MC-SQL: A Multi-Agent Collaborative and Multi-Reasoning-Chain Framework for Text-to-SQL

Peiwen He
School of Computer Science
South China Normal University
Guangzhou, China
2024023453@m.scnu.edu.cn

Mingkai Liu
School of Computer Science
South China Normal University
Guangzhou, China
2024023484@m.scnu.edu.cn

Mengchi Liu*
School of Computer Science
South China Normal University
Guangzhou, China
liumengchi@scnu.edu.cn

***Abstract*—In recent years, large language models (LLMs) have been increasingly adopted for Text-to-SQL tasks due to their strong emergent capabilities. However, most existing studies rely on single-strategy generation paradigms, which limits their ability to exploit the complementary strengths of different reasoning strategies, particularly in complex query scenarios. To address this limitation, this paper proposes MC-SQL, a Text-to-SQL framework that integrates multi-agent collaboration with multi–chain-of-thought generation. It consists of four cooperative agents: a Translator that alleviates column semantic ambiguity, a Pruner that compresses the database schema, a Generator that produces diverse SQL candidates through three distinct chains of thought, and a Post-processor that refines SQL based on execution feedback and selects the final output. By combining multi-strategy reasoning and agent collaboration, MC-SQL significantly improves the accuracy and robustness of SQL generation. Experimental results demonstrate that MC-SQL outperforms most single-strategy models across multiple benchmark datasets.**

***Index Terms*—Text-to-SQL, LLM Agent, Multi-Chain-of-Thought, Multi-Agent Collaboration, SQL Generation Robustness**

I. Introduction

With the rise of data-driven decision making, efficiently extracting information from massive databases has become a key concern [1]. SQL can quickly retrieve the required data and is widely used in areas such as business intelligence [2] and healthcare analysis [3]. However, using SQL requires certain expertise, making it difficult for non-expert users to obtain the information they need, thus creating a significant barrier to database access.

To address the "access barrier between users and databases," Text-to-SQL technology has emerged as a research hotspot. As shown in Figure 1, non-expert users can input everyday language, such as "What is the highest eligible free rate for K-12 students in the schools in Alameda County?" through a natural language interface (NLIDB). The system automatically generates SQL queries and retrieves information from the database, without requiring users to understand the internal processing (e.g., schema mapping, SQL generation, or underlying database operations). In other words, the task of Text-to-SQL is to establish a mapping from natural language to SQL, thereby simplifying user access to databases.

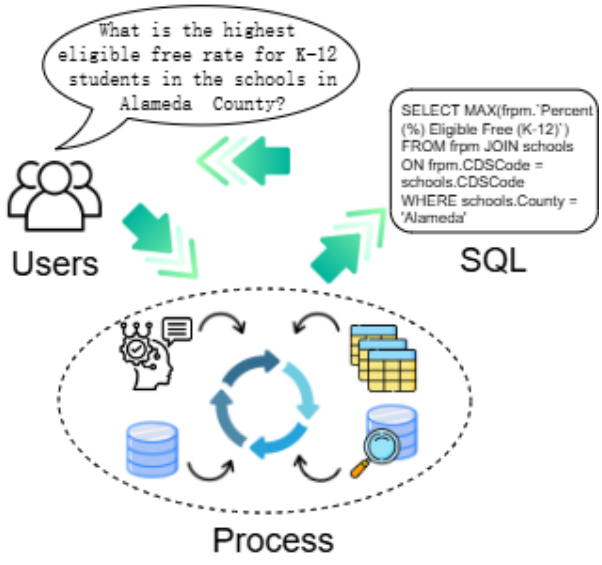

Fig. 1. Users obtain SQL queries using Text-to-SQL

In recent years, research on LLM-based Text-to-SQL techniques has flourished from multiple perspectives. For example, MAC-SQL [4] improves the generation accuracy of complex queries through a multi-agent collaborative framework, while TA-SQL [5] reduces hallucinations and enhances accuracy by aligning inputs before generation. Despite their advantages, most existing

Mengchi Liu* is the corresponding author.

Text-to-SQL models rely on a single-strategy generation mode, often resulting in homogenized problem-solving and limited robustness for complex queries.

