBFUSCATED_SAMPLE_ECOMMERCE". "EVENTS_20210105" UNION ALL SELECT "USER_PSEUDO_ID", "EVENT_PARAMS" FROM "GA4"."GA4_OBFUSCATED_SAMPLE_ECOMMERCE". "EVENTS_20210106" UNION ALL SELECT "USER_PSEUDO_ID", "EVENT_PARAMS" FROM "GA4"."GA4_OBFUSCATED_SAMPLE_ECOMMERCE". "EVENTS_20210107") t, LATERAL FLATTEN(input => t."EVENT_PARAMS") ep WHERE ep.value:key::string = 'engagement_time_msec' AND COALESCE(ep.value:value:int_value,0)::number > 0), "engaged_2d" AS (SELECT DISTINCT "USER_PSEUDO_ID" FROM (SELECT "USER_PSEUDO_ID", "EVENT_PARAMS" FROM "GA4"."GA4_OBFUSCATED_SAMPLE_ECOMMERCE"."EVENTS_20210106" UNION ALL SELECT "USER_PSEUDO_ID", "EVENT_PARAMS" FROM "GA4"."GA4_OBFUSCATED_SAMPLE_ECOMMERCE"."EVENTS_20210107") t, LATERAL FLATTEN(input => t."EVENT_PARAMS") ep WHERE ep.value:key::string = 'engagement_time_msec' AND COALESCE(ep.value:value:int_value,0)::number > 0) SELECT COUNT(*) AS "distinct_pseudo_users" FROM "engaged_5d" WHERE "USER_PSEUDO_ID" NOT IN (SELECT "USER_PSEUDO_ID" FROM "engaged_2d");</pre>

Table 13: Qualitative Example 2 showcasing how PEXA improves over some of the previous approaches.

Leaderboard

Spider 2.0-Snow

Spider 2.0-DBT

Spider 2.0-lite

Spider 2.0-Snow is a self-contained text-to-SQL task that includes well-prepared database metadata and documentation, includes **547** examples, all hosted on *Snowflake*, which offers participants free quotas.

Methods with *-** use special settings (ground-truth tables) and are not included in the ranking. Since we continually check the accuracy of the evaluation metrics (while the questions remain fixed), the scores may change slightly over time.

Rank	Method	Score
1 Sep 26, 2025	PExA <i>Bloomberg - AI Engineering group</i>	70.20
2 Aug 8, 2025	WindAgent + Claude-4-Sonnet <i>MeiTuan AI For FinData</i>	61.43
3 Sep 16, 2025	PAI-DataSurfer Agent <i>Alibaba Cloud Computing Platform</i>	60.33
4 Jul 19, 2025	AutoLink + DeepSeek-R1 <i>HUST VLR Lab</i> [Wang et al. '25]	54.84
5 Jul 7, 2025	Meituan-agent <i>Meituan FinData Intelligence</i>	45.34

Figure 7: Snapshot of PEXA's performance compared to other methods as of November 15, 2025.

Table 14: Detailed Case Study of PEXA’s exploration for a single datapoint.