In this paper, we explore Text-to-SQL methods leveraging diverse chains of thought. Specifically, we propose MC-SQL, which generates SQL via three reasoning chains. While components such as schema pruning, column enrichment, and execution-based refinement have been explored in prior work, MC-SQL uniquely combines multi-agent collaboration, parallel multi-chain reasoning, and post-selection of SQL candidates to exploit complementary strengths of different strategies. This design leads to significantly improved robustness and SQL generation quality, distinguishing MC-SQL from previous methods.

- MC-SQL employs multiple agents and three reasoning chains that generate multiple SQL candidates in parallel, followed by a post-processing selection mechanism to produce the final output. This multi-strategy approach enhances coverage of complex queries and improves overall SQL accuracy.
- While AST-based intermediate representations have been explored in prior Text-to-SQL research, in MC-SQL the Decoupled Chain operates in parallel with the Linear and Parallel Chains, serving as a structural stabilizer that complements the other chains and enhances multi-strategy reasoning robustness. The Linear Chain ensures sequential coherence, the Parallel Chain handles multi-branch queries, and the Decoupled Chain focuses on decoupling high-level logic from low-level syntax to reduce errors in complex SQL generation.

The rest of the paper is organized as follows. In Section 2, we review the development trends of Text-to-SQL and related work on LLM agents. Section 3 presents our proposed method in detail. Section 4 provides the experimental results and comparative analysis. Finally, we conclude our work in Section 5.

II. RELATED WORK

In this section, we review the related work relevant to our study, focusing on two main areas. First, we summarize the evolution of Text-to-SQL research, highlighting key stages, representative methods, and their strengths and limitations. Second, we discuss the development of Large Language Model (LLM) agents, emphasizing their architecture, capabilities, and applications in SQL generation tasks. These discussions provide the necessary background and motivation for our proposed multi-strategy LLM agent framework.

A. Text-to-SQL

Research on Text-to-SQL began in the 1990s and has gone through four main stages: rule-based, neural network-based, pre-trained model (PLM), and LLM stages [6]. The rule-based stage relies on semantic parsers and handcrafted rules. Although highly interpretable, it suffers from high manual cost and limited generalization [7]. The neural network stage improves performance by introducing end-to-end generation models but it still requires large amounts of annotated data and struggles with complex semantics [8]. The pre-trained model stage leverages large-scale corpora to enhance natural language understanding, but lacks sufficient structural reasoning ability and is sensitive to database schemas [9]. Recently, researchers turn to LLMs to exploit their emergent capabilities for performing complex reasoning tasks [10].

B. LLM Agent

LLM agents are autonomous systems driven by LLMs, capable of understanding natural language instructions, performing task planning, invoking external tools, and adjusting themselves based on feedback. Compared with traditional agents that rely on rules or reinforcement learning, LLM agents demonstrate superior adaptability to environments, reasoning capabilities, and cross-task generalization [11]. Typical frameworks include task parsing, planning and reasoning, tool usage, and self-reflection and correction, enabling them to handle complex tasks in zero-shot or few-shot scenarios and achieve continuous improvement over long-term tasks [12]. LLM agents have been widely applied to web automation, data analysis, code generation, and Text-to-SQL tasks, often leveraging multi-agent collaboration and self-optimization mechanisms to enhance performance [13].

Building on this, MAC-SQL introduces a multi-agent collaboration mechanism, where different LLM agents work together to generate, verify, and refine SQL queries, improving the accuracy and robustness of complex query generation. MAG-SQL [14] further incorporates soft schema linking and iterative subquery refinement mechanisms, enhancing the accuracy and robustness of SQL generation. However, existing Text-to-SQL methods based on LLM agents mostly rely on a single strategy, overlooking the complementarity between strategies. In this study, we propose a multi-strategy LLM agent framework that employs three distinct reasoning chains and achieves significant improvements on benchmark datasets.

III. MC-SQL FRAMEWORK

In Figure 2, we present MC-SQL, a novel multi-agent collaborative and multi–chain-of-thought Text-to-SQL framework that employs LLMs as four distinct agents—Translator, Pruner, Generator, and Post-processor—to achieve more robust Text-to-SQL parsing. In particular, the Generator adopts multiple chains of

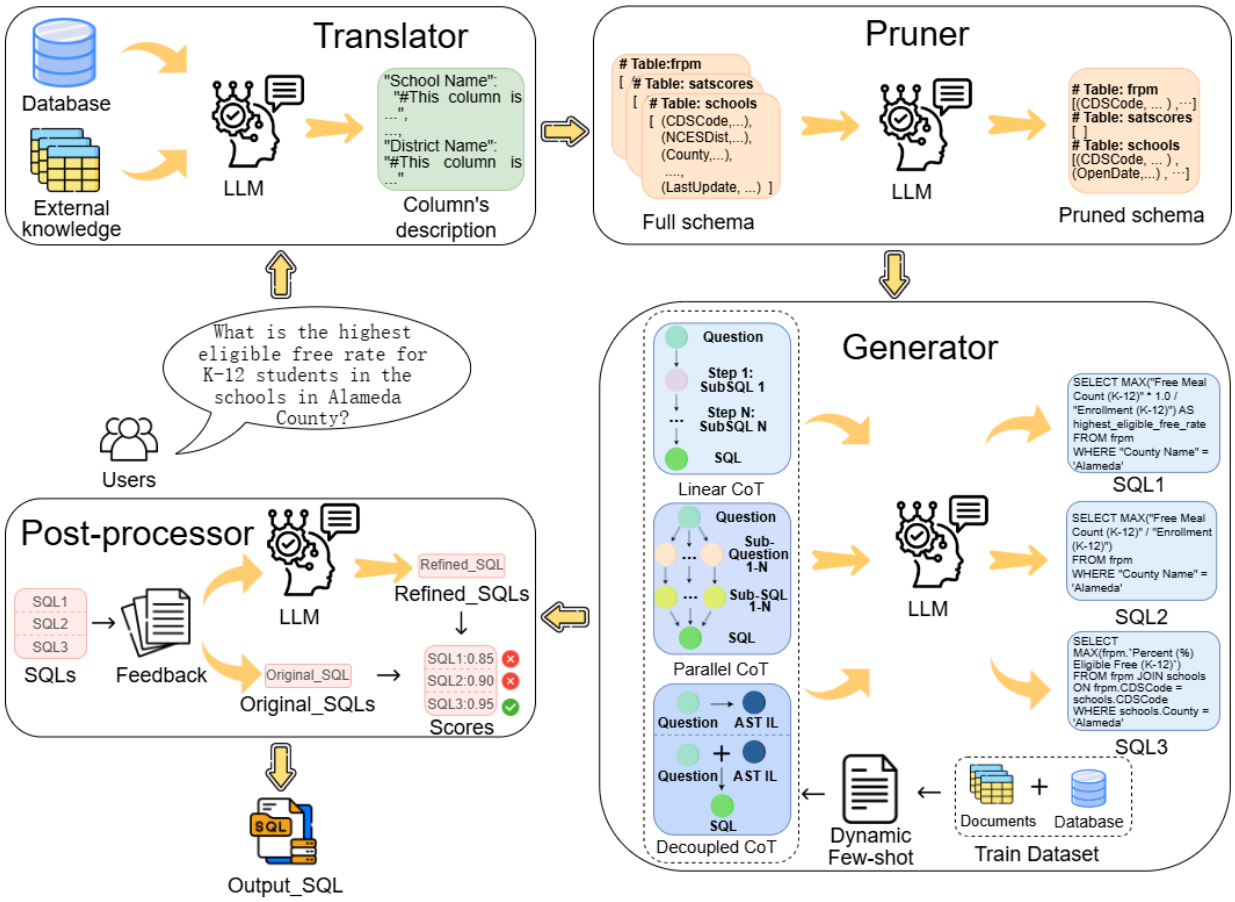

Fig. 2. The overview of our MC-SQL framework mainly consists of four components: (I) the Translator, which integrates database columns with external knowledge to generate column descriptions, reducing column ambiguity; (II) the Pruner, which simplifies complex database schemas into more concise versions to shorten context length; (III) the Generator, which produces three SQL queries using three reasoning chains combined with dynamic examples, achieving strategy complementarity; and (IV) the Post-Processor, which refines the SQL based on execution feedback and selects the final query.

thought to realize strategy complementarity, thereby improving the overall quality of the generated SQL. The following sections provide a detailed explanation of each component.