Question	I want to know the GitHub project URLs for the top 3 most downloaded PyPI packages based on download count. First, extract PyPI package metadata including name, version, and project URLs. Filter these URLs to only include those that link to GitHub repositories. Use a regular expression to clean the GitHub URLs by removing unnecessary parts like 'issues', 'pull', 'blob', and 'tree' paths, keeping only the main repository URL. For packages with multiple versions, use only the most recent version based on upload time. Join this data with download metrics to determine the most downloaded packages. Return only the cleaned GitHub repository URLs (without quotation marks) for the top 3 packages by total download count, ensuring that only packages with valid GitHub URLs are included in the results.
Planner Probes 1	['Retrieve a table listing PyPI package metadata with the columns: package name, version, project_urls (or project_url), and upload_time.', 'Retrieve a table listing each PyPI package together with its total download count across all versions.']
Test Case 1	Retrieve a table listing PyPI package metadata with the columns: package name, version, project_urls (or project_url), and upload_time.
Execution 1	<p>1. SQL: SELECT "name" AS "package_name", "version", "project_urls", "upload_time" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" WHERE "name" IS NOT NULL LIMIT 100; Execution Output (truncated): package_name,version,project_urls,upload_time sparganthis-vim,0.1.15,[],1720561327687084 sparganthis-vim,0.1.15,[],1720561419667601 safa,0.0.1,[],1720564065701497</p> <p>2. SQL: WITH "downloads" AS (SELECT "project" AS "name", "file":"version"::string AS "version", MIN("timestamp") AS "first_download_ts" FROM "PYPI"."PYPI"."FILE_DOWNLOADS" GROUP BY "project", "file":"version"::string) SELECT d."name" AS "package_name", d."version", d."project_urls", d."upload_time" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" d JOIN "downloads" f ON d."name" = f."name" AND d."version" = f."version" LIMIT 100; Execution Output (truncated): package_name,version,project_urls,upload_time segments-ai,1.0.1,[],1659525962095717 pyrish,0.3.1,[],1517356387930046 odoo9-addon-stock-quant-manual-assign,9.0.1.0.0.99.dev20,[],1500179644597289</p> <p>3. SQL: SELECT d."name" AS "package_name", d."version", d."project_urls", d."upload_time" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" d JOIN "PYPI"."INFORMATION_SCHEMA"."TABLES" t ON t."TABLE_NAME" = 'DISTRIBUTION_METADATA' LIMIT 100; Execution Output (truncated): package_name,version,project_urls,upload_time sparganthis-vim,0.1.15,[],1720561327687084 sparganthis-vim,0.1.15,[],1720561419667601 safa,0.0.1,[],1720564065701497</p> <p>4. SQL: SELECT "project" AS "package_name", "file":"version"::string AS "version", NULL AS "project_urls", MIN("timestamp") AS "upload_time" FROM "PYPI"."PYPI"."FILE_DOWNLOADS" GROUP BY "project", "file":"version"::string LIMIT 100; Execution Output (truncated): package_name,version,project_urls,upload_time bs-solevaluator,0.1.1.,1649607460000000 django-graphql-auth,0.1.9.,1706310942000000 col-labi,0.0.1.,1699984505000000</p>
Execution 2	<p>1. SQL: SELECT "name" AS "package_name", "version", "project_urls" AS "project_urls", "upload_time" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" LIMIT 10; Execution Output (truncated): package_name,version,project_urls,upload_time sparganthis-vim,0.1.15,[],1720561327687084 sparganthis-vim,0.1.15,[],1720561419667601 safa,0.0.1,[],1720564065701497</p>

Table 14 (continued)