A. Translator

Given an input pair $X_{\text{translator}} = (S', K)$ where the database subschema $S' = \{T', C'\}$ (T' denotes the set of tables, C' denotes the set of columns), and K represents external knowledge providing supplementary semantic information for each column, the Translator agent aims to generate clear, standardized, and semantically complete descriptions for all columns in the schema. This reduces column-level ambiguity and enhances interpretability and accuracy in subsequent SQL generation.

The operation of the Translator agent can be formalized as

$$D = f_{\text{translator}}(S', K \mid M_t), \tag{1}$$

where D denotes the set of generated column descriptions, and $f_{\text{translator}}(\cdot \mid M_t)$ represents the function that invokes the LLM with a prompt template M_t.

The Translator agent is motivated by two factors: real-world column names often use abbreviations or inconsistent semantics, which can mislead LLMs if fed directly; and incorporating external knowledge K allows generation of unified, structured column descriptions, improving LLM understanding of database structure and overall parsing robustness.

B. Pruner

Given an input triplet $X_{\text{pruner}} = (S', D, Q)$, where the database subschema $S' = \{T', C'\}$, D denotes the set of column descriptions generated by the Translator agent providing clear and standardized semantic information for each column, and Q represents the user's natural language query, the Pruner agent aims to select the most relevant tables and columns from the full schema,

producing a pruned subschema $S'' \subseteq S'$. This reduces the reasoning burden for the Generator and enhances the efficiency and accuracy of subsequent SQL generation.

The operation of the Pruner agent can be formalized as:

$$S'' = f_{\text{pruner}}(S', D, Q \mid M_p), \tag{2}$$

where S'' denotes the pruned subschema, M_p is the prompt template for the Pruner, and $f_{\text{pruner}}(\cdot \mid M_p)$ represents the LLM-based function that decides which tables and columns should be retained by analyzing column descriptions and query intent.

The Pruner agent is motivated by two factors: real-world schemas often have many tables and columns, which can overload the LLM and introduce redundancy; and using the Translator's standardized column descriptions, the Pruner accurately selects relevant tables and columns, reducing the search space while preserving semantics, thus improving SQL generation efficiency and accuracy.

C. Generator

Given an input triple $X_{\text{Generator}} = (S'', D, Q)$, where S'' denotes the pruner output of the database subschema, D represents the set of column descriptions generated by the Translator, and Q is the natural language query, the Generator agent aims to produce accurate and executable SQL statements based on the schema, column descriptions, and query semantics. The Generator maps the input to outputs O_i for each reasoning chain:

$$O_i = f_{\text{generator}}(S'', D, Q \mid M_{g_i}), \quad i = 1, 2, 3 \tag{3}$$

where O_i denotes the SQL generated by the i-th reasoning chain, and M_{g_i} represents the corresponding prompt template for that chain. In this manner, the Generator can produce multiple SQL candidates for the same input, with each chain independently generating SQL from the input.

The Generator agent aims to produce accurate and executable SQL statements based on the pruned schema, column descriptions, and query semantics. Importantly, MC-SQL leverages three reasoning chains that operate in parallel, generating multiple SQL candidates for each input, which are later evaluated and selected by the Post-processor. This design ensures multi-strategy coverage and complementarity, going beyond single-strategy or linear generation approaches.

The current framework employs three reasoning chain strategies (as illustrated in Figure 2):

- **Linear Chain**: The reasoning and SQL generation proceed sequentially, where each step depends on the previous step's output, ensuring a clear logical order.
- **Parallel Chain**: The input query is initially decomposed into several independent sub-questions, each generating its corresponding sub-SQL independently. The final SQL is obtained by merging all sub-SQL statements, suitable for multi-branch queries.
- **Decoupled Chain**: SQL generation is divided into two steps, as illustrated in Figure 3. In the first step, the query is distilled into a high-level AST (Abstract Syntax Tree) intermediate language (IL) to construct a logical skeleton, separating logical reasoning from concrete implementation. This approach forces the LLM to commit to the logical structure before generating specific SQL dialect tokens. In the second step, the final SQL is produced by combining the query and the AST, decoupling syntax generation from logical reasoning. This strategy reduces task complexity and allows the LLM to focus separately on logic and syntax.