Exploration 3	<p>1. SQL: SELECT COUNT(*) AS cnt_non_empty FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" WHERE "project_urls" IS NOT NULL AND ARRAY_SIZE("project_urls") > 0; Execution Output (truncated): CNT_NON_EMPTY 42893</p> <p>2. SQL: SELECT "name" AS "package_name", "version", "project_urls", "upload_time" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" WHERE "project_urls" IS NOT NULL AND ARRAY_SIZE("project_urls") > 0 LIMIT 10; Execution Output (truncated): package_name,version,project_urls,upload_time renovosolutions.aws-cdk-aspects-security-group,1.0.214,"[""Source, https://github.com/RenovoSolutions/cdk-aspects-library-security-group.git""]",1661773668691461 connectome-interpretor,0.5.0,"[""Documentation, https://connectome-interpretor.readthedocs.io/en/latest/"" , ""Larva connectome example, https://colab.research.google.com/drive/1VIMNFBp7dCgN5XOQ9vzvzPaqb80BGPZx4?usp=sharing"" , ""Adult connectome example (FAFB), https://colab.research.google.com/drive/1ECUagWN-r2rnKyfcYgtR1oG8Lox8m8BW?usp=sharing""]",1709899013955375 oomongo,1.1,"[""HomePage, https://github.com/lcctoor/arts/tree/main/arts/oomongo""]",1709687704051481 selectolax,0.3.21,"[""Source code, https://github.com/rushter/selectolax""]",1710085659536925 sweetrpg-model-core,0.0.154,"[""Documentation, https://sweetrpg.github.io/model-core""]",1709610651959794 bolero,0.0.19,"[""Documentation, https://bolero.readthedocs.io/"" , ""Source, https://github.com/lhqing/bolero"" , ""Home-page, https://github.com/lhqing/bolero""]",1709615613500068 microagent,1.7.3,"[""Homepage, https://github.com/scailer/microagent"" , ""Documentation, https://microagent.readthedocs.io/en/stable/"" , ""Repository, https://github.com/scailer/microagent.git"" , ""Changelog, https://github.com/scailer/microagent/blob/master/CHANGELOG.rst""]",1710257591115486 pysorters,0.5.6,"[""Source, https://github.com/yolam1/pysorters""]",1709687967377114 prot2d,0.4.1,"[""Repository, https://github.com/ConstantinCarl/prot2d"" , ""Documentation, https://constantincarl.github.io/prot2d/"" , ""Zenodo, https://doi.org/10.5281/zenodo.10674045"" , ""Example-Collab, https://colab.research.google.com/drive/17u0twE81kYYspNFsdXUHrCyP33hj0dO6?usp=sharing""]",1710169269545664 rcplus_alloy_common,3.7.0,"[""Repository, https://github.com/ringier-data/rcplus-alloy-lib-py-common""]",1709301162301115</p> <p>3. SQL: SELECT COUNT(*) FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" WHERE "project_url" IS NOT NULL; Failure Message: SQL execution generated an error: ##ERROR##000904 (42000): SQL compilation error: error line 3 at position 6 invalid identifier ""project_url"" Please correct the query.</p>
Finish	<pre>SELECT "name" AS "package_name", "version", "project_urls", "upload_time" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA";</pre>
Test Case 2	Retrieve a table listing each PyPI package together with its total download count across all versions.
Execution 1	<p>1. SQL: SELECT "project" AS "package_name", COUNT(*) AS "total_downloads" FROM "PYPI"."PYPI"."FILE_DOWNLOADS" GROUP BY "project" ORDER BY "total_downloads" DESC; Execution Output (truncated): package_name,total_downloads s3transfer,2962234 attrs,1787211 colorama,1567249</p> <p>2. SQL: SELECT dm."name" AS "package_name", COUNT(*) AS "total_downloads" FROM "PYPI"."PYPI"."FILE_DOWNLOADS" fd JOIN (SELECT DISTINCT "name" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA") dm ON fd."project" = dm."name" GROUP BY dm."name" ORDER BY "total_downloads" DESC; Execution Output (truncated): package_name,total_downloads wheel,1481586 tabulate,1478960 python-dateutil,1415163</p>
Finish	<pre>SELECT "project" AS "package_name", COUNT(*) AS "total_downloads" FROM "PYPI"."PYPI"."FILE_DOWNLOADS" GROUP BY "project" ORDER BY "total_downloads" DESC;</pre>
Planner Probes 2	["Retrieve 10 sample rows from the PyPI distribution metadata where project_urls contains 'github.com', returning package name, version, project_urls, and upload_time."]
Test Case 3	Retrieve 10 sample rows from the PyPI distribution metadata where project_urls contains 'github.com', returning package name, version, project_urls, and upload_time.

Table 14 (continued)

<p>Exploration 1</p>	<p>1. SQL: SELECT "name", "version", "project_urls", "upload_time" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" WHERE "project_urls"::string ILIKE '%github.com%' LIMIT 10; Execution Output (truncated): name,version,project_urls,upload_time track-viz,0.5.0,"[""Changelog, https://github.com/JulienMBABD/track-viz/releases"", ""Documentation, https://track-viz.readthedocs.io"", ""Repository, https://github.com/JulienMBABD/track-viz""]",1666370862718894 searchor,2.3.2,"[""Homepage, https://github.com/ArjunSharda/Searchor"", ""Bug Tracker, https://github.com/ArjunSharda/Searchor/issues""]",1665751033856419 jpeglib,0.11.3,"[""Homepage, https://pypi.org/project/jpeglib"", ""Documentation, https://jpeglib.readthedocs.io/en/latest"", ""Source, https://github.com/martinbenes1996/jpeglib""]",1666259913352607</p> <p>2. SQL: SELECT DISTINCT dm."name", dm."version", dm."project_urls", dm."upload_time" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" dm JOIN "PYPI"."PYPI"."FILE_DOWNLOADS" fd ON dm."name" = fd."project" WHERE dm."project_urls"::string ILIKE '%github.com%' LIMIT 10; Execution Output (truncated): name,version,project_urls,upload_time iron-toolbox,1.0.45,"[""Bug Tracker, https://github.com/IronTrainers/iron_data_toolbox/issues""]",1686764499727015 matplotlib,3.6.0rc2,"[""Documentation, https://matplotlib.org"", ""Source Code, https://github.com/matplotlib/matplotlib"", ""Bug Tracker, https://github.com/matplotlib/matplotlib/issues"", ""Forum, https://discourse.matplotlib.org"", ""Donate, https://numfocus.org/donate-to-matplotlib""]",1661585954791952 cramjam,2.8.2,"[""homepage, https://github.com/milesgranger/pyrus-cramjam"", ""documentation, https://docs.rs/cramjam/latest/cramjam"", ""repository, https://github.com/milesgranger/pyrus-cramjam""]",1709410989630295</p> <p>3. SQL: SELECT dm."name", dm."version", dm."project_urls", dm."upload_time" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" dm JOIN "PYPI"."INFORMATION_SCHEMA"."TABLES" t ON t."TABLE_NAME" = 'DISTRIBUTION_METADATA' AND t."TABLE_SCHEMA" = 'PYPI' WHERE dm."project_urls"::string ILIKE '%github.com%' LIMIT 10; Execution Output (truncated): name,version,project_urls,upload_time BottleSessions,21.9.18,"[""repo, https://github.com/Glocktober/BottleSessions"", ""overview, https://github.com/Glocktober/BottleSessions/blob/main/README.md""]",1632027763679760 PTX-now,2.1,"[""Bug Tracker, https://github.com/pypa/sampleproject/issues""]",1631016956804108 aafragpy-serkol,0.8.45,"[""Bug Tracker, https://github.com/skoldobskiy/aafragpy/issues""]",1632248257186784</p> <p>4. SQL: SELECT DISTINCT dm."name", dm."version", dm."project_urls", dm."upload_time" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" dm JOIN "PYPI"."PYPI"."FILE_DOWNLOADS" fd ON dm."name" = fd."project" JOIN "PYPI"."INFORMATION_SCHEMA"."TABLES" t ON t."TABLE_NAME" = 'DISTRIBUTION_METADATA' AND t."TABLE_SCHEMA" = 'PYPI' WHERE dm."project_urls"::string ILIKE '%github.com%' LIMIT 10; Execution Output (truncated): name,version,project_urls,upload_time magpylib,4.5.0rc0,"[""Bug Tracker, https://github.com/magpylib/magpylib/issues"", ""Changelog, https://github.com/magpylib/magpylib/blob/master/CHANGELOG.md"", ""Documentation, https://magpylib.readthedocs.io/en/latest"", ""Repository, https://github.com/magpylib/magpylib.git""]",1702456277638981 duckdb,0.5.2.dev1120,"[""Documentation, https://duckdb.org/docs/api/python/overview"", ""Source, https://github.com/duckdb/duckdb/blob/master/tools/pythonpkg"", ""Issues, https://github.com/duckdb/duckdb/issues"", ""Changelog, https://github.com/duckdb/duckdb/releases""]",1666636910274206 augraphy,8.2.3,"[""Bug Tracker, https://github.com/sparkfish/augraphy/issues""]",1682936295648019</p>
<p>Finish</p>	<pre>SELECT "name", "version", "project_urls", "upload_time" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" WHERE "project_urls"::string ILIKE '%github.com%' LIMIT 10;</pre>
<p>Planner</p>	<p>Call Proposer</p>