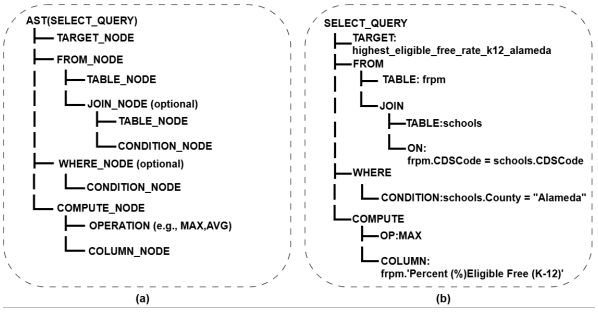

Fig. 3. The representation of the AST intermediate form: (a) illustrates the generic structure of an abstract syntax tree, while (b) presents the AST intermediate form instantiated from a specific example query.

D. Post-processor

The Post-processor agent aims to perform semantic validation, execution feedback correction, and multi-metric selection on the candidate SQL statements produced by the Generator, ensuring that the final SQL output satisfies logical, syntactic, and execution-level requirements, while achieving a *trade-off optimality* in terms of readability, execution efficiency, and maintainability.

Given the set of candidate SQL statements produced by the Generator: $O = \{O_1, O_2, \ldots, O_m\}$, the Post-processor first performs execution feedback correction. Each SQL statement is executed on the target database, and feedback is collected regarding syntax errors, execution time, and data quality. If a SQL statement fails to execute correctly or produces unexpected results, the feedback is combined with the original SQL and fed into the LLM to generate a corrected version that ensures syntactic correctness, logical consistency, and accurate query results. This allows the LLM to repair both syntactic and logical errors, while optimizing SQL

structure when necessary to improve execution efficiency and data quality.

After feedback correction, the Post-processor performs multi-metric evaluation and selection based on a set of rules and LLM-driven semantic verification mechanisms. This process can be formalized as:

$$\hat{O} = f_{\text{post}}(O, S' \mid M_{\text{post}}), \tag{4}$$

where $\hat{O}$ denotes the final SQL output, S' is the database subschema, M_{post} is the post-processing prompt template, and $f_{\text{post}}(\cdot)$ represents the Post-processor function that invokes the LLM for correction and multi-metric selection.

The Post-processor assesses SQL statements on Correctness, Efficiency, and Readability. Correctness ensures executability and logical validity; Efficiency favors simpler, faster queries; Readability promotes concise, maintainable SQL. By leveraging execution feedback and multi-metric evaluation, it corrects errors from the Generator and enhances logical consistency, robustness, and overall SQL quality in complex database scenarios.

IV. EXPERIMENT

A. Experimental Setup

1)Datasets: This study primarily evaluates and analyzes model performance on the Spider and BIRD datasets. Spider is a large-scale, cross-domain Text-to-SQL dataset [15], consisting of natural language questions paired with manually annotated SQL queries, covering over 200 databases. It emphasizes complex query generation and cross-domain generalization, including nested queries, JOINs, aggregations, and multi-table operations. BIRD is another large-scale [16], cross-domain benchmark, containing 12,751 natural language–SQL pairs across 95 large databases and 37 domains, totaling 33.4 GB. It requires models to generate semantically correct, executable, and efficient SQL queries in real-world database environments, challenging their ability to understand database schemas and handle noisy or inconsistent data.

2)Baselines: We evaluated our method on the BIRD and SPIDER datasets and compared it with several baselines: DeepSeek [17] leverages a reinforcement reasoning framework for efficient execution-driven SQL generation; DIN-SQL [18] decomposes the reasoning process into controllable sub-steps and uses self-correction to optimize complex queries; C3 [19] directly generates SQL using ChatGPT's zero-shot capability; DAIL-SQL [20] enhances accuracy and robustness in a dynamic few-shot setting through reinforcement learning; MAC-SQL employs a multi-agent collaborative framework to decompose tasks and improve SQL generation accuracy and efficiency; MAG-SQL combines soft schema linking with iterative sub-SQL refinement to enhance accuracy and consistency; and PURPLE [21] optimizes LLM SQL generation via reward mechanisms and reinforcement learning, thereby improving query accuracy and stability.