Table 14 (continued)

Proposer 1	<pre>-- Assumptions & reasoning in comments:-- 1."PYPI.PYPI.DISTRIBUTION_METADATA" keeps one row per (package, version) and holds anARRAY variant column "project_urls" -- whose elements are strings like"Source, https://github.com/org/repo" (possibly ending in .git or deeper paths).--2. We only want the *latest* version per package (highest "upload_time").-- 3. Weonly keep packages whose latest metadata contains at least one GitHub URL.-- 4.Using that latest-metadata list, we flatten the array, extract the first GitHub URLcontained in every element and-- clean it so that only the canonical repo rootremains (https://github.com/<owner>/<repo>), removing:-- trailing pathfragments such as /issues, /pull/123, /blob/main/README.md, /tree/master, etc.-- a trailing ".git"-- a trailing slash-- 5. We join this per-packagecleaned URL to the download totals derived from "PYPI.PYPI.FILE_DOWNLOADS".-- 6.Finally, we return the cleaned GitHub URLs (no quotes around them in SQL output) forthe top 3 packages by total download count.WITH "downloads" AS (-- totaldownloads for every package (all versions) SELECT "project"AS "package_name", COUNT(*) AS"total_downloads" FROM "PYPI"."PYPI"."FILE_DOWNLOADS" GROUP BY"project"),-- latest metadata row per package (most recentupload_time)"latest_metadata" AS (SELECT "name" AS"package_name", "project_urls", ROW_NUMBER() OVER (PARTITION BY"name" ORDER BY "upload_time" DESC) AS "rn" FROM"PYPI"."PYPI"."DISTRIBUTION_METADATA" WHERE "project_urls" IS NOT NULLAND "project_urls"::string ILIKE '\%github.com%\'),'latest_with_urls" AS (SELECT lm."package_name", lm."project_urls" FROM "latest_metadata" lmWHERE lm."rn" = 1 -- keep only the most-recent version perpackage),-- flatten project_urls array and pull out raw GitHubURLs"flattened" AS (SELECT lwu."package_name",TRIM(f.value::string) AS "url_element" FROM "latest_with_urls" lwu,LATERAL FLATTEN(input => lwu."project_urls") f),"raw_github_urls" AS (SELECT "package_name", -- extract the *first* substring that lookslike a GitHub URL REGEXP_SUBSTR("url_element",\texttt{'https?://github\\.com/[^\\s,\\}]+'})} AS "raw_url" FROM "flattened"WHERE REGEXP_LIKE("url_element", 'https?://github\\.com/'),-- clean theGitHub URLs down to repo root"cleaned_urls" AS (SELECT DISTINCTrgu."package_name", -- 1) keep only scheme+owner/repo, discardingeverything afterwards -- 2) drop trailing .git if present -- 3) droptailing slash REGEXP_REPLACE(REGEXP_REPLACE(LOWER(rgu."raw_url"), -- normalise case\texttt{'^https?://github\\.com/([^/]+/[^/]+)?/?.*\$', -- capture owner/repoonly 'https://github.com/\\\\1')},'\\\\.git/?\$', -- remove trailing.git (with optional /) ' ') AS "cleaned_url" FROM"raw_github_urls" rgu WHERE rgu."raw_url" IS NOT NULL),-- attachdownload counts and pick the main URL per package (if multiple URLs, picklexicographically smallest)"package_with_downloads" AS (SELECTd."package_name", d."total_downloads", MIN(cu."cleaned_url")AS "cleaned_url" -- deterministic pick per package FROM "downloads" dJOIN "cleaned_urls" cu ON d."package_name" = cu."package_name" GROUPBY d."package_name", d."total_downloads"),-- top-3 by downloadcount"top3" AS (SELECT * FROM "package_with_downloads" ORDER BY"total_downloads" DESC LIMIT 3)SELECT "cleaned_url"FROM"top3"ORDER BY "total_downloads" DESC;</pre>
SQL Executor	SQL executed, but no valid results could be extracted. There seems to be some logical error in the SQL query. Please correct the query.

Table 14 (continued)

<p>Proposer 2</p>	<pre>-- Goal: GitHub repository URLs for the 3 most-downloaded PyPI packages(by total file downloads)-- Steps----- 1. Get themost-recent metadata row (highest upload_time) per package.-- 2. FLATTEN theproject_urls (array / object) and keep only those elements that mentiongithub.com.-- 3. Extract the raw GitHub URL substring from every element, thennormalise it to the canonical-- repo root "https://github.com/<owner>/<repo>"(remove /issues, /blob/..., /tree/..., *.git, etc.).-- 4. Aggregate to one(deterministic) cleaned URL per package.-- 5. Join with total download counts andreturn the top-3 packages that have a valid GitHubURL.----- WITH latest_metadata AS (SELECT "name" AS "package_name","project_urls", ROW_NUMBER() OVER (PARTITION BY "name" ORDER BY"upload_time" DESC) AS "rn" FROM"PYPI"."PYPI"."DISTRIBUTION_METADATA" WHERE "project_urls" IS NOT NULL-- only rows that actually contain project_urls),latest AS (SELECT"package_name", "project_urls" FROM latest_metadata WHERE "rn" = 1-- keep only the newest version per package),flattened AS (-- explode thearray/object so each URL (or key-value) becomes its own row SELECTl."package_name", f.value::string AS "url_element" FROM latestl, LATERAL FLATTEN(input => l."project_urls") f WHERE f.value ISNOT NULL AND f.value::string ILIKE '%github.com%'),github_raw AS (-- pull out the first GitHub URL substring from each element SELECT DISTINCT"package_name", REGEXP_SUBSTR("url_element",'https?://github\\.com/[^\s)"\'']+' -- stops at space,), " or ')AS "raw_url" FROM flattened WHERE REGEXP_LIKE("url_element",'https?://github\\.com/'),cleaned_urls AS (-- 1) drop trailing .git(optional followed by /) -- 2) collapse to repo root by capturing justowner/repo and rebuilding the URL SELECT "package_name",REGEXP_REPLACE(REGEXP_REPLACE(LOWER("raw_url"), '\\\\.git/?\$', ''),-- step 1 '^https?://github\\.com/([^\s/]+/[^\s/]+)(?:/.*)?\$', --step 2 (capture) 'https://github.com/\\\\1') AS"cleaned_url" FROM github_raw WHERE "raw_url" IS NOTNULL),per_package_url AS (-- choose a single deterministic URL per package(lexicographically smallest) SELECT "package_name",MIN("cleaned_url") AS "cleaned_url" FROM cleaned_urls GROUP BY"package_name"),downloads AS (-- total file downloads (all versions) perpackage SELECT "project" AS "package_name", COUNT(*) AS"total_downloads" FROM "PYPI"."PYPI"."FILE_DOWNLOADS" GROUP BY"project"),ranked AS (SELECT d."total_downloads",u."cleaned_url" FROM downloads d JOIN per_package_url u ONd."package_name" = u."package_name" ORDER BY d."total_downloads" DESCLIMIT 3)SELECT "cleaned_url"FROM rankedORDER BY "total_downloads"DESC;</pre>
<p>SQL Executor</p>	<p>SQL execution generated an error ##ERROR##001003 (42000): SQL compilation error: syntax error line 33 at position 43 unexpected ']'. syntax error line 34 at position 10 unexpected 'AS'. Please correct the query.</p>

Table 14 (continued)