TABLE I
EXECUTION ACCURACY (EX) ON SPIDER (DEV, TEST)

Method	Dev	Test
C3 + ChatGPT	81.80	82.30
MAC-SQL + GPT-4	86.75	82.80
DIN-SQL + GPT-4	82.80	85.30
MAG-SQL+GPT-4	85.30	85.60
DAIL-SQL + GPT-4	84.40	86.60
PURPLE + GPT-4o	87.80	-
MC-SQL + GPT-4	87.20	85.80
MC-SQL + GPT-4o	87.50	86.20
MC-SQL + DeepSeek-R1	**88.20**	**87.10**

TABLE II
EXECUTION ACCURACY (EX) ON BIRD (DEV)

Method	Dev
DeepSeek	56.13
MAC-SQL + GPT-4	57.56
DIN-SQL + GPT-4	50.72
MAG-SQL+GPT-4	57.62
DAIL-SQL + GPT-4	54.76
PURPLE + GPT-4o	62.97
MC-SQL + GPT-4	59.85
MC-SQL + GPT-4o	64.12
MC-SQL + DeepSeek-R1	**66.43**

B. Overall Performance

Across both Spider and BIRD benchmarks, MC-SQL demonstrates strong and consistent performance gains. On Spider, it reaches 88.20%/87.10% (Dev/Test) with DeepSeek-R1, outperforming prior multi-agent and decomposition-based methods such as MAC-SQL, MAG-SQL, and DAIL-SQL, and even slightly surpassing strong proprietary baselines like PURPLE + GPT-4o. On the more challenging BIRD benchmark, MC-SQL further shows its advantage by achieving 66.43% on the Dev set with DeepSeek-R1, significantly exceeding existing approaches and maintaining competitive performance even with weaker backbones (e.g., 64.12% with GPT-4o). Overall, these results indicate that the performance improvements are not solely dependent on stronger base models, but are largely attributed to the MC-SQL framework itself—particularly its multi-candidate reasoning and selection mechanism—which enhances robustness, generalization, and scalability in complex, real-world text-to-SQL tasks.

TABLE III
ABLATION STUDY OF MC-SQL ON BIRD DEV DATASET

Method	Simple	Moderate	Challenging	All
MC-SQL	72.11	59.78	51.39	**66.43**
w/o Translator	69.84	56.99	48.61	63.95
w/o Pruner	68.97	55.70	48.61	63.04
w/o Generator	68.32	53.55	45.83	61.73
w/o Post-processor	68.32	55.48	47.92	62.52

TABLE IV
COMPARISON OF INDIVIDUAL COT METHODS ON BIRD DEV DATASET

Method	Simple	Moderate	Challenging	All
MC-SQL	72.11	59.78	51.39	**66.43**
Linear CoT	68.65	55.27	51.39	62.97
Parallel CoT	69.51	55.91	50.00	63.65
Decoupled CoT	68.32	53.55	45.83	61.73

C. Ablation Study

The ablation results in Table 3 indicate that each module contributes substantially to the overall performance, with the Generator module playing the most critical role. The results in Table 4 further confirm the strategy complementarity, showing that combining multi-agent collaboration with multiple reasoning chains significantly enhances SQL generation quality. Overall, the experiments clearly demonstrate that MC-SQL consistently outperforms both the ablated variants and the single-chain CoT methods across all query difficulty levels, exhibiting higher accuracy, stronger robustness, and better generalization capability.

V. CONCLUSION

In this paper, we propose MC-SQL, a multi-agent collaborative and multi-reasoning-chain framework for the Text-to-SQL task, which improves the accuracy and robustness of complex SQL generation through strategy complementarity and agent collaboration. MC-SQL decomposes the SQL generation task into multiple subtasks, handled collaboratively by modules such as Translator, Pruner, Generator, and Post-processor, and generates and refines SQL through multiple reasoning chains, effectively reducing errors and enhancing execution accuracy. Experiments on benchmark datasets including Bird and Spider demonstrate that MC-SQL achieves outstanding performance, with overall execution accuracy significantly surpassing most existing methods. Ablation studies further validate the complementarity of its modules and strategies. Future work can explore more dynamic module collaboration and reasoning chain designs, as well as adaptability across different language environments.