Proposer 3	<pre>-- GitHub repository URLs for the 3 most-downloaded PyPI packages ----- 1. Identify the most-recent metadata row for every package (by upload_time).-- 2. FLATTEN "project_urls" and keep only the elements that reference GitHub.-- 3. Extract the raw GitHub URL substring, then normalise it to the repo root--(https://github.com/<owner>/<repo>), removing trailing path fragments-- and a possible ".git".-- 4. Join these packages to their total download counts and pick the top 3.----- WITH "latest_metadata" AS (SELECT "name" AS "package_name", "project_urls", ROW_NUMBER() OVER (PARTITION BY "name" ORDER BY "upload_time" DESC) AS "rn" FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA" WHERE "project_urls" IS NOTNULL), "latest" AS (SELECT "package_name", "project_urls" FROM "latest_metadata" WHERE "rn" = 1 -- keep only the newest version per package), "flattened" AS (-- explode the project_urls array/object SELECT 1."package_name", f.value::string AS "url_element" FROM "latest" l, LATERAL FLATTEN(input => l."project_urls") f WHERE f.value IS NOT NULL AND f.value::string ILIKE '%github.com%'), "github_raw" AS (-- extract the GitHub URL substring (up to first whitespace) SELECT DISTINCT "package_name", REGEXP_SUBSTR("url_element", 'https://github\\.com/[^\s]+') AS "raw_url" FROM "flattened" WHERE REGEXP_LIKE("url_element", 'https?://github\\.com/')), "cleaned_urls" AS (-- normalise to canonical repo root and strip trailing .git SELECT "package_name", REGEXP_REPLACE(REGEXP_REPLACE(LOWER("raw_url"), '\\\\.git/?\$', ''), -- drop .git'^https?://github\\.com/([^/]+/[^/]+)(/.*)?\$', -- keep owner/repo only 'https://github.com/\\1') AS "cleaned_url" FROM "github_raw" WHERE "raw_url" IS NOT NULL), "per_package_url" AS (-- deterministic single URL per package (lexicographically smallest) SELECT "package_name", MIN("cleaned_url") AS "cleaned_url" FROM "cleaned_urls" GROUP BY "package_name"), "downloads" AS (-- total file downloads per package (all versions) SELECT "project" AS "package_name", COUNT(*) AS "total_downloads" FROM "PYPI"."PYPI"."FILE_DOWNLOADS" GROUP BY "project"), "ranked" AS (SELECT d."total_downloads", u."cleaned_url" FROM "downloads" d JOIN "per_package_url" u ON d."package_name" = u."package_name" ORDER BY d."total_downloads" DESC LIMIT 3) SELECT "cleaned_url" FROM "ranked" ORDER BY "total_downloads" DESC;</pre>
SQL Executor	SQL executed, but no valid results could be extracted. There seems to be some logical error in the SQL query. Please correct the query.

Table 14 (continued)

<p>Proposer 4</p>	<pre>-- GitHub repository URLs for the three most-downloaded PyPI packages -- ----- 1. Find the latest (by upload_time) metadata row for every package.-- 2. From thoseros, FLATTEN the "project_urls" array / object and keep-- only elementsmentioning GitHub.-- 3. Extract the repository-root URL (https://github.com/<owner>/<repo>),-- stripping any trailing ".git","/issues", "/pull/...", etc.-- 4. Keep one deterministic URL per package, joinwith total download-- counts, and return the URLs of the top-3packages. ----- WITH latest_metadata AS (SELECT "name" AS"package_name", "project_urls", ROW_NUMBER() OVER (PARTITION BY"name" ORDER BY "upload_time" DESC) AS "rn" FROM"PYPI"."PYPI"."DISTRIBUTION_METADATA" WHERE "project_urls" IS NOT NULL),latest AS (SELECT "package_name", "project_urls" FROM latest_metadata WHERE "rn" = 1 -- newest version perpackage),flattened AS (SELECT l."package_name",f.value::string AS "url_element" FROM latest l, LATERALFLATTEN(input => l."project_urls") f WHERE f.value IS NOT NULL ANDf.value::string ILIKE '%github.com%'),cleaned_urls AS (/* 1)Extract the substring up to repo root (owner/repo). 2) Remove a trailing ".git" (with optional slash). Examples handled: -https://github.com/tensorflow/tensorflow.git -> https://github.com/tensorflow/tensorflow -https://github.com/pallets/flask/tree/main -> https://github.com/pallets/flask - Source,https://github.com/psf/requests/issues -> https://github.com/psf/requests*/ SELECT DISTINCT "package_name", /* step-A: repo rootsubstring */ REGEXP_SUBSTR(LOWER("url_element"),'https?://github\\\\.com/[^/]+/[^/]+') AS "root_url" FROMflattened),normalised_urls AS (SELECT "package_name",-- step-B: drop trailing .git and any trailing slashRTRIM(REGEXP_REPLACE("root_url", '\\\\.git\$', ''), '/') AS "cleaned_url" FROMcleaned_urls WHERE "root_url" IS NOT NULL),per_package_url AS (--choose a single deterministic URL per package (lexicographically smallest) SELECT"package_name", MIN("cleaned_url") AS "cleaned_url" FROMnormalised_urls GROUP BY "package_name"),downloads AS (SELECT"project" AS "package_name", COUNT(*) AS "total_downloads" FROM"PYPI"."PYPI"."FILE_DOWNLOADS" GROUP BY "project"),ranked AS (SELECT d."total_downloads", u."cleaned_url" FROM downloads d JOINper_package_url u ON d."package_name" = u."package_name" ORDER BYd."total_downloads" DESC LIMIT 3)SELECT "cleaned_url"FROMrankedORDER BY "total_downloads" DESC;</pre>
	<p>SQL Executor SQL query executed successfully!</p>
<p>Semantic Verifier The SQL query is semantically correct!</p>	