REFERENCES

[1] Y. Huang, J. Guo, W. Mao, et al., "Exploring the Landscape of Text-to-SQL with Large Language Models: Progresses, Challenges and Opportunities," arXiv preprint arXiv:2505.23838, 2025.

[2] M. A. B. Mohammed, M. Al-Okaily, D. Qasim, et al., "Towards an understanding of business intelligence and analytics usage: evidence from the banking industry," *International Journal of Information Management Data Insights*, vol. 4, no. 1, p. 100215, 2024.

[3] D. Mendhe, A. Dogra, P. S. Nair, et al., "AI-enabled data-driven approaches for personalized medicine and healthcare analytics," in *Proc. 2024 Ninth International Conference on Science Technology Engineering and Mathematics (ICONSTEM)*, IEEE, 2024, pp. 1-5.

[4] B. Wang, C. Ren, J. Yang, et al., "MAC-SQL: a multi-agent collaborative framework for text-to-SQL," 2024. [Online]. Available: https://arxiv.org/abs/2312.11242.

[5] G. Qu, J. Li, B. Li, et al., "Before generation, align it! A novel and effective strategy for mitigating hallucinations in text-to-SQL generation," *arXiv preprint arXiv:2405.15307*, 2024.

[6] X. Liu, S. Shen, B. Li, et al., "A survey of Text-to-SQL in the era of LLMs: Where are we, and where are we going?," *IEEE Transactions on Knowledge and Data Engineering*, 2025.

[7] F. Li and H. V. Jagadish, "NaLIR: An interactive natural language interface for querying relational databases," in Proc. 2014 ACM SIGMOD Int. Conf. Manage. Data, 2014, pp. 709–712.

[8] B. Bogin, J. Berant, and M. Gardner, "Representing schema structure with graph neural networks for text-to-SQL parsing," in Proc. 57th Annu. Meet. Assoc. Comput. Linguistics (ACL), 2019, pp. 4560–4565.

[9] J. Li, B. Hui, R. Cheng, et al., "Graphix-T5: Mixing pre-trained transformers with graph-aware layers for text-to-SQL parsing," in Proc. AAAI Conf. Artif. Intell., vol. 37, no. 11, 2023, pp. 13076–13084.

[10] E. R. Nascimento, G. M. Garcia, L. Feijó, et al., "Text-to-SQL meets the real-world," in Proc. 26th Int. Conf. Enterprise Inf. Syst., vol. 1, 2024, pp. 61–72.

[11] S. Shinn, Y. Lababa, et al., "Reflexion: Language agents with verbal reinforcement learning," in *Proc. NeurIPS*, 2023.

[12] Y. Chen, C. Xia, et al., "Agent-as-a-Judge: Evaluate agents with agents," in *Proc. ICLR*, 2024.

[13] H. Wang, M. Deng, et al., "AgentBench: Evaluating LLMs as agents," in *Proc. NeurIPS*, 2023.

[14] W. Xie, G. Wu, and B. Zhou, "MAG-SQL: Multi-agent generative approach with soft schema linking and iterative sub-SQL refinement for text-to-SQL," *arXiv preprint arXiv:2408.07930*, 2024.

[15] T. Yu, R. Zhang, K. Yang, et al., "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task," arXiv preprint arXiv:1809.08887, 2018.

[16] J. Li, B. Hui, G. Qu, et al., "Can LLM already serve as a database interface," *CoRR, abs/2305.03111*, 2023.

[17] D. Guo, D. Yang, H. Zhang, et al., "DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning," arXiv preprint arXiv:2501.12948, 2025.

[18] M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed in-context learning of Text-to-SQL with self-correction," Advances in Neural Information Processing Systems, vol. 36, pp. 36339–36348, 2023.

[19] Dong X, Zhang C, Ge Y, et al. C3: Zero-shot text-to-sql with chatgpt[J]. arXiv preprint arXiv:2307.07306, 2023.

[20] D. Gao, H. Wang, Y. Li, et al., "Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation," arXiv preprint arXiv:2308.15363, 2023.

[21] T. Ren, Y. Fan, Z. He, et al., "PURPLE: Making a Large Language Model a Better SQL Writer," in 2024 IEEE 40th International Conference on Data Engineering (ICDE), IEEE, 2024, pp. 15–28.