Table 14 (continued)

Planner Finish

```

-- GitHub repository URLs for the three most-downloaded PyPI packages
-----
-- 1. Find the latest (by upload_time) metadata row for every package.
-- 2. From those rows, FLATTEN the "project_urls" array / object and keep
-- only elements mentioning GitHub.
-- 3. Extract the repository-root URL ( https://github.com/<owner>/<repo> ),
-- stripping any trailing ".git", "/issues", "/pull/...", etc.
-- 4. Keep one deterministic URL per package, join with total download
-- counts, and return the URLs of the top-3 packages.
-----
WITH latest_metadata AS (
  SELECT "name" AS "package_name", "project_urls", ROW_NUMBER() OVER (PARTITION BY
    "name" ORDER BY "upload_time" DESC) AS "rn"
  FROM "PYPI"."PYPI"."DISTRIBUTION_METADATA"
  WHERE "project_urls" IS NOT NULL
),
latest AS (
  SELECT "package_name", "project_urls"
  FROM latest_metadata
  WHERE "rn" = 1 -- newest version per package
),
flattened AS (
  SELECT l."package_name", f.value::string AS "url_element"
  FROM latest l, LATERAL FLATTEN(input => l."project_urls") f
  WHERE f.value IS NOT NULL AND f.value::string ILIKE '%github.com%'
),
cleaned_urls AS (
  /*
  1) Extract the substring up to repo root (owner/repo).
  2) Remove a trailing ".git" (with optional slash).
  */
  SELECT DISTINCT "package_name", /* step-A: repo root substring */
    REGEXP_SUBSTR(LOWER("url_element"), 'https?://github\\.com/[^\s]+/[^\s]+') AS
    "root_url"
  FROM flattened
),
normalised_urls AS (
  SELECT "package_name", -- step-B: drop trailing .git and any trailing slash
    RTRIM(REGEXP_REPLACE("root_url", '\\.git$', ''), '/') AS "cleaned_url"
  FROM cleaned_urls
  WHERE "root_url" IS NOT NULL
),
per_package_url AS (
  SELECT "package_name", MIN("cleaned_url") AS "cleaned_url"
  FROM normalised_urls
  GROUP BY "package_name"
),
downloads AS (
  SELECT "project" AS "package_name", COUNT(*) AS "total_downloads"
  FROM "PYPI"."PYPI"."FILE_DOWNLOADS"
  GROUP BY "project"
),
ranked AS (
  SELECT d."total_downloads", u."cleaned_url"
  FROM downloads d
  JOIN per_package_url u ON d."package_name" = u."package_name"
  ORDER BY d."total_downloads" DESC LIMIT 3
)

SELECT "cleaned_url"
FROM ranked
ORDER BY "total_downloads" DESC;

```